Lecture Handouts

CS501

Advance Computer Architecture

Table of Contents

Appendix A

Reading Material
Handouts

Summary

1. Introduction to FALSIM
2. Preparing source files for FALSIM
3. Using FALSIM
4. FALCON-A assembly language techniques

FALSIM

1. Introduction to FALSIM:

FALSIM is the name of the software application which consists of the FALCON-A assembler and the FALCON-A simulator. It runs under Windows XP.

FALCON-A Assembler:

Figure 1 shows a snapshot of the FALCON-A Assembler. This tool loads a FALCON-A assembly file with a (.asmfa) extension and parses it. It shows the parse results in an error log, lets the user view the assembled file's contents in the file listing and also provides the features of printing the machine code, an Instruction Table and a Symbol Table to a FALCON-A listing file. It also allows the user to run the FALCON-A Simulator.

The FALCON-A Assembler has two main modules, the 1st-pass and the 2nd-pass. The 1st-pass module takes an assembly file with a (.asmfa) extension and processes the file contents. It then creates a Symbol Table which corresponds to the storage of all program variables, labels and data values in a data structure at the implementation level. If the 1st-pass completes successfully a Symbol Table is produced as an output, which is used by the 2nd-pass module. Failures of the 1st-pass are handled by the assembler using its exception handling mechanism.

The 2nd-pass module sequentially processes the .asmfa file to interpret the instruction opcodes, register opcodes and constants using the symbol table. It then produces a list file with a .lstfa extension independent of successful or failed pass. If the pass is successful a binary file with a .binfa extension is produced which contains the machine code for the program in the assembly file.

FALCON-A Simulator:

Figure 6 shows a snapshot of the FALCON-A Simulator. This tool loads a FALCON-A binary file with a (.binfa) extension and presents its contents into different areas of the simulator. It allows the user to execute the program to a specific point within a time frame or just executes it, line by line. It also allows the user to view the registers, I/O port values and memory contents as the instructions execute.

FALSIM Features:

The FALCON-A Assembler provides its user with the following features:

Select Assembly File: Labeled as "**1**" in Figure 1, this feature enables the user to choose a FALCON-A assembly file and open it for processing by the assembler.

Assembler Options: Labeled as "**2**" in Figure 1.

- *Print Symbol Table*

This feature if selected writes the Symbol Table (produced after the execution of the 1st-pass of the assembler) to a FALCON-A list file with an extension of (.lstfa). The Symbol Table includes data members, data addresses and labels with their respective values.

- *Print Instruction Table*

This feature if selected writes the Instruction Table to a FALCON-A list file with an extension of (.lstfa).

List File: Labeled as "**3**", in Figure 1, the List File feature gives a detailed insight of the FALCON-A listing file, which is produced as a result of the execution of the 1st and 2nd-pass. It shows the Program Counter value in hexadecimal and decimal formats along with the machine code generated for every line of assembly code. These values are printed when the 2nd-pass is completed.

Error Log: The Error Log is labeled as "**4**" in Figure 1. It informs the user about the errors and their respective details, which occurs in any of the passes of the assembler.

Search: Search is labeled as "**5**" in Figure 1 and helps the user to search for a certain input with the options of searching with "**match whole**" and "**match any**" parts of the string. The search also has the option of checking with/without considering "**case-sensitivity**". It searches the List File area and highlights the search results using the yellow color. It also indicates the total number of matches found.

Start Simulator: This feature is labeled as "**6**" in Figure 1. The FALCON-A Simulator is run using the FALCON-A Assembler's Start Simulator option. The FALCON-A Simulator is invoked by the user from the FALCON-A Assembler. Its features are detailed as follows:

Load Binary File: The button labeled as "**11**" in Figure 6, allows the user to choose and open a FALCON-A binary file with a (.binfa) extension. When a file is being loaded into the simulator all the register, constants (if any) and memory values are set.

Registers: The area labeled as "**12**" in Figure 6. enables, the user to see values present in different registers before during and after execution.

Instruction: This area is labeled as "**13**" in Figure 6 and contains the value of PC, address of an instruction, its representation in Assembly, the Register Transfer Language, the op-code and the instruction type.

I/O Ports: I/O ports are labeled as "**14**" in Figure 6. These ports are available for the user to enter input operation values and visualize output operation values whenever an I/O operation takes place in the program. The input value for an input operation is given by the user before an instruction executes. The output values are visible in the I/O port area once the instruction has successfully executed.

Memory: The memory is divided into 2 areas and is labeled as "**15**" in Figure 6, to facilitate the view of data stored at different memory locations before, during and after program execution.

Processor's State: Labeled as "**16**" in Figure 6, this area shows the current values of the Instruction register and the Program Counter while the program executes.

Search: The search option for the FALCON-A simulator is labeled as "**17**" in Figure 6. This feature is similar to the way the search feature of the FALCON-A Assembler works. It offers to highlight the search string which goes as an input, with the "All " and " Part " option. The results of the search are highlighted in the color yellow. It also indicates the total number of matches.

The following is a description of the options available on the button panel labeled as "**18**" in Figure 6.

Single Step: "Single Step" lets the user execute the program, one instruction at a time. The next instruction is not executed unless the user does a "single step" again. By default, the instruction to be executed will be the one next in the sequence. It changes if the user specifies a different PC value using the Change PC option (explained below).

Change PC: This option lets the user change the value of PC (Program Counter). By changing the PC the user can execute the instruction to which the specified PC points.

Execute: By choosing this button the user is able to execute the instructions with the options of execution with/without breakpoint insertion (refer to Fig. 5). In case of breakpoint insertion, the user has the option to choose from a list of valid breakpoint values. It also has the option to set a limit on the time for execution. This "Max Execution Time" option restricts the program execution to a time frame specified by the user, and helps the simulator in exception handling.

Change Register: Using the Change Register feature, the user can change the value present in a particular register.

Change Memory Word: This feature enables the user to change values present at a particular memory location.

Display Memory: Display Memory shows an updated memory area, after a particular memory location other than the pre-existing ones is specified by the user.

Change I/O: Allows the user to give an I/O port value if the instruction to be executed requires an I/O operation. Giving in the input in any one of the I/O ports areas before instruction execution, indicates that a particular I/O operation will be a part of the program and it will have an input from some source. The value given by the user indicates the input type and source.

Display I/O: Display I/O works in a manner similar to Display Memory. Here the user specifies the starting index of an I/O port. This features displays the I/O ports stating from the index specified.

2. Preparing source files for FALSIM:

In order to use the FALCON-A assembler and simulator, FALSIM, the source file containing assembly language statements and directives should be prepared according to the following guidelines:

- The source file should contain ASCII text only. Each line should be terminated by a carriage return. The extension **.asmfa** should be used with each file name. After assembly, a list file with the original filename and an extension **.lstfa,** and a binary file with an extension **.binfa** will be generated by FALSIM.
- Comments are indicated by a semicolon (;) and can be placed anywhere in the source file. The FALSIM assembler ignores any text after the semicolon.
- Names in the source file can be of one of the following types:
- Variables: These are defined using the .equ directive. A value must also be assigned to variables when they are defined.
- Addresses in the "data and pointer area" within the memory: These can be defined using the .dw or the .sw directive. The difference between these two directives is that when .dw is used, it is not possible to store any value in the memory. The integer after .dw identifies the number of memory words to be reserved starting at the current address. (The directive .db can be used to reserve bytes in memory.) Using the .sw directive, it is possible to store a constant or the value of a name in the memory. It is also possible to use pointers with this directive to specify addresses larger than 127. Data tables and jump tables can also be set up in the memory using this directive.
- Labels: An assembly language statement can have a unique label associated with it. Two assembly language statements cannot have the same name. Every label should have a colon (:) after it.
- Use the .org 0 directive as the first line in the program. Although the use of this line is optional, its use will make sure that FALSIM will start simulation by picking up the first instruction stored at address 0 of the memory. (Address 0 is called the reset address of the processor). A jump [first] instruction can be placed at address 0, so that control is transferred to the first executable statement of the main program. Thus, the label first serves as the identifier of the "entry point" in the source file. The .org directive can also be used anywhere in the source file to force code at a particular address in the memory.
- Address 2 in the memory is reserved for the pointer to the Interrupt Service Routine (ISR). The .sw directive can be used to store the address of the first instruction in the ISR at this location.

- Address 4 to 125 can be used for addresses of data and pointers[1]. However, the main program must start at address 126 or less[2], otherwise FALSIM will generate an error at the jump [first] instruction.

- The main program should be followed by any subprograms or procedures. Each procedure should be terminated with a ret instruction. The ISR, if any, should be placed after the procedures and should be terminated with the iret instruction.

- The last line in the source file should be the .end directive.

- The .equ directive can be used anywhere in the source file to assign values to variables.

- It is the responsibility of the programmer to make sure that code does not overwrite data when the assembly process is performed, or vice versa. As an example, this can happen if care is not exercised during the use of the .org directive in the source file.

3. Using FALSIM:

- To start FALSIM (the FALCON-A assembler and simulator), double click on the FALSIM icon. This will display the assembler window, as shown in the Figure 1.

- Select one or both assembler options shown on the top right corner of the assembler window labeled as "2". If no option is selected, the symbol table and the instruction table will not be generated in the list (.lstfa) file.

- Click on the select assembly file button labeled as "1". This will open the dialog box as shown in the Figure 2.

- Select the path and file containing the source program that is to be assembled.

- Click on the open button. FALSIM will assemble the program and generate two files with the same filename, but with different extensions. A list file will be generated with an extension .lstfa, and a binary (executable) file will be generated with an extension .binfa. FALSIM will also display the list file and any error messages in two separate panes, as shown in Figure 3.

---

[1] Any address between 4 and 14 can be used in place of the displacement field in load or store instructions. Recall that the displacement field is just 5 bits in the instruction word.

[2] This restriction is because of the face that the immediate operand in the movi instruction must fit an 8-bit field in the instruction word.

- Double clicking on any error message highlights and displays the corresponding erroneous line in the program listing window pane for the user. This is shown in Figure 4. The highlight feature can also be used to display any text string, including statements with errors in them. If the assembler reported any errors in the source file, then these errors should be corrected and the program should be assembled again before simulation can be done. Additionally, if the source file had been assembled correctly at an earlier occasion, and a correct binary (.binfa) file exists, the simulator can be started directly without performing the assembly process.

- To start the simulator, click on the start simulation button labeled as "6". This will open the dialog box shown in Figure 6.

- Select the binary file to be simulated, and click open as shown in Figure 7.

- This will open the simulation window with the executable program loaded in it as shown in Figure 8. The details of the different panes in

- this window were given in section 1 earlier. Notice that the first instruction at address 0 is ready for execution. All registers are initialized to 0. The memory contains the address of the ISR (i.e., 64h which is 100 decimal) at location 2 and the address of the printer driver at location 4. These two addresses are determined at assembly time in our case. In a real situation, these addresses will be determined at execution time by the operating system, and thus the ISR and the printer driver will be located in the memory by the operating system (called re-locatable code). Subsequent memory locations contain constants defined in the program.

- Click single step button labeled as "19". FALSIM will execute the jump [main] instruction at address 0 and the PC will change to 20h (32 decimal), which is the address of the first instruction in the main program (i.e., the value of main).

- Although in a real situation, there will be many instructions in the main program, those instructions are not present in the dummy calling program. The first useful instruction is shown next. It loads the address of the printer driver in r6 from the pointer area in the memory. The registers r5 and r7 are also set up for passing the starting address of the print buffer and the number of bytes to be printed. In our dummy program, we bring these values in to these registers from the data area in the memory, and then pass these values to the printer driver using these two registers. Clicking on the single step button twice, executes these two instructions.

- The execution of the call instruction simulates the event of a print request by the user. This transfers control to the printer driver. Thus, when the call r4, r6 instruction is single stepped, the PC changes to 32h (50 decimal) for executing the first instruction in the printer driver.

- Double click on memory location 000A, which is being used for holding the PB (printer busy) flag. Enter a 1 and click the change memory button. This will store a 0001 in this location, indicating that a previous print job is in progress. Now click single step and note that this value is brought from memory location 000E into register r1. Clicking single step again will cause the jnz r1, [message] instruction to execute, and control will transfer to the message routine at address 0046h. The nop instruction is used here as a place holder.

- Click again on the single step button. Note that when the ret r4 instruction executes, the value in r4 (i.e., 28h) is brought into the PC. The blue highlight bar is placed on the next instruction after the call r4, r6 instruction in the main program. In case of the dummy calling program, this is the halt instruction.

- Double click on the value of the PC labeled as "20". This will open a dialog box shown below. Enter a value of the PC (i.e., 26h) corresponding to the call r4, r6 instruction, so that it can be executed again. A "list" of possible PC values can also be pulled down using, and 0026h can be selected from there as well.

- Click single step again to enter the printer driver again.

- Change memory location 000A to a 0, and then single step the first instruction in the printer driver. This will bring a 0 in r1, so that when the next jnz r1, [message] instruction is executed, the branch will not be taken and control will transfer to the next instruction after this instruction. This is mivi r1, 1 at address 0036h.

- Continue single stepping.

- Notice that a 1 has been stored in memory location 000A, and r1 contains 11h, which is then transferred to the output port at address 3Ch (60 decimal) when the out r1, controlp instruction executes. This can be verified by double clicking on the top left corner of the I/O port pane, and changing the address to 3Ch. Another way to display the value of an I/O port is to scroll the I/O window pane to the desired position.

- Continue single stepping till the int instruction and note the changes in different panes of the simulation window at each step.

- When the int instruction executes, the PC changes to 64h, which is the address of the first instruction in the ISR. Clicking single step executes this instruction, and loads the address of temp (i.e., 0010h) which is a temporary memory area for storing the environment. The five store instructions in the ISR save the CPU environment (working registers) before the ISR change them.

- Single step through the ISR while noting the effects on various registers, memory locations, and I/O ports till the iret instruction executes. This will pass control back to the printer driver by changing the PC to the address of the jump [finish] instruction, which is the next instruction after the int instruction.

- Double click on the value of the PC. Change it to point to the int instruction and click single step to execute it again. Continue to single step till the in r1, statusp instruction is ready for execution.

- Change the I/O port at address 3Ah (which represents the status port at address 58) to 80 and then single step the in r1, statusp instruction. The value in r1 should be 0080.

- Single step twice and notice that control is transferred to the movi r7, FFFF[3] instruction, which stores an error code of –1 in r1.

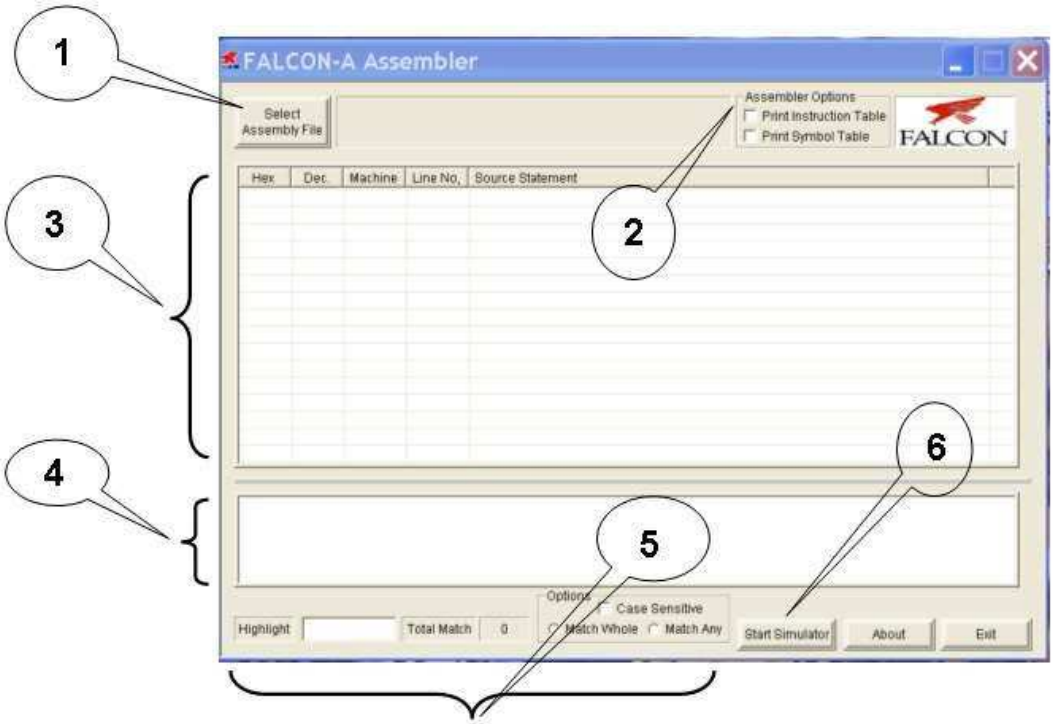


Figure 1

[3] The instruction was originally **movi r7, -1.** Since it was converted to machine language by the assembler, and then reverse assembled by the simulator, it became **movi r7, FFFF.** This is

because the machine code stores the number in 16-bits after sign-extension. The result will be the same in both cases.

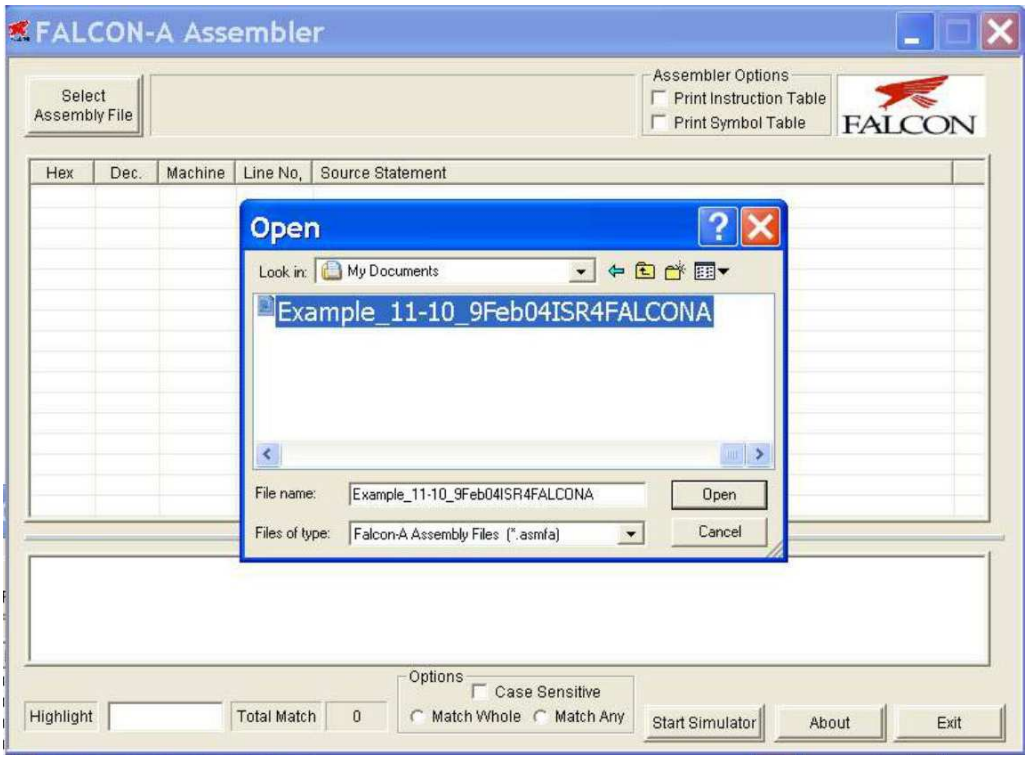


Figure 2

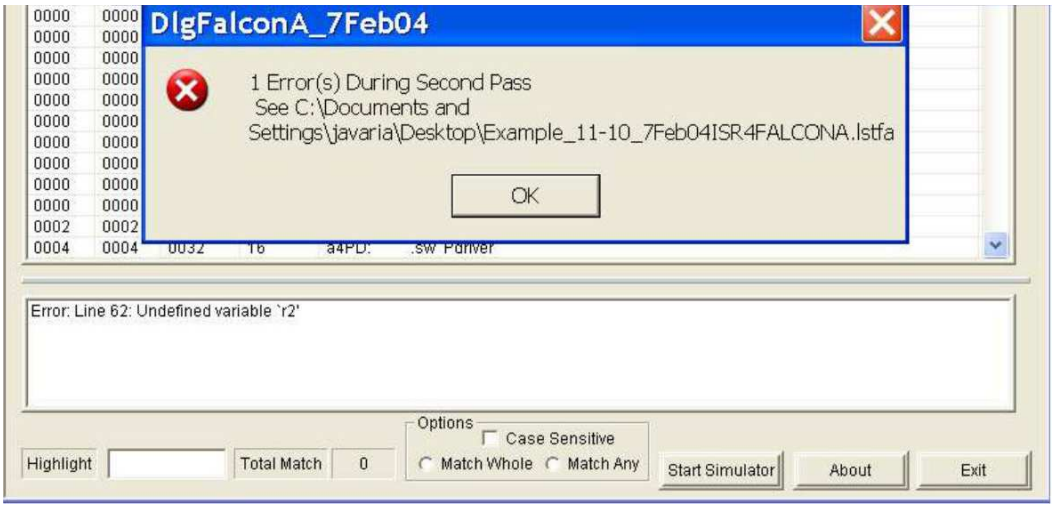


Figure 3

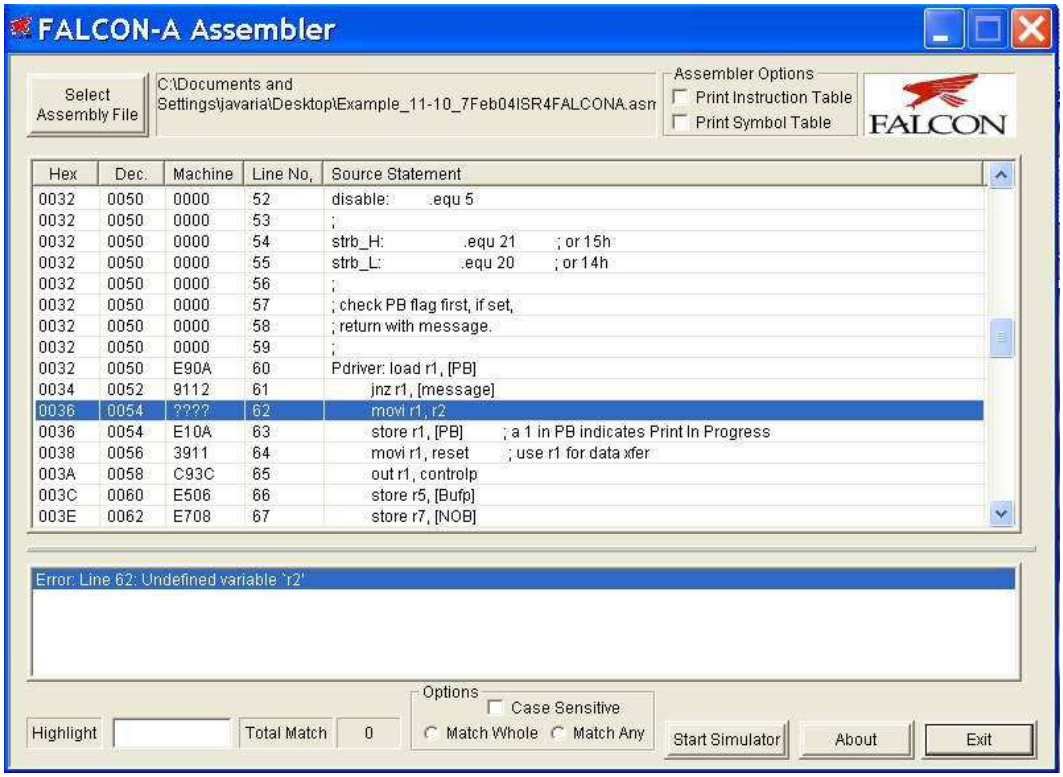


Figure 4

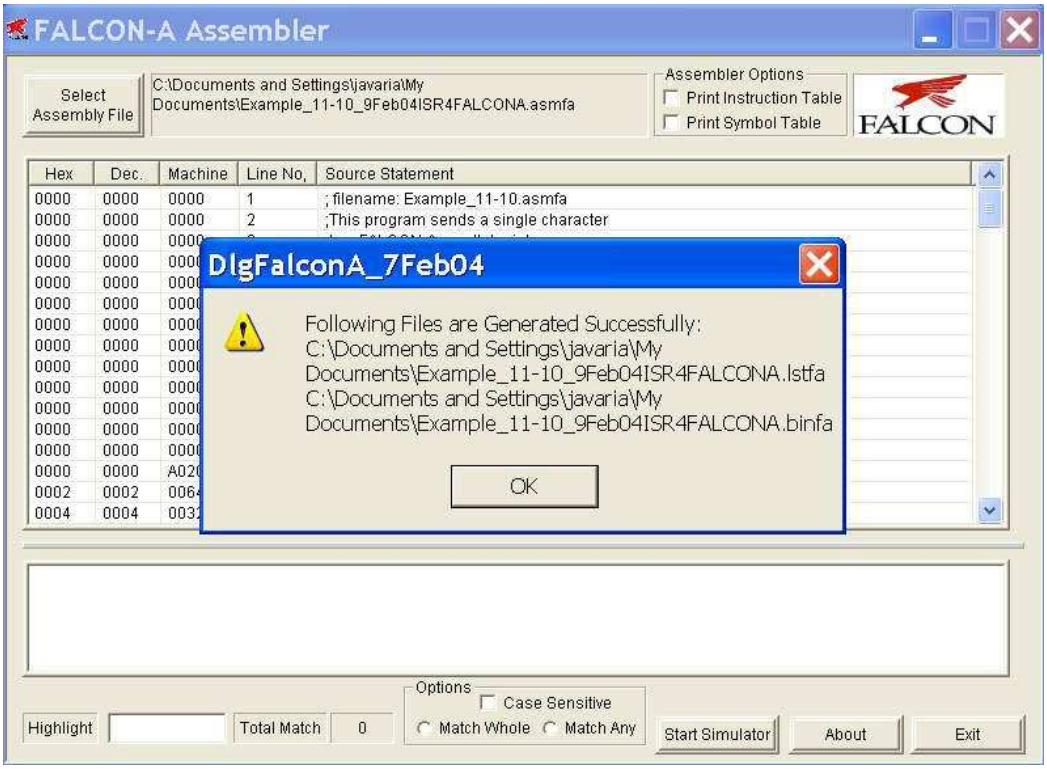
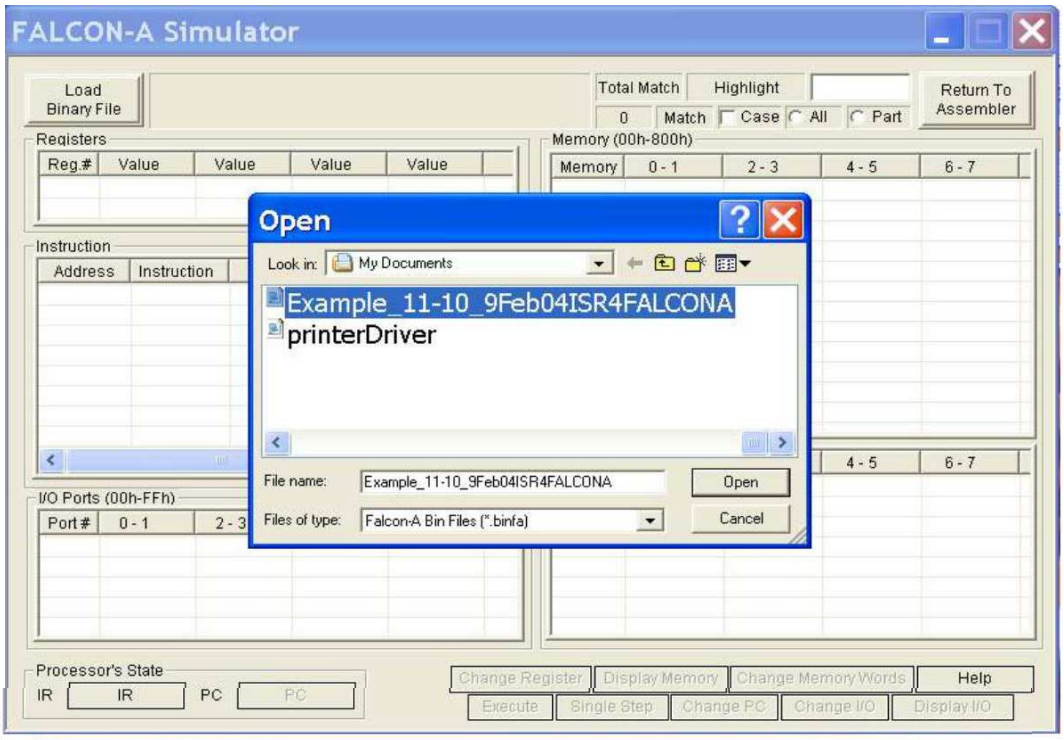


Figure 5

Figure 6



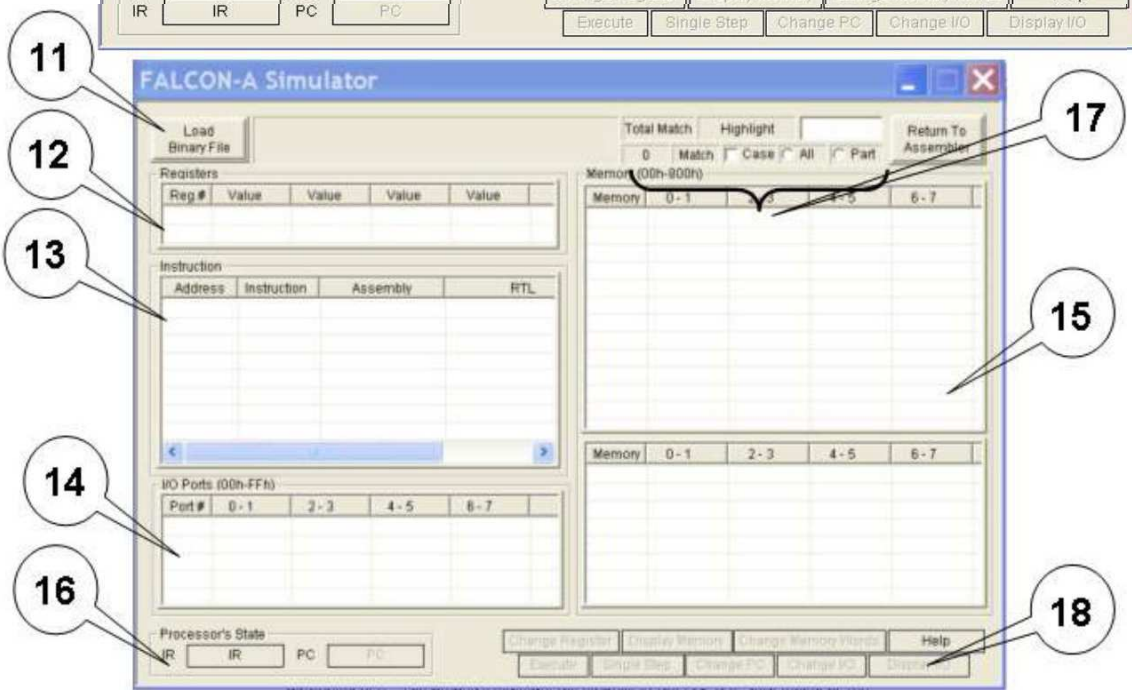


Figure 7

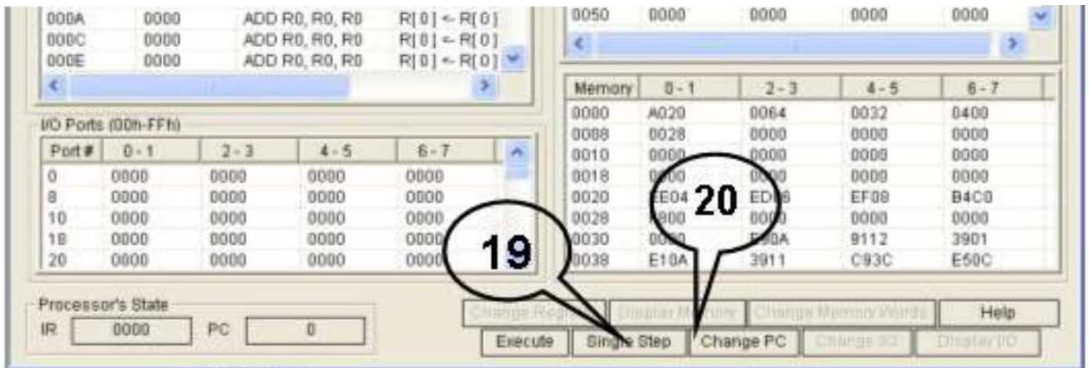


Figure 8

4. FALCON-A assembly language programming techniques:

- If a signed value, x, cannot fit in 5 bits (i.e., it is outside the range -16 to +15), FALSIM will report an error with a load r1, [x] or a store r1, [x] instruction. To overcome this problem, use movi r2, x followed by load r1, [r2].

- If a signed value, x, cannot fit in 8 bits (i.e., it is outside the range - 128 to +127), even the previous scheme will not work. FALSIM will report an error with the movi r2, x instruction. The following instruction sequence should be used to overcome this limitation of the FALCON-A. First store the 16-bit address in the memory using the .sw directive. Then use two load instructions as shown below:

.sw x load r2, [a] load r1, [r2]

- This is essentially a "memory-register-indirect" addressing. It has been made possible by the .sw directive. The value of a should be less than 15.

- A similar technique can be used with immediate ALU instructions for large values of the immediate data, and with the transfer of control (call and jump) instructions for large values of the target address.

- Large values (16-bit values) can also be stored in registers using the **mul** instruction combined with the **addi** instruction. The following instructions bring a 201 in register r1.

```
movi r2, 10
movi r3, 20
mul r1, r2, r3          ;     r1 contains 200 after this instruction
addi r1, r1, 1          ;     r1 now contains 201
```

- Moving from one register to another can be done by using the instruction **addi r2, r1, 0.**

- Bit setting and clearing can be done using the logical (and, or, not, etc) instructions.

- Using shift instructions (shiftl, asr, etc.) is faster that **mul** and **div,** if the multiplier or divisor is a power of 2.

Lecture No. 1

Introduction

Reading Material

Vincent P. Heuring & Harry F. Jordan	Chapter 1
Computer Systems Design and Architecture	1.1, 1.2, 1.3, 1.4, 1.5

Summary

- Distinction between computer architecture, organization and design
- Levels of abstraction in digital design
- Introduction to the course topics
- Perspectives of different people about computers
- General operation of a stored program digital computer
- The Fetch-Execute process
- Concept of an ISA(Instruction Set Architecture)

Introduction

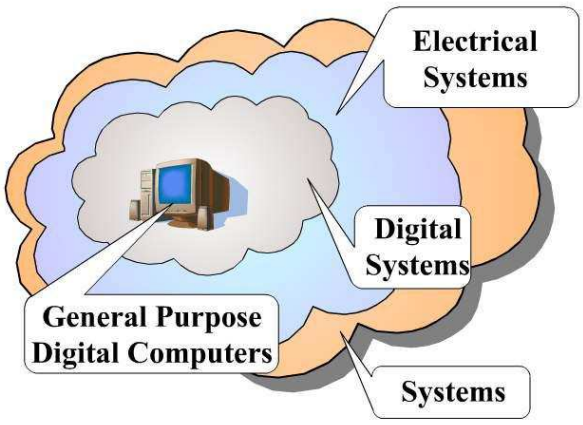
This course is about Computer Architecture. We start by explaining a few key terms.

The General Purpose Digital Computer

How can we define a 'computer'? There are several kinds of devices that can be termed "computers": from desktop machines to the microcontrollers used in appliances such as a microwave oven, from the Abacus to the cluster of tiny chips used in parallel processors, etc. For the purpose of this course, we will use the following definition of a computer:

"An electronic device, operating under the control of instructions stored in its own memory unit, that can accept data (input), process data arithmetically and logically, produce output from the processing, and store the results for future use." [1]

Thus, when we use the term computer, we actually mean a digital computer. There are many digital computers, which have dedicated purposes, for example, a computer is used in an automobile that controls the spark timing for the engine. This means that when we use the term computer, we actually mean a general-purpose digital computer that can perform a variety of arithmetic and logic tasks.



Notion of a System

The Computer as a System

Now we examine the notion of a system, and the place of digital computers in the general universal set of systems. A "system" is a collection of elements, or components, working

together on one or more inputs to produce one or more desired outputs. There are many types of systems in the world. Examples include:

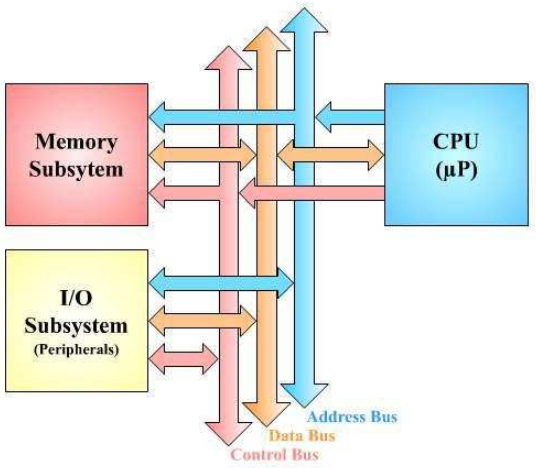
- Chemical systems
- Optical systems
- Biological systems
- Electrical systems
- Mechanical systems, etc.

These are all subsets of the general universal set of "systems". One particular subset of interest is an "electrical system". In case of electrical systems, the inputs as well as the outputs are electrical quantities, namely voltage and current. "Digital systems" are a subset of electrical systems. The inputs and outputs are digital quantities in this case. General-purpose digital computers are a subset of digital systems. We will focus on general-purpose digital computers in this course.

Essential Elements of a General Purpose Digital Computer

The figure shows the block diagram of a modern general-purpose digital computer.

We observe from the diagram that a general-purpose computer has three main components: a memory subsystem, an input/ output subsystem, and a central processing unit. Programs are stored in the memory, the execution of the program instructions takes place in the CPU, and the communication with the external world is achieved through the I/O subsystem (including the peripherals).



Block Diagram of a Computer System

Architecture

Now that we understand the term "computer" in our context, let us focus on the term architecture. The word architecture, as defined in standard dictionaries, is ***"the art or science of building"***, *or* ***"a method or style of building"***.[2]

Computer Architecture

This term was first used in 1964 by Amdahl, Blaauw, and Brooks at IBM [3]. They defined it as

"The structure of a computer that a machine language programmer must understand to write a correct (time independent) program for that machine."

By architecture, they meant the programmer visible portion of the instruction set. Thus, a family of machines of the same architecture should be able to run the same software (instructions). This concept is now so common that it is taken for granted. The x86 architecture is a well-known example.

The study of computer architecture includes

- a study of the structure of a computer
- a study of the instruction set of a computer
- a study of the process of designing a computer

Computer Organization versus Computer Architecture

It is difficult to make a sharp distinction between these two. However, architecture refers to the attributes of a computer that are visible to a programmer, including

- The instruction set
- The number of bits used to represent various data types
- I/O mechanisms
- Memory addressing modes, etc.

On the other hand, organization refers to the operational units of a computer and their interconnections that realize the architectural specifications. These include

- The control signals
- Interfaces between the computer and its peripherals
- Memory technology used, etc.

It is an architectural issue whether a computer will have a specific instruction or not, while it is an organizational issue how that instruction will be implemented.

Computer Architect

We can conclude from the discussion above that a computer architect is a person who designs computers.

Design

Design is defined as

"The process of devising a system, component, or process to meet desired needs."

Most people think of design as a "sketch". This is the usage of the term as a noun. However, the standard engineering usage of the term, as is quite evident from the above definition, is as a verb, i.e., "design is a process". A designer works with a set of stated requirements under a number of constraints to produce the best solution for a given problem. Best may mean a "cost-effective" solution, but not always. Additional or alternate requirements, like efficiency, the client or the designer may impose robustness, etc.. Therefore, design is a decision-making process (often iterative in nature), in which the basic sciences, mathematical concepts and engineering sciences are applied to convert a given set of resources optimally to meet a stated objective.

Knowledge base of a computer architect

There are many people in the world who know how to drive a car; these are the "users" of cars who are familiar with the behavior of a car and how to operate it. In the same way, there are people who can use computers. There are also a number of people in the world who know how to repair a car; these are "automobile technicians". In the same way, we have computer technicians. However, there are a very few people who know how to design a car; these are "automobile designers". In the same way, there are only very few experts in the world who can design computers. In this course, you will learn how to design computers!

These computer design experts are familiar with

- The structure of a computer
- The instruction set of a computer
- The process of designing a computer as well as few other related things.

At this point, we need to realize that it is not the job of a single person to design a computer from scratch. There are a number of levels of computer design. Domain experts of that particular level carry out the design activity for each level. These levels of abstraction of a digital computer's design are explained below.

Digital Design: Levels of Abstraction

Processor-Memory-Switch level (PMS level)

The highest is the processor-memory-switch level. This is the level at which an architect views the system. It is simply a description of the system components and their interconnections. The components are specified in the form of a block diagram.

Instruction Set Level

The next level is instruction set level. It defines the function of each instruction. The emphasis is on the behavior of the system rather than the hardware structure of the system.

Register Transfer Level

Next to the ISA (instruction set architecture) level is the register transfer level. Hardware structure is visible at this level. In addition to registers, the basic elements at this level are multiplexers, decoders, buses, buffers etc.

The above three levels relate to "system design".

Logic Design Level

The logic design level is also called the gate level. The basic elements at this level are gates and flip-flops. The behavior is less visible, while the hardware structure predominates.

The above level relates to "logic design".

Circuit Level

The key elements at this level are resistors, transistors, capacitors, diodes etc.

Mask Level

The lowest level is mask level dealing with the silicon structures and their layout that implement the system as an integrated circuit.

The above two levels relate to "circuit design".

The focus of this course will be the register transfer level and the instruction set level, although we will also deal with the PMS level and the Logic Design Level.

Objectives of the course

This course will provide the students with an understanding of the various levels of studying computer architecture, with emphasis on instruction set level and register transfer level. They will be able to use basic combinational and sequential building blocks to design larger structures like ALUs (Arithmetic Logic Units), memory subsystems, I/O subsystems etc. It will help them understand the various approaches used to design computer CPUs (Central Processing Units) of the RISC (Reduced Instruction Set Computers) and the CISC (Complex Instruction Set Computers) type, as well as the principles of cache memories.

Important topics to be covered

- Review of computer organization
- Classification of computers and their instructions
- Machine characteristics and performance
- Design of a Simple RISC Computer: the SRC
- Advanced topics in processor design
- Input-output (I/O) subsystems
- Arithmetic Logic Unit implementation
- Memory subsystems

Course Outline Introduction: • Distinction between Computer Architecture, Organization and design • Levels of abstraction in digital design • Introduction to the course topics
Brief review of computer organization: • Perspectives of different people about computers • General operation of a stored program digital computer • The Fetch – Execute process • Concept of an ISA
Foundations of Computer Architecture: • A taxonomy of computers and their instructions • Instruction set features • Addressing Modes • RISC and CISC architectures • Measures of performance
An example processor: The SRC: • Introduction to the ISA and instruction formats • Coding examples and Hand assembly • Using Behavioral RTL to describe the SRC • Implementing Register Transfers using Digital Logic Circuits
ISA: Design and Development • Outline of the thinking process for ISA design • Introduction to the ISA of the FALCON – A • Solved examples for FALCON-A • Learning Aids for the FALCON-A
Other example processors: • FALCON-E • EAGLE and Modified EAGLE • Comparison of the four ISAs
CPU Design: • The Design Process • A Uni-Bus implementation for the SRC • Structural RTL for the SRC instructions • Logic Design for the 1-Bus SRC • The Control Unit • The 2-and 3-Bus Processor Designs • The Machine Reset • Machine Exceptions
Term Exam – I
Advanced topics in processor design: • Pipelining • Instruction-Level Parallelism • Microprogramming

Input-output (I/O): • I/O interface design • Programmed I/O • Interrupt driven I/O • Direct memory access (DMA) Term Exam – II
Arithmetic Logic Shift Unit (ALSU) implementation: • Addition, subtraction, multiplication & division for integer unit • Floating point unit
Memory subsystems: • Memory organization and design • Memory hierarchy • Cache memories • Virtual memory

References

[1] Shelly G.B., Cashman T.J., Waggoner G.A., Waggoner W.C., Complete Computer Concepts: Microcomputer and Applications. Ferncroft Village Danvers, Massachusetts: Boyd & Fraser, 1992.

[2] Merriam-Webster Online; The Language Centre, May 12, 2003 (http://www.m-w.com/home.htm).

[3] Patterson, D.A. and Hennessy, J.L., Computer Architecture- A Quantitative Approach, 2nd ed., San Francisco, CA: Morgan Kauffman Publishers Inc., 1996.

[4] Heuring V.P. and Jordan H.F., Computer Systems Design and Architecture. Melano Park, CA: Addison Wesley, 1997.

A brief review of Computer Organization Perceptions of Different People about Computers

There are various perspectives that a computer can take depending on the person viewing it. For example, the way a child perceives a computer is quite different from how a computer programmer or a designer views it. There are a number of perceptions of the computer, however, for the purpose of understanding the machine, generally the following four views are considered.

The User's View

A user is the person for whom the machine is designed, and who employs it to perform some useful work through application software. This useful work may be composing some reports in word processing software, maintaining credit history in a spreadsheet, or even developing some application software using high-level languages such as C or Java. The list of "useful work" is not all-inclusive. Children playing games on a computer may argue that playing games is also "useful work", maybe more so than preparing an internal office memo.

At the user's level, one is only concerned with things like speed of the computer, the storage capacity available, and the behavior of the peripheral devices. Besides performance, the user is not involved in the implementation details of the computer, as the internal structure of the machine is made obscure by the operating system interface.

The Programmer's View

By "programmer" we imply machine or assembly language programmer. The machine or the assembly language programmer is responsible for the implementation of software required to execute various commands or sequences of commands (programs) on the computer. Understanding some key terms first will help us better understand this view, the associated tasks, responsibilities and tools of the trade.

Machine Language

Machine language consists of all the primitive instructions that a computer understands and is able to execute. These are strings of 1s and 0s.Machine language is the computer's native language. Commands in the machine language are expressed as strings of 1s and 0s. It is the lowest level language of a computer, and requires no further interpretation.

Instruction Set

A collection of all possible machine language commands that a computer can understand and execute is called its instruction set. Every processor has its own unique instruction set. Therefore, programs written for one processor will generally not run on another processor. This is quite unlike programs written in higher-level languages, which may be portable. Assembly/machine languages are generally unique to the processors on which they are run, because of the differences in computer architecture. Three ways to list instructions in an instruction set of a computer:

- by function categories
- by an alphabetic ordering of mnemonics
- by an ascending order of op-codes

Assembly Language

Since it is extremely tiring as well as error-prone to work with strings of 1s and 0s for writing entire programs, assembly language is used as a substitute symbolic representation using "English like" key words called mnemonics. A pure assembly language is a language in which each statement produces exactly one machine instruction, i.e. there is a one-to-one correspondence between machine instructions and statements in the assembly language. However, there are a few exceptions to this rule, the

Pentium jump instruction shown in the table below serves as an example.

Example

The table provides us with some assembly statement and the machine language equivalents of the Intel x 86 processor families.

Assembly Language	Machine Language (Binary)	Machine Language (Hex)	Instruction type
add cx, dx	0000 0001 1101 0001	01 D1	Arithmetic
mov alx, 34h	1011 1000 0011 0100 0000 0000	B8 34 00	Data transfer
xor ax, bx	0011 0001 1101 1000	31 D8	Logic
jmp alpha	1110 1011 1111 1100	EB FC	Control

Alpha is a label, and its value will be determined by the position of the jmp instruction in the program and the position of the instruction whose address is alpha. So the second byte in the last instruction can be different for different programs.

Hence there is a one-to-many correspondence of the assembly to machine language in this instruction.

Users of Assembly Language

- **The machine designer**

The designer of a new machine needs to be familiar with the instruction sets of other machines in order to be able to understand the trade-offs implicit in the design of those instruction sets.

- **The compiler writer**

A compiler is a program that converts programs written in high-level languages to machine language. It is quite evident that a compiler writer must be familiar with the machine language of the processor for which the compiler is being designed. This understanding is crucial for the design of a compiler that produces correct and optimized code.

- **The writer of time or space critical code**

A complier may not always produce optimal code. Performance goals may force program-specific optimizations in the assembly language.

- **Special purpose or embedded processor programmer**

Higher-level languages may not be appropriate for programming special purpose or embedded processors that are now in common use in various appliances. This is because the functionality required in such applications is highly specialized. In such a case, assembly language programming is required to implement the required functionality.

Useful tools for assembly language programmers

- **The assembler:**

Programs written in assembly language require translation to the machine language, and an assembler performs this translation. This conversion process is termed as the assembly process. The assembly process can be done manually as well, but it is very tedious and error-prone.

An "assembler" that runs on one processor and translates an assembly language program written for another processor into the machine language of the other processor is called a "cross assembler".

- **The linker:**

When developing large programs, different people working at the same time can develop separate modules of functionality. These modules can then be 'linked' to form a single module that can be loaded and executed. The modularity of programs, that the linking step in assembly language makes possible, provides the same convenience as it does in higher-level languages; namely abstraction and separation of concerns. Once the functionality of a module has been verified for correctness, it can be re-used in any number of other modules. The programmer can focus on other parts of the program. This is the so-called "modular" approach, or the "top-down" approach.

- **The debugger or monitor:**

Assembly language programs are very lengthy and non-intuitive, hence quite tedious and error-prone. There is also the disadvantage of the absence of an operating system to handle run-time errors that can often crash a system, as opposed to the higher-level language programming, where control is smoothly returned to the operating system. In addition to run-time errors (such as a divide-by-zero error), there are syntax or logical errors.

A "debugger", also called a "monitor", is a computer program used to aid in detecting these errors in a program. Commonly, debuggers provide functionality such as

- The display and altering of the contents of memory, CPU registers and flags
- Disassembly of machine code (translating the machine code back to assembly language)
- Single stepping and breakpoints that allow the examination of the status of the program and registers at desired points during execution.

While syntax errors and many logical errors can be detected by using debuggers, the best debugger in the world can catch not every logical error.

- **The development system**

The development system is a complete set of (hardware and software) tools available to the system developer. It includes

- Assemblers
- Linkers and loaders

- Debuggers
- Compilers
- Emulators
- Hardware-level debuggers
- Logic analyzers, etc.

Difference between Higher-Level Languages and Assembly Language Higher-level languages are generally used to develop application software. These high-level programs are then converted to assembly language programs using compilers. So it is the task of a compiler writer to determine the mapping between the high-level-language constructs and assembly language constructs. Generally, there is a "many-to-many" mapping between high-level languages and assembly language constructs. This means that a given HLL construct can generally be represented by many different equivalent assembly language constructs. Alternately, a given assembly language construct can be represented by many different equivalent HLL constructs.

High-level languages provide various primitive data types, such as integer, Boolean and a string, that a programmer can use. Type checking provides for the verification of proper usage of these data types. It allows the compiler to determine memory requirements for variables and helping in the detection of bad programming practices.

On the other hand, there is generally no provision for type checking at the machine level, and hence, no provision for type checking in assembly language. The machine only sees strings of bits. Instructions interpret the strings as a type, and it is usually limited to signed or unsigned integers and floating point numbers. A given 32-bit word might be an instruction, an integer, a floating-point number, or 4 ASCII characters. It is the task of the compiler writer to determine how high-level language data types will be implemented using the data types available at the machine level, and how type checking will be implemented.

The Stored Program Concept

This concept is fundamental to all the general-purpose computers today. It states that the program is stored with data in computer's memory, and the computer is able to manipulate it as data. For example, the computer can load the program from disk, move it around in memory, and store it back to the disk.

Even though all computers have unique machine language instruction sets, the 'stored program' concept and the existence of a 'program counter' is common to all machines. The sequence of instructions to perform some useful task is called a program. All of the digital computers (the general purpose machine defined above) are able to store these sequences of instructions as stored programs. Relevant data is also stored on the computer's secondary memory. These stored programs are treated as data and the computer is able to manipulate them, for example, these can be loaded into the memory for execution and then saved back onto the storage.

General Operation of a Stored Program Computer

The machine language programs are brought into the memory and then executed instruction by instruction. Unless a branch instruction is encountered, the program is executed in sequence. The instruction that is to be executed is fetched from the memory and temporarily stored in a CPU register, called the instruction register (IR). The instruction register holds the instruction while it is decoded and executed by the central processing unit (CPU) of the computer. However, before loading an instruction into the instruction register for execution, the computer needs to know which instruction to load. The program counter (PC), also called the instruction pointer in some texts, is the register that holds the address of the next instruction in memory that is to be executed.

When the execution of an instruction is completed, the contents of the program counter (which is the address of the next instruction) are placed on the address bus. The memory places the instruction on the corresponding address on the data bus. The CPU puts this instruction onto the IR (**instruction register**) to decode and execute. While this instruction is decoded, its length in

bytes is determined, and the PC (**program counter**) is incremented by the length, so that the PC will point to the next instruction in the memory.

Note that the length of the instruction is not determined in the case of RISC machines, as the instruction length is fixed in these architectures, and so the program counter is always incremented by a fixed number. In case of branch instructions, the contents of the PC are replaced by the address of the next instruction contained in the present branch instruction, and the current status of the processor is stored in a register called the **Processor Status Word** (PSW). Another name for the PSW is the flag register. It contains the status bits, and control bits corresponding to the state of the processor. Examples of status bits include the sign bit, overflow bit, etc. Examples of control bits include interrupt enable flag, etc. When the execution of this instruction is completed, the contents of the program counter are placed on the address bus, and the entire cycle is repeated. This entire process of reading memory, incrementing the PC, and decoding the instruction is known as the **Fetch and Execute** principle of the stored program computer. This is actually an oversimplified situation. In case of the advanced processors of this age, a lot more is going on than just the simple "fetch and execute" operation, such as pipelining etc. The details of some of these more involved techniques will be studied later on during the course.

The Concept of Instruction Set Architecture (ISA)

Now that we have an understanding of some of the relevant key terms, we revert to the assembly language programmer's perception of the computer. The programmer's view is limited to the set of all the assembly instructions or commands that can the particular computer at hand execute understood/, in addition to the resources that these instructions may help manage. These resources include the memory space and the entire programmer accessible registers. Note that we use the term 'memory space' instead of memory, because not all the memory space has to be filled with memory chips for a particular implementation, but it is still a resource available to the programmer.

This set of instructions or operations and the resources together form the instruction set architecture (ISA). It is the ISA, which serves as an interface between the program and the functional units of a computer, i.e., through which, the computer's resources, are accessed and controlled.

The Computer Architect's View important long

The computer architect's view is concerned with the design of the entire system as well as ensuring its optimum performance. The optimality is measured against some quantifiable objectives that are set out before the design process begins. These objectives are set on the basis of the functionality required from the machine to be designed. The computer architect

- Designs the ISA for optimum programming utility as well as for optimum performance of implementation
- Designs the hardware for best implementation of instructions that are made available in the ISA to the programmer
- Uses performance measurement tools, such as benchmark programs, to verify that the performance objectives are met by the machine designed
- Balances performance of building blocks such as CPU, memory, I/O devices, and interconnections
- Strives to meet performance goals at the lowest possible cost
- Software models, simulators and emulators
- Performance benchmark programs
- Specialized measurement programs
- Data flow and bottleneck analysis
- Subsystem balance analysis

- Parts, manufacturing, and testing cost analysis

The Logic Designer's View

The logic designer is responsible for the design of the machine at the logic gate level. It is the design process at this level that determines whether the computer architect meets cost and performance goals. The computer architect and the logic designer have to work in collaboration to meet the cost and performance objectives of a machine. This is the reason why a single person or a single team may be performing the tasks of system's architectural design as well as the logic design.

Useful Tools for the Logic Designer

Some of the tools available that aid the logic designer in the logic design process are

- CAD tools

Logic design and simulation packages Printed circuit layout tools
IC (integrated circuit) design and layout tools

- Logic analyzers and oscilloscopes
- Hardware development systems

The Concept of the Implementation Domain

The collection of hardware devices, with which the logic designer works for the digital logic gate implementation and interconnection of the machine, is termed as the implementation domain. The logic gate implementation domain may be

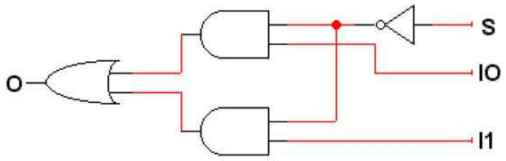
- VLSI (very large scale integration) on silicon
- TTL (transistor-transistor logic) or ECL (emitter-coupled logic) chips
- Gallium arsenide chips
- PLAs (programmable-logic arrays) or sea-of-gates arrays
- Fluidic logic or optical switches

Similarly, the implementation domains used for gate, board and module interconnections are

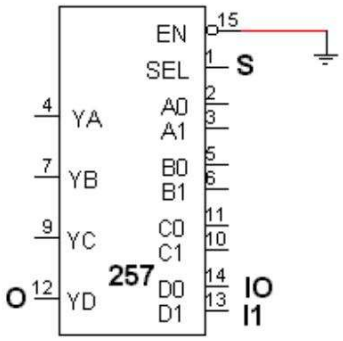
- Poly-silicon lines in ICs
- Conductive traces on a printed circuit board
- Electrical cable
- Optical fiber, etc.

At the lower levels of logic design, the designer is concerned mainly with the functional details represented in a symbolic form. The implementation details are not considered at these lower levels. They only become an issue at higher levels of logic design. An example of a two-to-one multiplexer in various implementation domains will illustrate this point. Figure (a) is the generic logic gate (abstract domain) representation of a 2-to-1 multiplexer.

Figure (b) shows the 2-to-1 multiplexer logic gate implementation in the domain of TTL (VLSI on Silicon) logic using part number '257, with interconnections in the domain of printed circuit board traces.



(a) Abstract view of Boolean logic

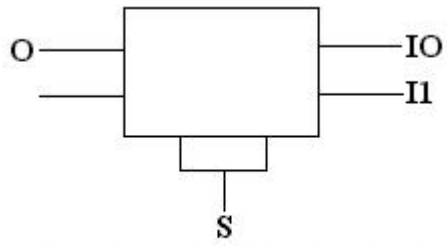


(b) TTL implementation domain

Figure (c) is the implementation of the 2-to-1 multiplexer with a fiber optic directional coupler switch, which has an interconnection domain of optical fiber.

Classical logic design versus computer logic design

We have already studied the sequential circuit design concepts in the course on Digital Logic Design, and thus are familiar with the techniques used. However, these traditional techniques for a finite state machine are not very practical when it comes to the design of a computer, in spite of the fact that a computer is a finite state machine. The reason is that employing these techniques is much too complex as the computer can assume hundreds of states.



(c) Optical switch implementation

Sequential Logic Circuit Design

When designing a sequential logic circuit, the problem is first coded in the form of a state diagram. The redundant states may be eliminated, and then the state diagram is translated into the next state table. The minimum number of flip-flops needed to implement the design is calculated by making "state assignments" in terms of the flip-flop "states". A "transition table" is made using the state assignments and the next state table. The flip-flop control characteristics are used to complete a set of "excitation tables". The excitation equations are determined through minimization. The logic circuit can then be drawn to implement the design. A detailed discussion of these steps can be found in most books on Logic Design.

Computer Logic Design

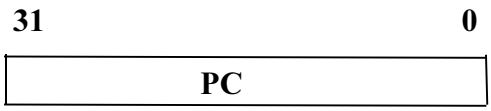
Traditional Finite State Machine (FSM) design techniques are not suitable for the design of computer logic. Since there is a natural separation between the data path and the control path in case of a digital computer, a modular approach can be used in this case.

The data path consists of the storage cells, the arithmetic and logic components and their interconnections. Control path is the circuitry that manages the data path information flow. So considering the behavior first can carry out the design. Then the structure can be considered and dealt with. For this purpose, well-defined logic blocks such as multiplexers, decoders, adders etc. can be used repeatedly.

Two Views of the CPU Program Counter Register

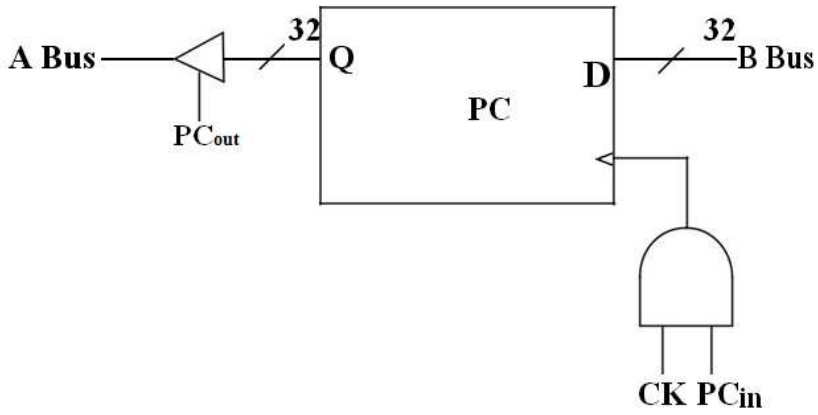
The view of a logic designer is more detailed than that of a programmer. Details of the mechanism used to control the machine are unimportant to the programmer, but of vital importance to the logic designer. This can be illustrated through the following two views of the program counter of a machine.

As shown in figure (a), to a programmer the program counter is just a register, and in this case, of length 32 bits or 4 bytes.



(a) Program Counter: Programmer's view

Figure (b) illustrates the logic designer's view of a 32-bit program counter, implemented as an array of 32 D flip-flops. It shows the contents of the program counter being gated out on 'A bus' (the address bus) by applying a control signal PC_{out}. The contents of the 'B bus' (also the address bus), can be stored in the program counter by asserting the signal PC_{in} on the leading edge of the clock signal CK, thus storing the address of the next instruction in the program counter.



(b) Program Counter: Logic Designer's View

Lecture No. 2

Instruction Set Architecture

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 2,Chapter3
Computer Systems Design and Architecture — 2.1, 2.2, 3.2

Summary

- A taxonomy of computers and their instructions
- Instruction set features
- Addressing modes
- RISC and CISC architectures

Foundations of Computer Architecture

Taxonomy of computers and their instructions

Processors can be classified on the basis of their instruction set architectures. The instruction set architecture, described in the previous module gives us a 'programmer's view' of the machine. This module discussed a number of topics related to the classifications of computers and their instructions.

CLASSES OF INSTRUCTION SET ARCHITECTURE:

The mechanism used by the CPU to store instructions and data can be used to classify the ISA (Instruction Set Architecture). There are three types of machines based on this classification.

- Accumulator based machines
- Stack based machines
- General purpose register (GPR) machines

ACCUMULATOR BASED MACHINES

Accumulator based machines use special registers called the accumulators to hold one source operand and also the result of the arithmetic or logic operations performed. Thus the accumulator registers collect (or 'accumulate') data. Since the accumulator holds one of the operands, one more register may be required to hold the address of another operand. The accumulator is not used to hold an address. So accumulator based machines are also called 1-address machines. Accumulator machines employ a very small number of accumulator registers, generally only one. These machines were useful at the time when memory was quite expensive; as they used one register to hold the source operand as well as the result of the operation. However, now that the memory is relatively inexpensive, these are not considered very useful, and their use is severely limited for the computation of expressions with many operands.

STACK BASED MACHINES

A stack is a group of registers organized as a last-in-first-out (LIFO) structure. In such a structure, the operands stored first, through the push operation, can only be accessed last, through

a pop operation; the order of access to the operands is reverse of the storage operation. An analogy of the stack is a "plate-dispenser" found in several self-service cafeterias. Arithmetic and logic operations successively pick operands from the top-of-the-stack (TOS), and push the results on the TOS at the end of the operation. In stack based machines, operand addresses need not be specified during the arithmetic or logical operations. Therefore, these machines are also called 0-address machines.

GENERAL-PURPOSE-REGISTER MACHINES

In general purpose register machines, a number of registers are available within the CPU. These registers do not have dedicated functions, and can be employed for a variety of purposes. To identify the register within an instruction, a small number of bits are required in an instruction word. For example, to identify one of the 64 registers of the CPU, a 6-bit field is required in the instruction.

CPU registers are faster than cache memory. Registers are also easily and more effectively used by the compiler compared to other forms of internal storage. Registers can also be used to hold variables, thereby reducing memory traffic. This increases the execution speed and reduces code size (fewer bits required to code register names compared to memory) .In addition to data, registers can also hold addresses and pointers (i.e., the address of an address). This increases the flexibility available to the programmer.

A number of dedicated, or special purpose registers are also available in general-purpose machines, but many of them are not available to the programmer. Examples of transparent registers include the stack pointer, the program counter, memory address register, memory data register and condition codes (or flags) register, etc.

We should understand that in reality, most machines are a combination of these machine types. Accumulator machines have the advantage of being more efficient as these can store intermediate results of an operation within the CPU.

INSTRUCTION SET

An instruction set is a collection of all possible machine language commands that are understood and can be executed by a processor.

ESSENTIAL ELEMENTS OF COMPUTER INSTRUCTIONS:

There are four essential elements of an instruction; the type of operation to be performed, the place to find the source operand(s), the place to store the result(s) and the source of the next instruction to be executed by the processor.

Type of operation

In module 1, we described three ways to list the instruction set of a machine; one way of enlisting the instruction set is by grouping the instructions in accordance with the functions they perform. The type of operation that is to be performed can be encoded in the op-code (or the operation code) field of the machine language instruction. Examples of operations are mov, jmp, add; these are the assembly mnemonics, and should not be confused with op-codes. Op-codes are simply bit-patterns in the machine language format of an instruction.

Place to find source operands

An instruction needs to specify the place from where the source operands will be retrieved and used. Possible locations of the source operands are CPU registers, memory cells and I/O locations. The source operands can also be part of an instruction itself; such operands are called immediate operands.

Place to store the results

An instruction also specifies the location in which the result of the operation, specified by the instruction, is to be stored. Possible locations are CPU registers, memory cells and I/O locations.

Source of the next instruction

By default, in a program the next instruction in sequence is executed. So in cases where the next-in-sequence instruction execution is desired, the place of next instruction need not be encoded within the instruction, as it is implicit. However, in case of a branch, this information needs to be encoded in the instruction. A branch may be conditional or unconditional, a subroutine call, as well as a call to an interrupt service routine.

Example

The table provides examples of assembly language commands and their machine language equivalents. In the instruction add cx, dx, the contents of the location dx are added to the contents of the location cx, and the result is stored in cx. The instruction type is arithmetic, and the op-code for the add instruction is 0000, as shown in this example.

Assembly Language	Machine Language (Binary)	Machine Language (Hex)	Instruction type
add cx, dx	0000 0001 1101 0001	01 D1	Arithmetic
mov al, 34h	1011 1000 0011 0100 0000 0000	B8 34 00	Data transfer
xor ax, bx	0011 0001 1101 1000	31 D8	Logic
jmp alpha	1110 1011 1111 1100	EB FC	Control

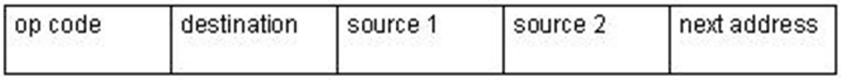
CLASSIFICATIONS OF INSTRUCTIONS:

We can classify instructions according to the format shown below.

- 4-address instructions
- 3-address instructions
- 2-address instructions
- 1-address instructions
- 0-address instructions

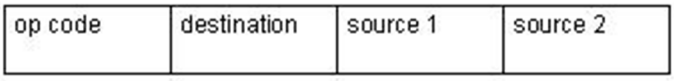
The distinction is based on the fact that some operands are accessed from memory, and therefore require a memory address, while others may be in the registers within the CPU or they are specified implicitly.

4-address instructions



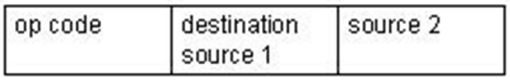
The four address instructions specify the addresses of two source operands, the address of the destination operand and the next instruction address.
4-address instructions are not very common because the next instruction to be executed is sequentially stored next to the current instruction in the memory. Therefore, specifying its address is redundant. These instructions are used in the micro-coded control unit, which will be studied later.

3-address instruction



A 3-address instruction specifies the addresses of two operands and the address of the destination operand.

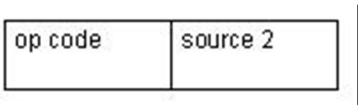
2-address instruction



A 2-address instruction has three fields; one for the op-code, the second field specifies the address of one of the source operands as well as the destination operand, and the last field is used for holding the address of the second source operand. So one of the fields serves two purposes; specifying a source operand address and a destination operand address.

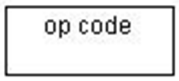
1-address instruction

A 1-address instruction has a dedicated CPU register, called the accumulator, to hold one operand and to store the result. There is no need of encoding the address of the accumulator register to access the operand or to store the result, as its usage is implicit. There are two fields in the instruction, one for specifying a source operand address and a destination operand address.



0-address instruction

A 0-address instruction uses a stack to hold both the operands and the result. Operations are performed on the operands stored on the top of the stack and the second value on the stack. The result is stored on the top of the stack. Just like the use of an accumulator register, the addresses of the stack registers need not be specified, their usage is implicit. Therefore, only one field is required in 0-address instruction; it specifies the op-code.



COMPARISON OF INSTRUCTION FORMATS:

Basis for comparison

Two parameters are used as the basis for comparison of the instruction sets discussed above. These are

- Code size

Code size has an effect on the storage requirements for the instructions; the greater the code size, the larger the memory required.

- Number of memory accesses

The number of memory accesses has an effect on the execution time of instructions; the greater the numbers of memory accesses, the larger the time required for the execution cycle, as memory accesses are generally slow.

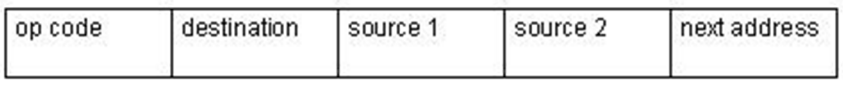
Assumptions

We make a few assumptions, which are

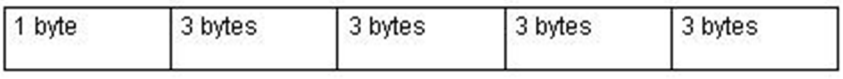
- A single byte is used for the op code, so 256 instructions can be encoded using these 8 bits, as $2^8 = 256$
- The size of the memory address space is 16 Mbytes
- A single addressable memory unit is a byte
- Size of operands is 24 bits. As the memory size is 16Mbytes, with byte-addressable memory, 24 bits are required to encode the address of the operands.
- The size of the address bus is 24 bits
- Data bus size is 8 bits

Discussion

4-address instruction



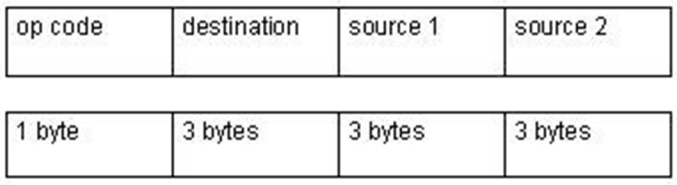
- The code size is 13 bytes (1+3+3+3+3 = 13 bytes)



- Number of bytes accessed from memory is 22 (13 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 22 bytes)

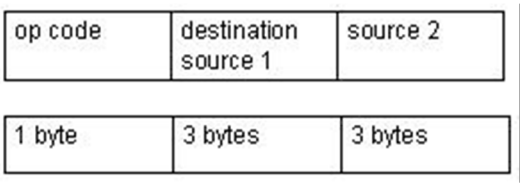
Note that there is no need for an additional memory access for the operand corresponding to the next instruction, as it has already been brought into the CPU during instruction fetch.

3-address instruction



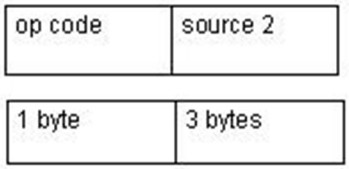
- The code size is 10 bytes (1+3+3+3 = 10 bytes)
- Number of bytes accessed from memory is 19 (10 bytes for instruction fetch 6 bytes for source operand fetch + 3 bytes for storing destination operand = 19 bytes)

2-address instruction



- The code size is 7 bytes (1+3+3 = 7 bytes)
- Number of bytes accessed from memory is 16(7 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 16 bytes)

1-address instruction



- The code size is 4 bytes (1+3= 4 bytes)
- Number of bytes accessed from memory is 7 (4 bytes for instruction fetch + 3 bytes for source operand fetch + 0 bytes for storing destination operand = 7 bytes)

0-address instruction

- The code size is 1 byte
- Number of bytes accessed from memory is 10 (1 byte for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 10 bytes)

The following table summarizes this information

Instruction Format	Code size	Number of memory bytes
4-address instruction	13	22
3-address instruction	10	19
2-address instruction	7	16
1-address instruction	4	7
0-address instruction	1	10

HALF ADDRESSES

In the preceding discussion we have talked about memory addresses. This discussion also applies to CPU registers. However, to specify/ encode a CPU register, less number of bits is required as compared to the memory addresses. Therefore, these addresses are also called "half-addresses". An instruction that specifies one memory address and one CPU register can be called as a 1½-address instruction

Example
mov al, [34h]

THE PRACTICAL SITUATION

Real machines are not as simple as the classifications presented above. In fact, these machines have a mixture of 3, 2, 1, 0, and 1½-address instructions. For example, the VAX 11 includes instructions from all classes.

CLASSIFICATION OF MACHINES ON THE BASIS OF OPERAND AND RESULT LOCATION:

A distinction between machines can be made on the basis of the ALU instructions; whether these instructions use data from the memory or not. If the ALU instructions use only the CPU registers for the operands and result, the machine type is called "**load-store**". Other machines may have a mixture of register-memory, or memory-memory instructions.

The number of memory operands supported by a typical ALU instruction may vary from 0 to 3.

Example

The SPARC, MIPS, Power PC, ALPHA: 0 memory addresses, max operands allowed = 3
X86, 68x series: 1 memory address, max operands allowed = 2

LOAD- STORE MACHINES

These machines are also called the register-to-register machines. They typically use the 1½ address instruction format. Only the load and store instructions can access the memory. The load instruction fetches the required data from the memory and temporarily stores it in the CPU registers. Other instructions may use this data from the CPU registers. Then later, the results can be stored back into the memory by the store instruction. Most RISC computers fall under this category of machines.

Advantages (of register-register instructions)

Register-register instructions use 0 memory operands out of a total of 3 operands. The advantages of such a scheme is:

- The instructions are simple and fixed in length
- The corresponding code generation model is simple
- All instructions take similar number of clock cycles for execution

Disadvantages (register-register instructions)

- The instruction count is higher; the number of instructions required to complete a particular task is more as separate instructions will be required for load and store operations of the memory
- Since the instruction size is fixed, the instructions that do not require all fields waste memory bits

Register-memory machines

In register-memory machines, some operands are in the memory and some are in registers. These machines typically employ 1 or 1½ address instruction format, in which one of the operands is an accumulator or a general-purpose CPU registers.

Advantages

Register-memory operations use one memory operand out of a total of two operands. The advantages of this instruction format are

- Operands in the memory can be accessed without having to load these first through a separate load instruction
- Encoding is easy due to the elimination of the need of loading operands into registers first
- Instruction bit usage is relatively better, as more instructions are provided per fixed number of bits

Disadvantages

- Operands are not equivalent since one operand may have two functions (both source operand and destination operand), and the source operand may be destroyed
- Different size encoding for memory and registers may restrict the number of registers

- The number of clock cycles per instruction execution vary, depending on the operand location operand fetch from memory is slow as compared to operands in CPU registers

Memory-Memory Machines

In memory-memory machines, all three of the operands (2 source operands and a destination operand) are in the memory. If one of the operands is being used both as a source and a destination, then the 2-address format is used. Otherwise, memory-memory machines use 3-address formats of instructions.

Advantages

- The memory-memory instructions are the most compact instruction where encoding wastage is minimal.
- As operands are fetched from and stored in the memory directly, no CPU registers are wasted for temporary storage

Disadvantages

- The instruction size is not fixed; the large variation in instruction sizes makes decoding complex
- The cycles per instruction execution also vary from instruction to instruction
- Memory accesses are generally slow, so too many references cause performance degradation

Example 1

The expression **a = (b+c)*d – e** is evaluated with the 3, 2, 1, and 0-address machines to provide a comparison of their advantages and disadvantages discussed above.

3-Address	2-Address	1-Address	0-Address

add a, b, c
mpy a, a, d
sub a, a, e
	load a, b
	add a, c
	mpy a, d
	sub a, e
		lda b
		add c
		mpy d
		sub e
		sta a
			push b
			push c
			add
			push d
			mpy
			push e
			sub
			pop a

The instructions shown in the table are the minimal instructions required to evaluate the given expression. Note that these are not machine language instructions, rather the pseudo-code.

Example 2

The instruction **z = 4(a +b) – 16(c+58)** is with the 3, 2, 1, and 0-address machines in the table.

3-Address	2-Address	1-Address	0-Address

add x, a, b
mul y, x, 4
add r, c, 58
mul s, r, 16
sub z, y, s
	load y, a
	add y, b
	mul y, 4
	load s, c
	add s, 58
	mul s, 16
	sub y, s
	store z, y
		; order changed to reduce code size
		lda c
		adda 58
		mula 16
		sta s
		lda a
		adda b ;add b to acc
		mula 4
		suba s ;subtract acc from s
		sta z
			push a
			push b
			add
			push 4
			mul
			push c
			push 58
			add
			push 16
			mul
			sub
			pop z

Functional classification of instruction sets:

Instructions can be classified into the following four categories based on their functionality.

- Data processing
- Data storage (main memory)
- Data movement (I/O)
- Program flow control

- **Data processing**

Data processing instructions are the ones that perform some mathematical or logical operation on some operands. The Arithmetic Logic

Unit performs these operations; therefore the data processing instructions can also be called ALU instructions.

- **Data storage (main memory)**

The primary storage for the operands is the main memory. When an operation needs to be performed on these operands, these can be temporarily brought into the CPU registers, and after completion, these can be stored back to the memory. The instructions for data access and storage between the memory and the CPU can be categorized as the data storage instructions.

- **Data movement (I/O)**

The ultimate sources of the data are input devices e.g. keyboard. The destination of the data is an output device, for example, a monitor, etc. The instructions that enable such operations are called data movement instructions.

- **Program flow control**

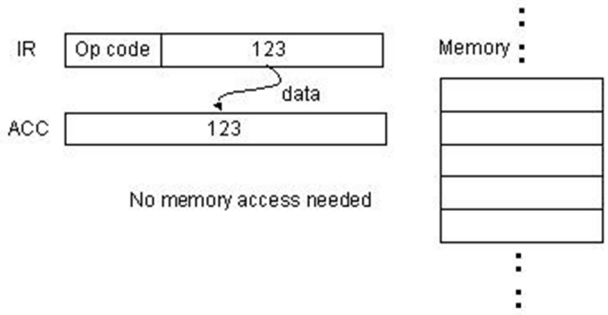
A CPU executes instructions sequentially, unless a program flow-change instruction is encountered. This flow change, also called a branch, may be conditional or unconditional. In case of a conditional branch, if the branch condition is met, the target address is loaded into the program counter.

ADDRESSING MODES:

Addressing modes are the different ways in which the CPU generates the address of operands. In other words, they provide access paths to memory locations and CPU registers.

Effective address

An "effective address" is the address (binary bit pattern) issued by the CPU to the memory. The CPU may use various ways to compute the effective address. The memory may interpret the effective address differently under different situations.



COMMONLY USED ADDRESSING MODES

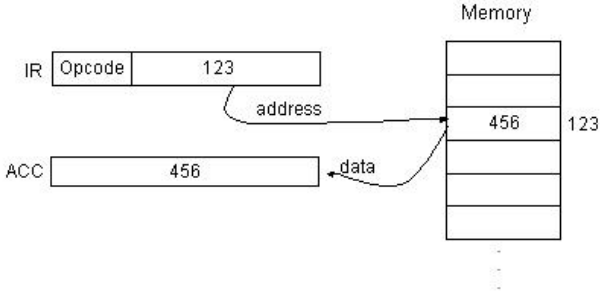
Some commonly used addressing modes are explained below.

Immediate addressing mode

In this addressing mode, data is the part of the instruction itself, and so there is no need of address calculation. However, immediate addressing mode is used to hold source operands only; cannot be used for storing results. The range of the operands is limited by the number of bits available for encoding the operands in the instruction; for n bit fields, the range is $-2^{(n-1)}$ to $+(2^{(n-1)}-1)$.

Example: lda 123

In this example, the immediate operand, 123, is loaded onto the accumulator. No address calculation is required.



Direct Addressing Mode

The address of the operand is specified as a constant, and this constant is coded as part of the instruction. The address space that can be accessed is limited address space by the operand field size ($2^{\text{operand field size}}$ locations).

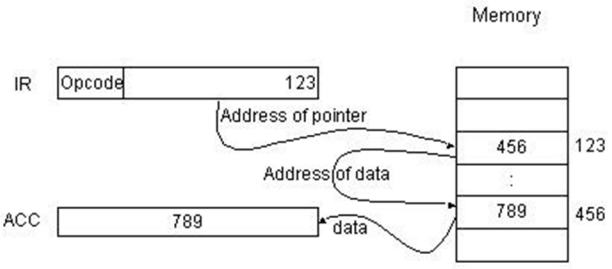
Example: lda [123]

As shown in the figure, the address of the operand is stored in the instruction. The operand is then fetched from that memory address.

Indirect Addressing Mode

The address of the location where the address of the data is to be found is stored in the instruction as the operand.

Thus, the operand is the address of a memory location, which holds the address of the operand. Indirect addressing mode can access a large address space ($2^{\text{memory word size}}$ locations).



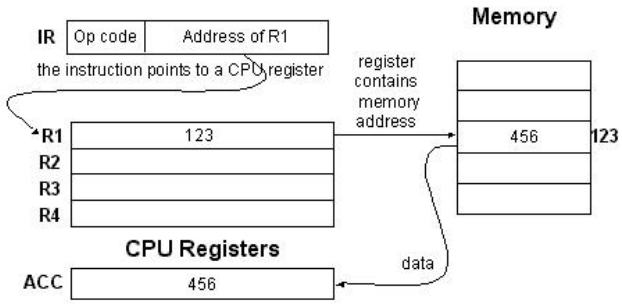
To fetch the operand in this addressing mode, two memory accesses are required. Since memory accesses are slow, this is not efficient for frequent memory accesses. The indirect addressing mode may be used to implement pointers.

Example: lda [[123]]

As shown in the figure, the address of the memory location that holds the address of the data in the memory is part of the instruction.

Register (Direct) Addressing Mode

The operand is contained in a CPU register, and the address of this register is encoded in the instruction. As no memory access is needed, operand fetch is efficient. However, there are only a limited number of CPU registers available, and this imposes a limitation on the use of this addressing mode.

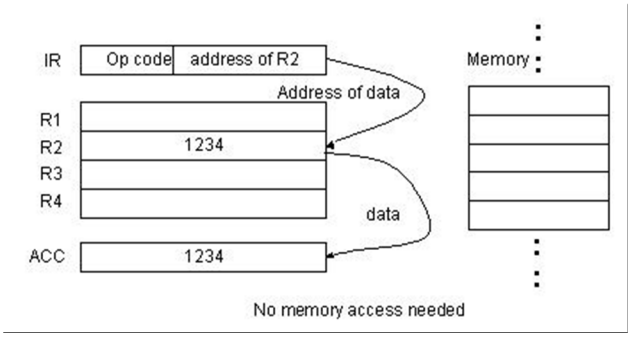


Example: lda R2

This load instruction specifies the address of the register and the operand is fetched from this register. As is clear from the diagram, no memory access is involved in this addressing mode.

REGISTER INDIRECT ADDRESSING MODE

In the register indirect mode, the address of memory location that contains the operand is in a CPU register. The address of this CPU register is encoded in the instruction. A large address space can be accessed using this addressing mode ($2^{\text{register size}}$ locations). It involves fewer memory accesses compared to indirect addressing.



Example: lda [R1]

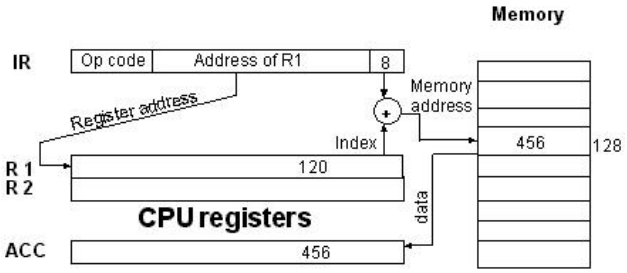
The address of the register that contains the address of memory location holding the operand is encoded in the instruction. There is one memory access involved.

Displacement addressing mode

The displacement-addressing mode is also called based or indexed addressing mode. Effective memory address is calculated by adding a constant (which is usually a part of the instruction) to the value in a CPU register. This addressing mode is useful for accessing arrays. The addressing mode may be called 'indexed' in the situation when the constant refers to the first element of the array (base) and the register contains the 'index'. Similarly, 'based' refers to the situation when the constant refers to the offset (displacement) of an array element with respect to the first element. The address of the first element is stored in a register.

Example: lda [R1 + 8]

In this example, R1 is the address of the register that holds a memory address, which is to be used to calculate the effective address of the operand. The constant (8) is added to this address held by the register and this effective address is used to retrieve the operand.

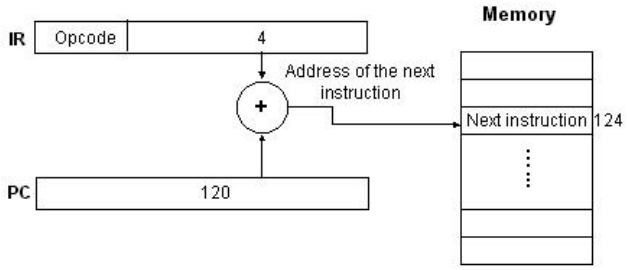


Relative addressing mode

The relative addressing mode is similar to the indexed addressing mode with the exception that the PC holds the base address. This allows the storage of memory operands at a fixed offset from the current instruction and is useful for 'short' jumps.

Example: jump 4

The constant offset (4) is a part of the instruction, and it is added to the address held by the Program Counter.



RISC and CISC architectures:

Generally, computers can be classified as being RISC machines or CISC machines. These concepts are explained in the following discussion.

RISC (Reduced instruction set computers)

RISC is more of a philosophy of computer design than a set of architectural features. The underlying idea is to reduce the number and complexity of instructions. However, new RISC machines have some instructions that may be quite complex and the number of instructions may also be large. The common features of RISC machines are

- **One instruction per clock period**

This is the most important feature of the RISC machines. Since the program execution depends on throughput and not on individual execution time, this feature is achievable by using pipelining and other techniques. In such a case, the goal is issuing an average of one instruction per cycle without increasing the cycle time.

- **Fixed size instructions**

Generally, the size of the instructions is 32 bits.

- **CPU accesses memory only for Load and Store operations**

This means that all the operands are in the CPU registers at the time these are used in an instruction. For this purpose, they are first brought into the CPU registers from the memory and later stored back through the load and store operation respectively.

- **Simple and few addressing modes**

The disadvantage associated with using complex addressing modes is that complex decoding is required to calculate these addresses, which reduces the processor performance as it takes significant time. Therefore, in RISC machines, few simple addressing modes are used.

- **Less work per instruction**

As the instructions are simple, less work is done per instruction, and hence the clock period T can be reduced.

- **Improved usage of delay slots**

A 'delay slot' is the waiting time for a load or store operation to access memory or for a branch instruction to access the target instruction. RISC designs allow the execution of the next instruction after these instructions are issued. If the program or compiler places an instruction in the delay slot that does not depend on the result of the previous instruction, the delay slot can be used efficiently. For the implementation of this feature, improved compilers are required that can check the dependencies of instructions before issuing them to utilize the delay slots.

- **Efficient usage of Pre-fetching and Speculative Execution Techniques**

Pre-fetching and speculative execution techniques are used with a pipelined architecture. Instruction pipelining means having multiple instructions in different stages of execution as instructions are issued before the previous instruction has completed its execution; pipelining will be studied in detail later. The RISC machines examine the instructions to check if operand fetches or branch instructions are involved. In such a case, the operands or the branch target instructions can be 'pre-fetched'. As instructions are issued before the preceding instructions have completed execution, the processor will not know in case of a conditional branch instruction, whether the condition will be met and the branch will be taken or not. But instead of waiting for this information to be available, the branch can be "speculated" as taken or not taken, and the instructions can be issued. Later if the speculation is found to be wrong, the results can be discarded and actual target instructions can be issued. These techniques help improve the performance of processors.

CISC (Complex Instruction Set Computers)

The complex instruction set computers does not have an underlying philosophy. The CISC machines have resulted from the efforts of computer designers to efficiently utilize memory and minimize execution time, yet add in more instruction formats and addressing modes. The common attributes of CISC machines are discussed below.

- **More work per instruction**

This feature was very useful at the time when memory was expensive as well as slow; it allows the execution of compact programs with more functionality per instruction.

- **Wide variety of addressing modes**

CISC machines support a number of addressing modes, which helps reduce the program instruction count. There are 14 addressing modes in MC68000 and 25 in MC68020.

- **Variable instruction lengths and execution times per instruction**

The instruction size is not fixed and so the execution times vary from instruction to instruction.

- **CISC machines attempt to reduce the "semantic gap"**

'Semantic gap' is the gap between machine level instruction sets and high-level language constructs. CISC designers believed that narrowing this gap by providing complicated instructions and complex-addressing modes would improve performance. The concept did not work because compiler writes did not find these "improvements" useful. The following are some of the disadvantages of CISC machines.

- **Clock period T, cannot be reduced beyond a certain limit**

When more capabilities are added to an instruction the CPU circuits required for the execution of these instructions become complex. This results in more stages of logic circuitry and adds propagation delays in signal paths.

This in turn places a limit on the smallest possible value of T and hence, the maximum value of clock frequency.

- **Complex addressing modes delay operand fetch from memory**

The operand fetch is delayed because more time is required to decode complex instructions.

- **Difficult to make efficient use of speedup techniques**

These speedup techniques include

- Pipelining
- Pre-fetching (Intel 8086 has a 6 byte queue)
- Super scalar operation
- Speculative execution

Lecture No. 3

Introduction to SRC Processor

Reading Material

Vincent P. Heuring & Harry F. Jordan	Chapter2, Chapter 3
Computer Systems Design and Architecture	2.3, 2.4, 3.1

Summary

- Measures of performance
- Introduction to an example processor SRC
- SRC Notation
- SRC features and instruction formats

Measures of performance:

Performance testing

To test or compare the performance of machines, programs can be run and their execution times can be measured. However, the execution speed may depend on the particular program being run, and matching it exactly to the actual needs of the customer can be quite complex. To overcome this problem, standard programs called "benchmark programs" have been devised. These programs are intended to approximate the real workload that the user will want to run on the machine. Actual execution time can be measured by running the program on the machines.

Commonly used measures of performance

The basic measure of performance of a machine is time. Some commonly used measures of this time, used for comparison of the performance of various machines, are

- Execution time
- MIPS
- MFLOPS
- Whetstones
- Dhrystones
- SPEC

Execution time

Execution time is simply the time it takes a processor to execute a given program. The time it takes for a particular program depends on a number of factors other than the performance of the CPU, most of which are ignored in this measure. These factors include waits for I/O, instruction fetch times, pipeline delays, etc.

The execution time of a program with respect to the processor, is defined as

Execution Time = IC x CPI x T

Where,

IC = instruction count

CPI = average number of system clock periods to execute an instruction T = clock period

Strictly speaking, (IC×CPI) should be the sum of the clock periods needed to execute each instruction. The manufacturers for each instruction in the instruction set usually provide such information. Using the average is a simplification.

MIPS (Millions of Instructions per Second)

Another measure of performance is the millions of instructions that are executed by the processor per second. It is defined as

$$MIPS = IC/ (ET \times 10^6)$$

This measure is not a very accurate basis for comparison of different processors. This is because of the architectural differences of the machines; some machines will require more instructions to perform the same job as compared to other machines. For example, RISC machines have simpler instructions, so the same job will require more instructions. This measure of performance was popular in the late 70s and early 80s when the VAX 11/780 was treated as a reference.

MFLOPS (Millions of Floating Point Instructions per Second)

For computation intensive applications, the floating-point instruction execution is a better measure than the simple instructions. The measure MFLOPS was devised with this in mind. This measure has two advantages over MIPS:

- Floating point operations are complex, and therefore, provide a better picture of the hardware capabilities on which they are run
- Overheads (operand fetch from memory, result storage to the memory, etc.) are effectively lumped with the floating point operations they support

Whetstones

Whetstone is the first benchmark program developed specifically as a benchmark program for performance measurement. Named after the Whetstone Algol compiler, this benchmark program was developed by using the statistics collected during the compiler development. It was originally an Algol program, but it has been ported to FORTRAN, Pascal and C. This benchmark has been specifically designed to test floating point instructions. The performance is stated in MWIPS (millions of Whetstone instructions per second).

Dhrystones

Developed in 1984, this is a small benchmark program to measure the integer instruction performance of processors, as opposed to the Whetstone's emphasis on floating point instructions. It is a very small program, about a hundred high-level-language statements, and compiles to about 1~ 1½ kilobytes of code.

Disadvantages of using Whetstones and Dhrystones

Both Whetstones and Dhrystones are now considered obsolete because of the following reasons.

- Small, fit in cache
- Obsolete instruction mix
- Prone to compiler tricks
- Difficult to reproduce results
- Uncontrolled source code

We should note that both the Whetstone and Dhrystone benchmarks are small programs, which encourage 'over-optimization', and can be used with optimizing compilers to distort results.

SPEC

SPEC, System Performance Evaluation Cooperative, is an association of a number of computer companies to define standard benchmarks for fair evaluation and comparison of different processors. The standard SPEC benchmark suite includes:

- A compiler
- A Boolean minimization program
- A spreadsheet program

- A number of other programs that stress arithmetic processing speed the latest version of these benchmarks is SPEC CPU2000.

Advantages

- It provides for ease of publication.
- Each benchmark carries the same weight.
- SPEC ratio is dimensionless.
- It is not unduly influenced by long running programs.
- It is relatively immune to performance variation on individual benchmarks.
- It provides a consistent and fair metric.

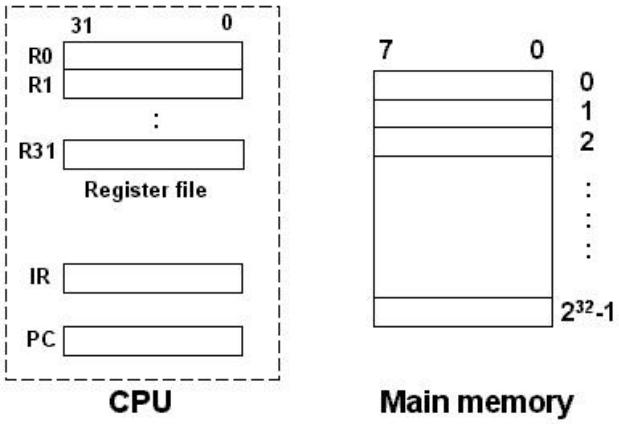
An example computer: the SRC: "simple RISC computer"

An example machine is introduced here to facilitate our understanding of various design steps and concepts in computer architecture. This example machine is quite simple, and leaves out a lot of details of a real machine, yet it is complex enough to illustrate the fundamentals.

SRC Introduction imp long

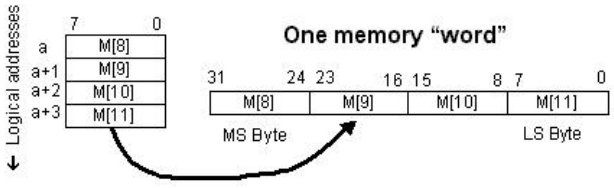
Attributes of the SRC

- The SRC contains 32 General Purpose Registers: R0, R1, …, R31; each register is of size 32-bits.
- Two special purpose registers are included: Program Counter (PC) and Instruction Register (IR)
- Memory word size is 32 bits
- Memory space size is 2^{32} bytes
- Memory organization is 2^{32} x 8 bits, this means that the memory is byte aligned
- Memory is accessed in 32 bit words (i.e., 4 byte chunks)
- Big-endian byte storage is used



Programmer's View of the SRC

The figure shows the attributes of the SRC; the 32, 32-bit registers that are a part of the CPU, the two additional CPU registers (PC & IR), and the main memory which is 2^{32} 1-byte cells.



SRC Notation

We examine the notation used for the SRC with the help of some examples.

- R[3] means contents of register
- 3 (R for register)
- M[8] means contents of memory location 8 (M for memory)
- A memory word at address 8 is defined as the 32 bits at address 8,9,10 and 11 in the memory. This is shown in the figure.
- A special notation for 32-bit memory words is M[8]<31…0>:=M[8] M[9] M[10] M[11] is used for concatenation.

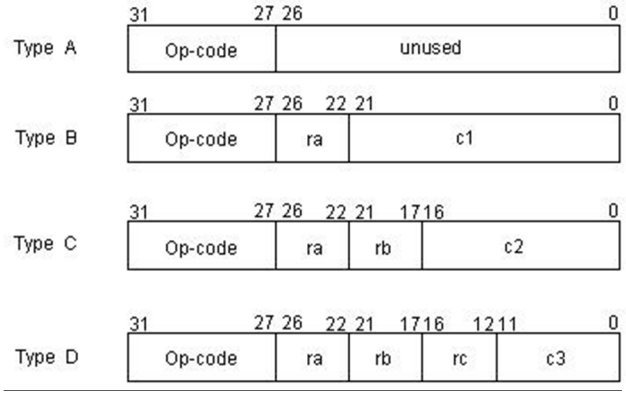
Some more SRC Attributes

- All instructions are 32 bits long (i.e., instruction size is 1 word)
- All ALU instructions have three operands
- The only way to access memory is through load and store operations
- Only a few addressing modes are supported

SRC: Instruction Formats

Four types of instructions are supported by the SRC. Their representation is given in the figure shown.

Before discussing these instruction types in detail, we take a look at the encoding of general purpose registers (the ra, rb and rc fields).



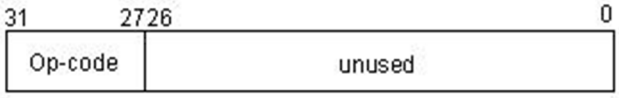
Encoding of the General Purpose Registers

The encoding for the general purpose registers is shown in the table; it will be used in place of ra, rb and rc in the instruction formats shown above. Note that this is a simple 5 bit encoding. ra, rb and rc are names of fields used as "place-holders", and can represent any one of these 32 registers. An exception is rb = 0; it does not mean the register R0, rather it means no operand. This will be explained in the following discussion.

Register	Code	Register	Code	Register	Code	Register	Code
R0	00000	R8	01000	R16	10000	R24	11000
R1	00001	R9	01001	R17	10001	R25	11001
R2	00010	R10	01010	R18	10010	R26	11010
R3	00011	R11	01011	R19	10011	R27	11011
R4	00100	R12	01100	R20	10100	R28	11100
R5	00101	R13	01101	R21	10101	R29	11101
R6	00110	R14	01110	R22	10110	R30	11110
R7	00111	R15	01111	R23	10111	R31	11111

Type A

Type A is used for only two instructions:

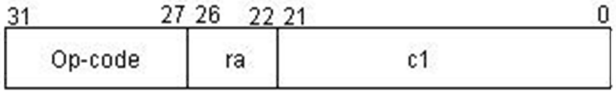


- No operation or nop, for which the op-code = 0. This is useful in pipelining
- Stop operation stop, the op-code is 31 for this instruction.

Both of these instructions do not need an operand (are 0-operand instructions).

Type B

Type B format includes three instructions; all three use relative addressing mode. These are

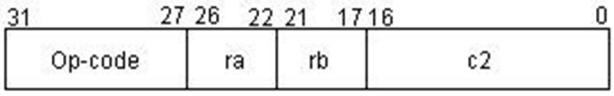


- The ldr instruction, used to load register from memory using a relative address. (op-code = 2).
  - Example:
    ldr R3, 56
    This instruction will load the register R3 with the contents of the memory location M [PC+56]
- The lar instruction, for loading a register with relative address (op-code = 6)

- Example: lar R3, 56
  This instruction will load the register R3 with the relative address itself (PC+56).

- The str is used to store register to memory using relative address (op-code = 4)
  - Example: str R8, 34
    This instruction will store the register R8 contents to the memory location M [PC+34]

The effective address is computed at run-time by adding a constant to the PC. This makes the instructions 're-locatable'.



Type C

Type C format has three load/store instructions, plus three ALU instructions. These load/ store instructions are

- ld, the load register from memory instruction (op-code = 1)
  - Example 1:
    ld R3, 56
    This instruction will load the register R3 with the contents of the memory location M [56]; the rb field is 0 in this instruction, i.e., it is not used. This is an example of direct addressing mode.
  - Example 2:
    ld R3, 56(R5)
    The contents of the memory location M [56+R [5]] are loaded to the register R3; the rb field ≠ 0. This is an instance of indexed addressing mode.
- la is the instruction to load a register with an immediate data value (which can be an address) (op-code = 5)
  - Example1:
    la R3, 56
    The register R3 is loaded with the immediate value 56. This is an instance of immediate addressing mode.
  - Example 2:
    la R3, 56(R5)
    The register R3 is loaded with the indexed address 56+R [5]. This is an example of indexed addressing mode.
- The st instruction is used to store register contents to memory (op-code = 3)
  - Example 1: st R8, 34
    This is the direct addressing mode; the contents of register R8 (R [8]) are stored to the memory location M [34]
  - Example 2: st R8, 34(R6)
    An instance of indexed addressing mode, M [34+R [6]] stores the contents of R8(R [8])

The ALU instructions are

- addi, immediate 2's complement addition (op-code = 13)
  - Example:
    addi R3, R4, 56
    R[3] ← R[4]+56 *(rb field = R4)*

- andi, the instruction to obtain immediate logical AND, (op-code = 42)
  - Example:
    andi R3, R4, 56

- R3 is loaded with the immediate logical AND of the contents of register R4 and 56 (constant value)

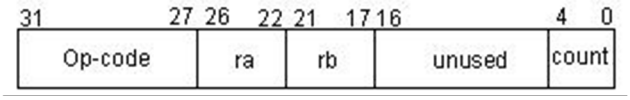
- ori, the instruction to obtain immediate logical OR (op-code = 23)
  - Example: ori R3, R4, 56

- R3 is loaded with the immediate logical OR of the contents of register R4 and 56(constant value)

Note:

1. Since the constant c2 field is 17 bits,
   - For direct addressing mode, only the first 2^{16} bytes of memory can be accessed (or the last 2^{16} bytes if c2 is negative)
   - In case of the la instruction, only constants with magnitudes less than $\pm 2^{16}$ can be loaded
   - During address calculation using c2, sign extension to 32 bits must be performed before the addition

2. **Type C instructions**, with some modifications, may also be used for shift instructions. Note the modification in the following figure.



The four shift instructions are

- shr is the instruction used to shift the bits right by using value in (5-bit) c3 field(shift count) (op-code = 26)
  - Example: shr R3, R4, 7
    shift R4 right 7 times in to R3. Immediate addressing mode is used.

- shra, arithmetic shift right by using value in c3 field (op-code = 27)
  - Example: shra R3, R4, 7
    This instruction has the effect of shift R4 right 7 times in to R3. Immediate addressing mode is used.

- The shl instruction is for shift left by using value in (5-bit) c3 field (op-code = 28)
  - Example: shl R8, R5, 6
    shift R5 left 6 times in to R8. Immediate addressing mode is used.

- shc, shift left circular by using value in c3 field (op-code = 29)
  - Example: shc R3, R4, 3
    shift R4 circular 3 times in to R3. Immediate addressing mode is used.

Lecture No. 4

ISA and Instruction Formats

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 2
Computer Systems Design and Architecture — 2.3, 2.4, slides

Summary

- Introduction to ISA and instruction formats
- Coding examples and Hand assembly

An example computer: the SRC: "simple RISC computer"

An example machine is introduced here to facilitate our understanding of various design steps and concepts in computer architecture. This example machine is quite simple, and leaves out a lot of details of a real machine, yet it is complex enough to illustrate the fundamentals.

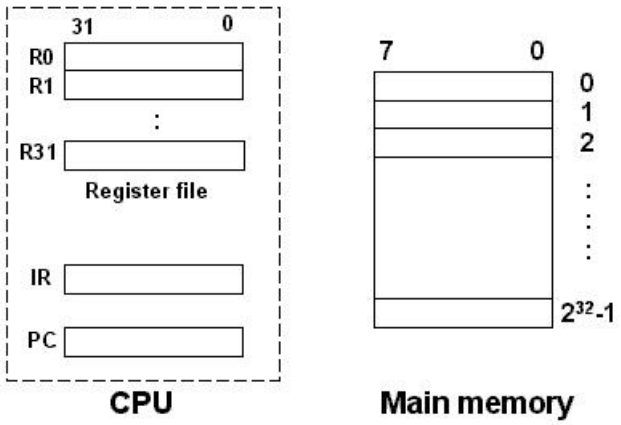
SRC Introduction

Attributes of the SRC

- The SRC contains 32 General Purpose Registers: R0, R1… R31; each register is of size 32-bits.
- Two special purpose registers are included: Program Counter (PC) and Instruction Register (IR)
- Memory word size is 32 bits
- Memory space size is 2^{32} bytes
- Memory organization is 2^{32} x 8 bits, this means that the memory is byte aligned
- Memory is accessed in 32 bit words (i.e., 4 byte chunks)
- Big-endian byte storage is used

Programmer's View of the SRC

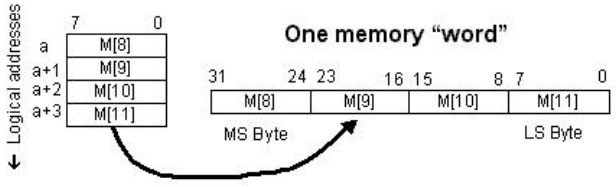
The figure below shows the attributes of the SRC; the 32, 32-bit registers that are a part of the CPU, the two additional CPU registers (PC & IR), and the main memory which is 2^{32} 1-byte cells.



SRC Notation

We examine the notation used for the SRC with the help of some examples.

- R[3] means contents of register 3 (R for register)
- M[8] means contents of memory location 8 (M for memory)



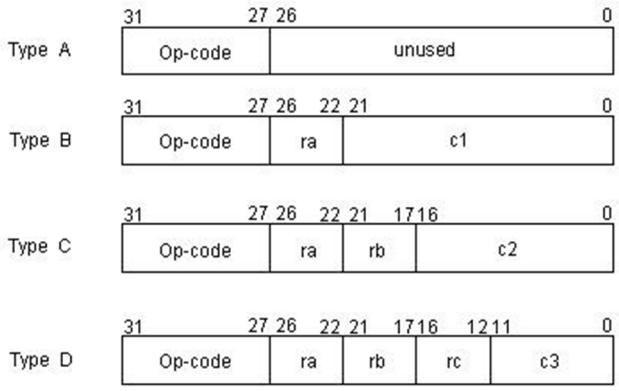
- A memory word at address 8 is defined as the 32 bits at address 8,9,10 and 11 in the memory. This is shown in the figure below.
- A special notation for 32-bit memory words is M[8]<31…0>:=M[8] M[9] M[10] M[11] is used for concatenation.

Some more SRC Attributes

- All instructions are 32 bits long (i.e., instruction size is 1 word)
- All ALU instructions have three operands
- The only way to access memory is through load and store operations
- Only a few addressing modes are supported

SRC: Instruction Formats

Four types of instructions are supported by the SRC. Their representation is given in the following figure. Before discussing these instruction types in detail, we take a look at the encoding of general-purpose registers (the ra, rb and rc fields).



Encoding of the General Purpose Registers

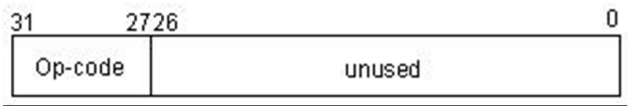
The encoding for the general purpose registers is shown in the following table; it will be used in place of ra, rb and rc in the instruction formats shown above. Note that this is a simple 5 bit encoding. ra, rb and rc are names of fields used as "place-holders", and can represent any one of these 32 registers. An exception is rb = 0; it does not mean the register R0, rather it means no operand. This will be explained in the following discussion.

Register	Code	Register	Code	Register	Code	Register	Code
R0	00000	R8	01000	R16	10000	R24	11000
R1	00001	R9	01001	R17	10001	R25	11001
R2	00010	R10	01010	R18	10010	R26	11010
R3	00011	R11	01011	R19	10011	R27	11011
R4	00100	R12	01100	R20	10100	R28	11100
R5	00101	R13	01101	R21	10101	R29	11101
R6	00110	R14	01110	R22	10110	R30	11110
R7	00111	R15	01111	R23	10111	R31	11111

Type A

Type A is used for only two instructions:

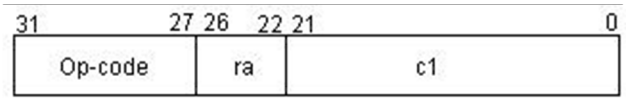
- No operation or nop, for which the op-code = 0. This is useful in pipelining
- Stop operation stop, the op-code is 31 for this instruction.



Both of these instructions do not need an operand (are 0-operand instructions).

Type B

Type B format includes three instructions; all three use relative addressing mode.

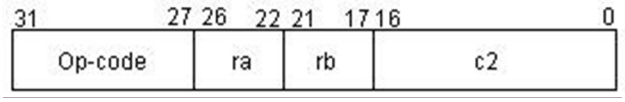


These are

- The ldr instruction, used to load register from memory using a relative address. (op-code = 2).
  - Example: ldr R3, 56

    This instruction will load the register R3 with the contents of the memory location M [PC+56]
- The lar instruction, for loading a register with relative address (op-code = 6)
  - Example: lar R3, 56
    This instruction will load the register R3 with the relative address itself (PC+56).
- The str is used to store register to memory using relative address (op-code = 4)
  - Example: str R8, 34
    This instruction will store the register R8 contents to the memory location M [PC+34]
- The effective address is computed at run-time by adding a constant to the PC. This makes the instructions 're-locatable'.

Type C



Type C format has three load/store instructions, plus three ALU instructions. These load/ store instructions are

- ld, the load register from memory instruction (op-code = 1)
  - Example 1:
    ld R3, 56
    This instruction will load the register R3 with the contents of the memory location M [56]; the rb field is 0 in this instruction, i.e., it is not used. This is an example of direct addressing mode.
  - Example 2: ld R3, 56(R5)
    The contents of the memory location M [56+R [5]] are loaded to the register R3; the rb field ≠ 0. This is an instance of indexed addressing mode.
- la is the instruction to load a register with an immediate data value (which can be an address) (op-code = 5)
  - Example1: la R3, 56

    The register R3 is loaded with the immediate value 56. This is an instance of immediate addressing mode.
  - Example 2: la R3, 56(R5)

    The register R3 is loaded with the indexed address 56+R [5]. This is an example of indexed addressing mode.
- The st instruction is used to store register contents to memory (op-code = 3)
  - Example 1: st R8, 34

    This is the direct addressing mode; the contents of register R8 (R [8]) are stored to the memory location M [34]
  - Example 2: st R8, 34(R6)

    An instance of indexed addressing mode, M [34+R [6]] stores the contents of R8 (R [8])

The ALU instructions are

- addi, immediate 2's complement addition (op-code = 13)
  - Example: addi R3, R4, 56

  R[3] ← R[4]+56 *(rb field = R4)*

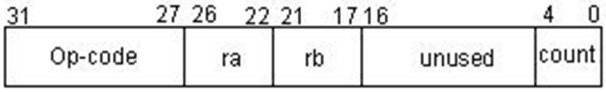
- andi, the instruction to obtain immediate logical AND, (op-code = 21)
  - Example: andi R3, R4, 56

  R3 is loaded with the immediate logical AND of the contents of register R4 and 56(constant value)
- ori, the instruction to obtain immediate logical OR (op-code = 23)
  - Example: ori R3, R4, 56

  R3 is loaded with the immediate logical OR of the contents of register R4 and 56(constant value)

Note:

1. Since the constant c2 field is 17 bits,
   - For direct addressing mode, only the first 2^{16} bytes of memory can be accessed (or the last 2^{16} bytes if c2 is negative)
   - In case of the la instruction, only constants with magnitudes less than $\pm 2^{16}$ can be loaded
   - During address calculation using c2, sign extension to 32 bits must be performed before the addition
2. Type C instructions, with some modifications, may also be used for shift instructions. Note the modification in the following figure.



The four shift instructions are

- shr is the instruction used to shift the bits right by using value in (5-bit) c3 field(shift count) (op-code = 26)
  - Example: shr R3, R4, 7

  Shift R4 right 7 times in to R3 and shifts zeros in from the left as the value is shifted right. Immediate addressing mode is used.

- shra, arithmetic shift right by using value in c3 field (op-code = 27)
  - Example: shra R3, R4, 7

  This instruction has the effect of shift R4 right 7 times in to R3 and copies the msb into the word on left as contents are shifted right. Immediate addressing mode is used.

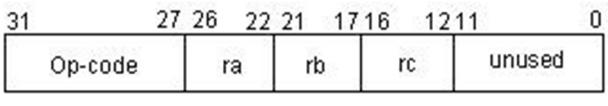
- The shl instruction is for shift left by using value in (5-bit) c3 field (op-code = 28)
  - Example: shl R8, R5, 6

  Shift R5 left 6 times in to R8 and shifts zeros in from the right as the value is shifted left. Immediate addressing mode is used.
- shc, shift left circular by using value in c3 field (op-code = 29)
  - Example: shc R3, R4, 3

  Shift R4 circular 3 times in to R3 and copies the value shifted out of the register on the left is placed back into the register on the right. Immediate addressing mode is used.

Type D

Type D includes four ALU instructions, four register based shift instructions, two logical instructions and two branch instructions.

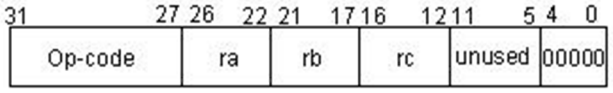


The four ALU instructions are given below

- add, the instruction for 2's complement register addition (op-code = 12)
  - Example:
    add R3, R5, R6
    Result of 2's complement addition R[5] + R[6] is stored in R3. Register addressing mode is used.

- sub , the instruction for 2's complement register subtraction (op-code = 14)
  - Example:
    sub R3, R5, R6
    R3 will store the 2's complement subtraction, R[5] - R[6]. Register addressing mode is used.
- and, the instruction for logical AND operation between registers (op-code = 20)
  - Example:
    and R8, R3, R4
    R8 will store the logical AND of registers R3 and R4. Register addressing mode is used.
- or, the instruction for logical OR operation between registers (op-code = 22)
  - Example:
    or R8, R3, R4
    R8 is loaded with the value R[3] v R[4], the logical OR of registers R3 and R4. Register addressing mode is used.

The four register based shift instructions use register addressing mode. These use a modified form of type D, as shown in figure

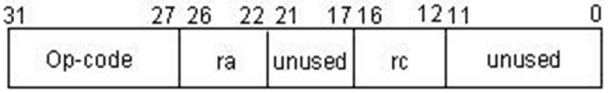


- shr, shift right by using value in register rc (op-code = 26)
  - Example: shr R3, R4, R5
    This instruction will shift R4 right in to R3 using number in R5
- shra, the arithmetic shift right by using register rc (op-code = 27)
  - Example:
    shra R3, R4, R5
    A shift of R4 right using R5, and the result is stored in R3
- shl is shift left by using register rc (op-code = 28)
  - Example:
    shl R8, R5, R6

  The instruction shifts R5 left in to R8 using number in R6
- shc, shifts left circular by using register rc (op-code = 29)
  - Example:
    shc R3, R4, R6
    This instruction will shift R4 circular in to R3 using value in R6

The two logical instructions also use a modified form of the Type D, and are the following.

- neg stores the 2's complement of register rc in ra (op-code = 15)
  - Example: neg R3, R4
    Negates (obtains 2's complement) of R4 and stores in R3. 2-address format and register addressing mode is used.



- not stores the 1's complement of register rc in ra (op-code = 24)
  - Example:
    not R3, R4
    Logically inverts R4 and stores in R3. 2-address format with register addressing mode is used.

Type D has two-branch instruction, modified forms of type D.

- br , the instruction to branch to address in rb depending on the condition in rc. There are five possible conditions, explained through examples. (op-code = 8). All branch instructions use register-addressing mode.

  - Example 1:
    brzr R3, R4
    Branch to address in R3 (if R4 == 0)
  - Example 2:
    brnz R3, R4
    Branch to address in R3 (if R4 ≠ 0)
  - Example 3:
    brpl R3, R4
    Branch to address in R3 (if R4 ≥ 0)
  - Example 4:
    brmi R3, R4
    Branch to address in R3 (if R4 < 0)

  - Example 5:
    br R3, R4
    Branch to address in R3 (unconditional)

- Brl the instruction to branch to address in rb depending on condition in rc. Additionally, it copies the PC in to ra before branching (op-code = 9)

  - Example 1:
    brlzr R1,R3, R4
    R1 will store the contents of PC, then branch to address in R3 (if R4 == 0)
  - Example 2:
    brlnz R1,R3, R4
    R1 stores the contents of PC, then a branch is taken, to address in R3 (if R4 ≠ 0)
  - Example 3:
    brlpl R1,R3, R4
    R1 will store PC, then branch to address in R3 (if R4≥ 0)
  - Example 4:
    brlmi R1,R3, R4
    R1 will store PC and then branch to address in R3 (if R4 < 0)

Mnemonic	c3<2..0>	Branch Condition
brlnv	000	Link but never branch*
br, brl	001	Unconditional branch
brzr, brlzr	010	Branch if rc is zero
brnz, brlnz	011	Branch if rc is not zero
brpl, brlpl	100	Branch if rc is positive
brmi, brlmi	101	Branch if rc is negative

- Example 5:
  brl R1,R3, R4
  R1 will store PC, then it will ALWAYS branch to address in R3
- Example 6:
  brlnv R1,R3, R4
  R1 just stores the contents of PC but a branch is not taken (NEVER BRANCH)

In the modified type D instructions for branch, the bits <2...0> are used for specifying the condition; these condition codes are shown in the table.

The SRC Instruction Summary

The instructions implemented by the SRC are listed, grouped on functionality basis.

Functional Groups of Instructions

Logic		Opcode				
Shift right by count	shr	1	1	0	1	0
Shift right by count in a register	shr	1	1	0	1	0
AShift right by count	shra	1	1	0	1	1
AShift right by count in a register	shra	1	1	0	1	1
Shift left by count	shl	1	1	1	0	0
Shift left by count in a register	shl	1	1	1	0	0
Shift circ. by count	shc	1	1	1	0	1
Shift circ. by count in a register	shc	1	1	1	0	1

Control						
		opcode				
No operation	nop	0	0	0	0	0
Halt Machine	stop	1	1	1	1	1
Data Transfer						
		Opcode				
Load register	ld	0	0	0	0	1
Load relative register	ldr	0	0	0	1	0
Store register	st	0	0	0	1	1
Store relative register	str	0	0	1	0	0
Load address	la	0	0	1	0	1
Load relative address	lar	0	0	1	1	0

Alphabetical Listing based on SRC Mnemonics

Notice that the op code field for all br instructions is the same. The difference is in the condition code field, which is in effect, an op code extension.

Mnemonic	31	30	29	28	27
	Opcode				
add	0	1	1	0	0
addi	0	1	1	0	1
and	1	0	1	0	0
andi	1	0	1	0	1
br	0	1	0	0	0
brl	0	1	0	0	1
brlmi	0	1	0	0	1
brlnv	0	1	0	0	1
brlnz	0	1	0	0	1
brlpl	0	1	0	0	1
brlzr	0	1	0	0	1
brmi	0	1	0	0	0
brnv	0	1	0	0	0
brnz	0	1	0	0	0
brpl	0	1	0	0	0
brzr	0	1	0	0	0

la	0	0	1	0	1
lar	0	0	1	1	0
ld	0	0	0	0	1
ldr	0	0	0	1	0
neg	0	1	1	1	1
nop	0	0	0	0	0
not	1	1	0	0	0
or	1	0	1	1	0
ori	1	0	1	1	1
shc	1	1	1	0	1
shc	1	1	1	0	1
shl	1	1	1	0	0
shl	1	1	1	0	0
shr	1	1	0	1	0
shr	1	1	0	1	0
shra	1	1	0	1	1
shra	1	1	0	1	1
st	0	0	0	1	1
stop	1	1	1	1	1
str	0	0	1	0	0
sub	0	1	1	1	0

Examples

Some examples are studied in this section to enhance the student's understanding of the SRC.

Branching Instructions						
Branch when ...		Opcode				
never	brnv	0	1	0	0	0
always	br	0	1	0	0	0
zero	brzr	0	1	0	0	0
non zero	brnz	0	1	0	0	0
postive (including zero)	brpl	0	1	0	0	0
negative	brmi	0	1	0	0	0
Branch and link when ...						
never	brlnv	0	1	0	0	1
always	brl	0	1	0	0	1
zero	brlzr	0	1	0	0	1
non zero	brlnz	0	1	0	0	1
postive (including zero)	brlpl	0	1	0	0	1
negative	brlmi	0	1	0	0	1

Example 1: Expression Evaluation

Write an SRC assembly language program to evaluate the expression:
z = 4(a +b) – 16(c+58)
Your code should not change the source operands.

Solution A: Notice that the SRC does not have a multiply instruction. We will make use of the fact that multiplication with powers of 2 can be achieved by repeated shift left operations. A possible solution is give below:

```
ld R1, c                ; c is a label used for a memory location
addi R3, R1, 58         ; R3 contains (c+58)
shl R7, R3, 4           ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5          ; R6 contains (a+b)
shl R8, R6, 2           ; R8 contains 4(a+b)
sub R9, R8, R7          ; the result is in R9
st R9, z                ; store the result in memory location z
```

Note:
The memory labels a, b, c and z can be defined by using assembler directives like .dw or .db, etc. in the source file.
A semicolon ';' is used for comments in assembly language.

Solution B:
We may solve the problem by assuming that a multiply instruction, similar to the add instruction, exists in the instruction set of the SRC. The shl instruction will be replaced by the mul instruction as given below.

```
ld R1, c                ; c is a label used for a memory location
addi R3, R1, 58         ; R3 contains (c+58)
mul R7, R3, 4           : R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5          ; R6 contains (a+b)
mul R8, R6, 2           ; R8 contains 4(a+b)
sub R9, R8, R7          ; the result is in R9
st R9, z
                        ; store the result in memory location z
```

Note:
The memory labels a, b, c and z can be defined by using assembler directives like .dw or .db, etc. in the source file.

Solution C:
We can perform multiplication with a multiplier that is not a power of 2 by doing addition in a loop. The number of times the loop will execute will be equal to the multiplier.

Example 2: Hand Assembly

Convert the given SRC assembly language program in to an equivalent SRC machine language program.

```
ld R1, c                ; c is a label used for a memory location
```

```
addi R3, R1, 58          ; R3 contains (c+58)
shl R7, R3, 4            ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5           ; R6 contains (a+b)
shl R8, R6, 2            ; R8 contains 4(a+b)
sub R9, R8, R7           ; The result is in R9
st R9, z                 ; store the result in memory location z
```

Note:
This program uses memory labels a,b,c and z. We need to define them for the assembler by using assembler directives like .dw or .equ etc. in the source file.

Assembler Directives
Assembler directives, also called pseudo op-codes, are commands to the assembler to direct the assembly process. The directives may be slightly different for different assemblers. All the necessary directives are available with most assemblers. We explain the directives as we encounter them. More information on assemblers can be looked up in the assembler user manuals.

Source program with directives

```
            .ORG    200     ; start the next line at address 200
a:          .DW     1       ; reserve one word for the label a in the memory
b:          .DW     1       ; reserve a word for b, this will be at address 204
c:          .DW     1       ; reserve a word for c, will be at address 208
z:          .DW     1       ; reserve one word for the result
```

```
.ORG 400 ; start the code at address 400 ; all numbers are in decimal unless otherwise stated
ld R1, c                        ; c is a label used for a memory location
addi R3, R1, 58 ; R3 contains (c+58)
shl R7, R3, 4                   ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5                  ; R6 contains (a+b)
shl R8, R6, 2                   ; R8 contains 4(a+b)
sub R9, R8, R7                  ; the result is in R9
st R9, z                        ; store the result in memory location z
```

This is the way an assembly program will appear in the source file. Most assemblers require that the file be saved with an .asm extension.

Solution:
Observe the first line of the program
.ORG 200 ; start the next line at address 200
This is a directive to let the following code/ variables 'originate' at the specified address of the memory, 200 in this case.
Variable statements and another .ORG directive follow the .ORG directive.

```
a:      .DW         1; reserve one word for the label a in the memory
b:      .DW         1; reserve a word for b, this will be at address 204
c:      .DW         1; reserve a word for c, will be at address 208
z:      .DW         1; reserve one word for the result
        .ORG 400     ; start the code at address 400
```

We conclude the following from the above statements: The code starts at address 400 and each instruction takes 32 bits in the memory. The memory map for the program is shown in given table.

Memory Map for the SRC example program

Memory Address	Memory Contents
200	unknown
204	unknown
208	unknown
212	unknown
...	...
400	ld R1, c
404	addi R3, R1, 58
408	shl R7, R3, 4
412	ld R4, a
416	ld R5, b
420	add R6, R4, R5
424	shl R8, R6, 2
428	sub R9, R7, R8
432	st R9, z

Label	Address	Value
a	200	unknown
b	204	unknown
c	208	unknown
z	212	unknown

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
....	....	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	
408	shl R7, R3, 4	
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

We have to convert these instructions to machine language. Let us start with the first instruction:

ld R1, c

Notice that this is a type C instruction with the rb field missing.

1. We pick the op-code for this load instruction from the SRC instruction tables given in the SRC instruction summary section. The op-code for the load register 'ld' instruction is 00001.
2. Next we pick the register code corresponding to register R1 from the register table (given in the section 'encoding of general purpose registers'). The register code for R1 is 00001.
3. The rb field is missing, so we place zeros in the field: 00000
4. The value of c is provided by the assembler, and should be converted to 17 bits. As c has been assigned the memory address 208, the binary value to be encoded is 00000 0000 1101 0000.
5. So the instruction ld R1, c is 00001 00001 00000 00000 0000 1101 0000 in the machine language.
6. The hexadecimal representation of this instruction is 0 8 4 0 0 0 D 0 h.

We can update the memory map with these values.

We consider the next instruction,

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
....	....	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	68C2003A h
408	shl R7, R3, 4	
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

addi R3, R1, 58.

Notice that this is a type C instruction.

1. We pick the op-code for the instruction addi from the SRC instruction table. It is 01101

2. We pick the register codes for the registers R3 and R1, these codes are 00011 and 00001 respectively

3. For the immediate data, 58, we use the binary value, 00000 0000 0011 1010

4. So the complete instruction becomes: 01101 00011 00001 00000 0000 0011 1010

5. The hexadecimal representation of the instruction is 6 8 C 2 0 0 3 A h

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

We update the memory map, as shown in table.

The next instruction is **shl R7,R3, 4**, at address 408.

Again, this is a type C instruction.

1. The op-code for the instruction shl is picked from the SRC instruction table. It is 11100

2. The register codes for the registers R7 and R3 from the register table are 00111 and 00011 respectively

3. For the immediate data, 4, the corresponding binary value 00000 0000 0000 0100 is used.

4. We can place these codes in accordance with the type C instruction format to obtain the complete instruction: 11100 00111 00011 00000 0000 0000 0100

5. The hexadecimal representation of the instruction is E1C60004

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	090000C8h
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

The memory map is updated, as shown in table.

The next instruction, **ld R4, a**, is also a type C instruction. Rb field is missing in this instruction. To obtain the machine equivalent, we follow the steps given below.

1. The op-code of the load instruction 'ld' is 00001

2. The register code corresponding to the register R4 is obtained from the register table, and it is 00100

3. As the 5 bit rb field is missing, we can encode zeros in its place: 00000

4. The value of a is provided by the assembler, and is converted to 17 bits. It has been assigned the memory address 200, the binary equivalent of which is: 00000 0000 1100 1000

5. The complete instruction becomes: 00001 00100 00000 00000 0000 1100 1000

6. The hexadecimal equivalent of the instruction is 0 9 0 0 0 0 C 8 h

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	090000C8h
416	ld R5, b	094000CCh
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

Memory map can be updated with this value.

The next instruction is also a load type C instruction, with the rb field missing.

ld R5, b

The machine language conversion steps are

1. The op-code of the load instruction is obtained from the SRC instruction table; it is 00001

2. The register code for R5, obtained from the register table, is 00101

3. Again, the 5 bit rb field is missing. We encode zeros in its place: 00000

4. The value of label b is provided by the assembler, and should be converted to 17 bits. It has been assigned the memory address 204, so the binary value is: 00000 0000 1100 1100

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
....	...	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	68C2003A h
408	shl R7, R3, 4	E1C60004 h
412	ld R4, a	090000C8 h
416	ld R5, b	094000CC h
420	add R6, R4, R5	61885000 h
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

5. The complete instruction is: 00001 00101 00000 00000 0000 1100 1100
6. The hexadecimal value of this instruction is 0 9 4 0 0 0 C C h

Memory map is then updated with this value.

The next instruction is a type D-add instruction, with the constant field missing:

add R6,R4,R5

The steps followed to obtain the assembly code for this instruction are

1. The op-code of the instruction is obtained from the SRC instruction table; it is 01100
2. The register codes for the registers R6, R4 and R5 are obtained from the register table; these are 00110, 00100 and 00101 respectively.
3. The 12 bit constant field is unused in this instruction, therefore we encode zeros in its place: 0000 0000 0000
4. The complete instruction becomes: 01100 00110 00100 00101 0000 0000 0000
5. The hexadecimal value of the instruction is 6 1 8 8 5 0 0 0 h

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
....	...	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	68C2003A h
408	shl R7, R3, 4	E1C60004 h
412	ld R4, a	090000C8 h
416	ld R5, b	094000CC h
420	add R6, R4, R5	61885000 h
424	shl R8, R6, 2	E20C0002 h
428	sub R9, R7, R8	
432	st R9, z	

Memory map is then updated with this value.

The instruction **shl R8,R6, 2** is a type C instruction with the rc field missing. The steps taken to obtain the machine code of the instruction are

1. The op-code of the shift left instruction 'shl', obtained from the SRC instruction table, is 11100
2. The register codes of R8 and R6 are 01000 and 00110 respectively
3. Binary code is used for the immediate data 2: 00000 0000 0000 0010
4. The complete instruction becomes: 11100 01000 00110 00000 0000 0000 0010
5. The hexadecimal equivalent of the instruction is E 2 0 C 0 0 0 2

Memory map is then updated with this value.

The instruction at the memory address 428 is **sub R9, R7, R8**. This is a type D instruction.

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
....	...	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	68C2003A h
408	shl R7, R3, 4	E1C60004 h
412	ld R4, a	090000C8 h
416	ld R5, b	094000CC h
420	add R6, R4, R5	61885000 h
424	shl R8, R6, 2	E20C0002 h
428	sub R9, R7, R8	724E8000 h
432	st R9, z	

We decode it into the machine language, as follows:

1. The op-code of the subtract instruction 'sub' is 01110
2. The register codes of R9, R7 and R8, obtained from the register table, are 01001, 00111 and 01000 respectively
3. The 12 bit immediate data field is not used, zeros are encoded in its place: 0000 0000 0000
4. The complete instruction becomes: 01110 01001 00111 01000 0000 0000 0000
5. The hexadecimal equivalent is 7 2 4 E 8 0 0 0 h We again update the memory map

The last instruction is is a type C instruction with the rb field missing:

st R9, z

The machine equivalent of this instruction is obtained through the following steps:

1. The op-code of the store instruction ‘st’, obtained from the SRC instruction table, is 00011
2. The register code of R9 is 01001
3. Notice that there is no register coded in the 5 bit rb field, therefore, we encode zeros: 00000The value of the label z is provided by the assembler, and should be converted to 17 bits. Notice that the memory address assigned to z is 212. The 17 bit binary equivalent is: 00000 0000 1101 0100
4. The complete instruction becomes: 00011 01001 00000 00000 0000 1101 0100
5. The hexadecimal form of this instruction is 1 A 4 0 0 0 D 4 h

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0 h
404	addi R3, R1, 58	68C2003A h
408	shl R7, R3, 4	E1C60004 h
412	ld R4, a	090000C8 h
416	ld R5, b	094000CC h
420	add R6, R4, R5	61885000 h
424	shl R8, R6, 2	E20C0002 h
428	sub R9, R7, R8	724E8000 h
432	st R9, z	1A4000D4 h

The memory map, after the conversion of all the instructions, is We have shown the memory map as an array of 4 byte cells in the above solution. However, since the memory of the SRC is arranged in 8 bit cells (i.e. memory is byte aligned), the real representation of the memory map is:

Memory Address	Memory contents
...	...
400	08 h
401	40 h
402	00 h
403	D0 h
404	68 h
405	C2 h
406	00 h
407	3A h
408	E1 h
409	C6 h
410	00 h
411	04 h
412	09 h
413	00 h
414	00 h
415	C8 h
416	09 h
417	40 h
...	...

Example 3: SRC instruction analysis

Identify the formats of following SRC instructions and specify the values in the fields

Instruction	format	ra	rb	rc	c1	c2	c3
neg r1, r2							
add r0,r2,r3							
nop							
ld r2,6							
shl r0,r1,3							

Solution:

Instruction	format	ra	rb	rc	c1	c2	c3
neg r1, r2	D	r1	–	r2	–	–	–
add r0,r2,r3	D	r0	r2	r3	–	–	–
nop	A	–	–	–	–	–	–
ld r2,6	C	r2	–	–	6	–	–
shl r0,r1,3	C	r0	r1	–	–	–	3

Lecture No. 5

Description of SRC in RTL

Reading Material

Handouts Slides

Summary

- Reverse Assembly
- Description of SRC in the form of RTL
- Behavioral and Structural description in terms of RTL

Reverse Assembly

Typical Problem:

Given a machine language instruction for the SRC, it may be required to find the equivalent SRC assembly language instruction

Example:

Reverse assemble the following SRC machine language instructions:

68C2003A h

E1C60004 h

61885000 h

724E8000 h

1A4000D4 h

084000D0 h

Solution:

1. Write the given hexadecimal instruction in binary form 68C2003A h → 0110 1000 1100 0010 0000 0000 0011 1010 b
2. Examine the first five bits of the instruction, and pick the corresponding mnemonic from the SRC instruction set listing arranged according to ascending order of op-codes 01101 b → 13 d → addi → add immediate
3. Now we know that this instruction uses the type C format, the two 5-bit fields after the op-code field represent the destination and the source registers respectively, and that the remaining 17-bits in the instruction represent a constant

01101	00011	00001	00000 0000 0011 1010 b
op-code	ra field	rb field	17-bit c1 field
↓	↓	↓	↓
addi	R3	R1	3A h=58 d

4. Therefore, the assembly language instruction is

addi R3, R1, 58

Summary

Given machine language instruction	Equivalent assembly language instruction
68C2003A h	addi R3, R1, 58
E1C60004 h	
61885000 h	
724E8000 h	
1A4000D4 h	
084000D0 h	

We can do it a bit faster now! **Step1:** Here is step1 for all instructions

Given instruction in hexadecimal	Equivalent instruction in binary
E1C60004 h	1110 0001 1100 0110 0000 0000 0000 0100 b
61885000 h	0110 0001 1000 1000 0101 0000 0000 0000 b
724E8000 h	0111 0010 0100 1110 1000 0000 0000 0000 b
1A4000D4 h	0001 1010 0100 0000 0000 0000 1101 0100 b
084000D0 h	0000 1000 0100 0000 0000 0000 1101 0000 b

Step 2: Pick up the op code for each instruction

Given instruction in hexadecimal	Op-code field	mnemonic	
E1C60004 h	1110 0 b	shl	
61885000 h	0110 0 b	add	
724E8000 h	0111 0 b	sub	
1A4000D4 h	0001 1 b	st	
084000D0 h	0000 1 b	ld	

Step 3: Determine the instruction type for each instruction

Given instruction in hexadecimal	mnemonic	Instruction type
E1C60004 h	shl	
61885000 h	add	
724E8000 h	sub	
1A4000D4 h	st	
084000D0 h	ld	

The meaning of the remaining fields will depend on the instruction type (i.e., the instruction format)

Summary

Given machine language instruction	Equivalent assembly language instruction
68C2003A h	addi R3, R1, 58
E1C60004 h	
61885000 h	
724E8000 h	
1A4000D4 h	
084000D0 h	

Note: rest of the fields of above given tables are left as an exercise for students. **Using RTL to describe the SRC**

RTL stands for Register Transfer Language. The Register Transfer Language provides a formal way for the description of the behavior and structure of a computer. The RTL facilitates the design process of the computer as it provides a precise, mathematical representation of its functionality. In this section, a Register Transfer Language is presented and introduced, for the SRC (Simple 'RISC' Computer), described in the previous discussion.

Behavioral RTL

Behavioral RTL is used to describe the 'functionality' of the machine only, i.e. what the machine does.

Structural RTL

Structural RTL describes the 'hardware implementation' of the machine, i.e. how the functionality made available by the machine is implemented.

Behavioral versus Structural RTL:

In computer design, a top-down approach is adopted. The computer design process typically starts with defining the behavior of the overall system. This is then broken down into the behavior of the different modules. The process continues, till we are able to define, design and implement the structure of the individual modules. Behavioral RTL is used for describing the behavior of machine whereas structural RTL is used to define the structure of machine, which brings us to the some more hardware features.

Using RTL to describe the static properties of the SRC

In this section we introduce the RTL by using it to describe the various static properties of the SRC.

Specifying Registers

The format used to specify registers is

Register Name<register bits>

For example, IR<31..0> means bits numbered 31 to 0 of a 32-bit register named "IR" (Instruction Register).

"Naming" using the := naming operator:

The **:=** operator is used to 'name' registers, or part of registers, in the Register Transfer Language. It does not create a new register; it just generates another name, or "alias" for an already existing register or part of a register. For example,

Op<4..0>: = IR<31..27> means that the five most significant bits of the register IR will be called op, with bits 4..0.

Fields in the SRC instruction

In this section, we examine the various fields of an SRC instruction, using the RTL.

op<4..0>: = IR<31..27>; operation code field

The five most significant bits of an SRC instruction, (stored in the instruction register in this example), are named op, and this field is used for specifying the operation. ra<4..0>: = IR<26..22>; target register field

The next five bits of the SRC instruction, bits 26 through 22, are used to hold the address of the target register field, i.e., the result of the operation performed by the instruction is stored in the register specified by this field.

rb<4..0>: = IR<21..17>; operand, address index, or branch target register

The bits 21 through 17 of the instruction are used for the rb field. rb field is used to hold an operand, an address index, or a branch target register.

rc<4..0>: = IR<16..12>; second operand, conditional test, or shift count register

The bits 16 through 12, are the rc field. This field may hold the second operand, conditional test, or a shift count.

c1<21..0>: = IR<21..0>; long displacement field

In some instructions, the bits 21 through 0 may be used as long displacement field. Notice that there is an overlap of fields. The fields are distinguished in a particular instruction depending on the operation.

c2<16..0>: = IR<16..0>; short displacement or immediate field

The bits 16 through 0 may be used as short displacement or to specify an immediate operand.

c3<11..0>: = IR<11..0>; count or modifier field

The bits 11 through 0 of the SRC instruction may be used for count or modifier field.

Describing the processor state using RTL

The Register Transfer Language can be used to describe the processor state. The following registers and bits together form the processor state set.

PC<31..0>; program counter (it holds the memory address of next instruction to be executed)

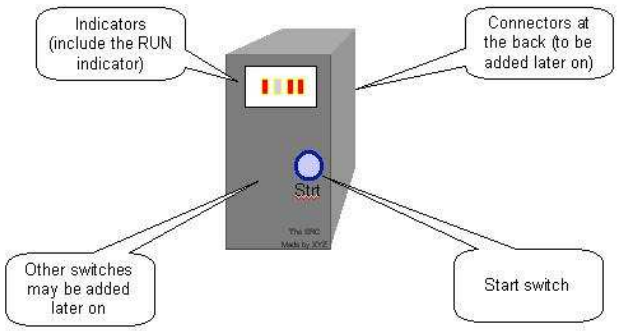
IR<31..0>; instruction register, used to hold the current instruction

Run; one bit run/halt indicator

Strt; start signal

R [0..31]<31..0>; 32, 32 bit general purpose registers

SRC in a Black Box



Difference between our notation and notation used by the text (H&J)

Our Symbols	Meaning	Symbols in text
:	Conditional transfer	➔
;	Sequential statements	;
,	Concurrent statements	:
:=	Naming operator	:=
←	Assignment	←
&	Logical AND	^
~	Logical OR	v
!	Logical NOT	¬
©	Concatenation	#
α	Replication	@
%	Remainder after division (modulo)	none

Our Symbol or terminology	Meaning	Symbol used by H&J
RTL	Register Transfer Language	RTN
Behavioral RTL		Abstract RTN
Structural RTL		Concrete RTN
implementation		Micro architecture
MAR	Memory Address Register	MA
MBR	Memory Buffer Register	MD

Difference between "," and ";" in RTL

Statements separated by a "," take place during the same clock pulse. In other words, the order of execution of statements separated by "**,**" does not matter.

On the other hand, statements separated by a "**;**" take place on successive clock pulses. In other words, if statements are separated by "**;**" the one on the left must complete before the one on the right starts. However, some things written with one RTL statement can take several clocks to complete.

So in the instruction interpretation, fetch-execute cycle, we can see that the first statement. ! Run & Strt : Run ← 1, executes first. After this statement has executed and set run to 1, the statements IR ← M [PC] and PC ← PC + 4 are executed concurrently.

Note that in statements separated by ",", all right hand sides of Register Transfers are evaluated before any left hand side is modified (generally though assignment).

Using RTL to describe the dynamic properties of the SRC The RTL can be used to describe the dynamic properties.

Conditional expressions can be specified through the use of RTL. The following example will illustrate this

(op=14) : R [ra] ← R [rb] - R[rc];

The ← operator is the RTL assignment operator. ';' is the termination operator. This conditional expression implies that "IF the op field is equal to 14, THEN calculate the difference of the value in the register specified by the rb field and the value in the register specified by the rc field, and store the result in the register specified by the ra field."

Effective address calculations in RTL (performed at runtime)

In some instructions, the address of an operand or the destination register may not be specified directly. Instead, the effective address may have to be calculated at runtime.

These effective address calculations can be represented in RTL, as illustrated through the examples below.

Displacement address

disp<31..0> := ((rb=0) : c2<16..0> {sign extend},

(rb≠0) : R [rb] + c2<16..0> {sign extend}),

The displacement (or the direct) address is being calculated in this example. The "," operator separates statements in a single instruction, and indicates that these statements are to be executed

simultaneously. However, since in this example these are two disjoint conditions, therefore, only one action will be performed at one time.

Note that register R0 cannot be added to displacement. rb = 0 just implies we do not need to use the R [rb] field.

Relative address

rel<31..0> := PC<31..0> + c1<21..0> {sign extend},

In the above example, a relative address is being calculated by adding the displacement after sign extension to the contents of the program counter register (that holds the next instruction to be executed in a program execution sequence).

Range of memory addresses

The range of memory addresses that can be accessed using the displacement (or the direct) addressing and the relative addressing is given.

- Direct addressing (displacement with rb=0)
  - If c2<16>=0 (positive displacement) absolute addresses range from 00000000h to 0000FFFFh
  - If c2<16>=1 (negative displacement) absolute addresses range from FFFF0000h to FFFFFFFFh
- Relative addressing
  - The largest positive value of C1<21..0> is 2^{21}-1 and its most negative value is -2^{21}, so addresses up to 2^{21}-1 forward and 2^{21} backward from the current PC value can be specified

Instruction Interpretation

(Describing the Fetch operation using RTL)

The action performed for all the instructions before they are decoded is called 'instruction interpretation'. Here, an example is that of starting the machine. If the machine is not already running (¬Run, or 'not' running), AND (&) it the condition start (Strt) becomes true, then Run bit (of the processor state) is set to 1 (i.e. true).

instruction_Fetch := (

! Run & Strt: Run ← 1 ; instruction_Fetch

Run : (IR ← M [PC], PC ← PC + 4; instruction_Execution));

The := is the naming operator. The ; operator is used to add comments in RTL. The , operator, specifies that the statements are to be executed simultaneously, (i.e. in a single clock pulse). The ; operator is used to separate sequential statements. ← is an assignment operator. & is a logical AND, ~ is a logical OR, and ! is the logical NOT. In the instruction interpretation phase of the fetch-execute cycle, if the machine is running (Run is true), the instruction register is loaded with the instruction at the location M [PC] (the program counter specifies the address of the memory at which the instruction to be executed is located). Simultaneously, the program counter is incremented by 4, so as to point to the next instruction, as shown in the example above. This completes the instruction interpretation.

Instruction Execution

(Describing the Execute operation using RTL)

Once the instruction is fetched and the PC is incremented, execution of the instruction starts. In the following, we denote instruction Fetch by "iF" and instruction execution by "iE".

iE:= (

(op<4..0>= 1) : R [ra] ← M [disp],

(op<4..0>= 2) : R [ra] ← M [rel],

. . .

. . .

(op<4..0>=31) : Run ← 0,); iF);

As shown above, Instruction Execution can be described by using a long list of conditional operations, which are inherently "disjoint".

One of these statements is executed, depending on the condition met, and then the instruction fetch statement (iF) is invoked again at the end of the list of concurrent statements. Thus, instruction fetch (iF) and instruction execution statements invoke each other in a loop. This is the fetch-execute cycle of the SRC.

Concurrent Statements

The long list of concurrent, disjoint instructions of the instruction execution (iE) is basically the complete instruction set of the processor. A brief overview of these instructions is given below.

Load-Store Instructions

(op<4..0>= 1) : R [ra] ← M [disp], load register (ld)

This instruction is to load a register using a displacement address specified by the instruction, i.e. the contents of the memory at the address 'disp' are placed in the register R [ra].

(op<4..0>= 2) : R [ra] ← M [rel], load register relative (ldr)

If the operation field 'op' of the instruction decoded is 2, the instruction that is executed is loading a register (target address of this register is specified by the field ra) with memory contents at a relative address, 'rel'. The relative address calculation has been explained in this section earlier.

(op<4..0>= 3) : M [disp] ← R [ra], store register (st)

If the op-code is 3, the contents of the register specified by address ra, are stored back to the memory, at a displacement location 'disp'.

(op<4..0>= 4) : M[rel] ← R[ra], store register relative (str)

If the op-code is 4, the contents of the register specified by the target register address ra, are stored back to the memory, at a relative address location 'rel'.

(op<4..0>= 5) : R [ra] ← disp, load displacement address (la)

For op-code 5, the displacement address disp is loaded to the register R (specified by the target register address ra).

(op<4..0>= 6) : R [ra] ← rel, load relative address (lar)

For op-code 6, the relative address rel is loaded to the register R (specified by the target register address ra).

Branch Instructions

(op<4..0>= 8) : (cond : PC ← R [rb]), conditional branch (br)

If the op-code is 8, a conditional branch is taken, that is, the program counter is set to the target instruction address specified by rb, if the condition 'cond' is true.

(op<4..0>= 9) : (R [ra] ← PC,
cond : (PC ← R [rb])), branch and link (brl)

If the op field is 9, branch and link instruction is executed, i.e. the contents of the program counter are stored in a register specified by ra field, (so control can be returned to it later), and then the conditional branch is taken to a branch target address specified by rb. The branch and link instruction is useful for returning control to the calling program after a procedure call returns.

The conditions that these 'conditional' branches depend on are specified by the field c3 that has 3 bits. This simply means that when c3<2..0> is equal to one of these six values. We substitute the expression on the right hand side of the : in place of cond These conditions are explained here briefly.

cond := (

c3<2..0>=0 : 0, never

If the c3 field is 0, the branch is never taken.

c3<2..0>=1 : 1, always

If the field is 1, branch is taken

c3<2..0>=2 : R [rc]=0, if register is zero
If c3 = 2, a branch is taken if the register rc = 0.
c3<2..0>=3 : R [rc] ≠ 0, if register is nonzero
If c3 = 3, a branch is taken if the register rc is not equal to 0.
c3<2..0>=4 : R [rc]<31>=0 if positive or zero
If c3 is 4, a branch is taken if the register value in the register specified by rc is greater than or equal to 0.
c3<2..0>=5 : R [rc]<31>=1), if negative
If c3 = 5, a branch is taken if the value stored in the register specified by rc is negative.

Arithmetic and Logical instructions
(op<4..0>=12) : R [ra] ← R [rb] + R [rc],
If the op-code is 12, the contents of the registers rb and rc are added and the result is stored in the register ra.
(op<4..0>=13) : R [ra] ← R [rb] + c2<16..0> {sign extend},
If the op-code is 13, the content of the register rb is added with the immediate data in the field c2, and the result is stored in the register ra.
(op<4..0>=14) : R [ra] ← R [rb] – R [rc],
If the op-code is 14, the content of the register rc is subtracted from that of rb, and the result is stored in ra.
(op<4..0>=15) : R [ra] ← -R [rc],
If the op-code is 15, the content of the register rc is negated, and the result is stored in ra.
(op<4..0>=20) : R [ra] ← R [rb] & R [rc],
If the op field equals 20, logical AND of the contents of the registers rb and rc is obtained and the result is stored in register ra.
(op<4..0>=21) : R [ra] ← R [rb] & c2<16..0> {sign extend},
If the op field equals 21, logical AND of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.
(op<4..0>=22) : R [ra] ← R [rb] ~ R [rc],
If the op field equals 22, logical OR of the contents of the registers rb and rc is obtained and the result is stored in register ra.
(op<4..0>=23) : R [ra] ← R [rb] ~ c2<16..0> {sign extend},
If the op field equals 23, logical OR of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.
(op<4..0>=24) : R [ra] ← ¬R [rc],
If the op-code equals 24, the content of the logical NOT of the register rc is obtained, and the result is stored in ra.

Shift instructions
(op<4..0>=26): R [ra]<31..0 > ← (n α 0) © R [rb] <31..n>,
If the op-code is 26, the contents of the register rb are shifted right n bits times. The bits that are shifted out of the register are discarded. 0s are added in their place, i.e. n number of 0s is added (or concatenated) with the register contents. The result is copied to the register ra.
(op<4..0>=27) : R [ra]<31..0 > ← (n α R [rb] <31>) © R [rb] <31..n>,
For op-code 27, shift arithmetic operation is carried out. In this operation, the contents of the register rb are shifted right n times, with the most significant bit, bit 31, of the register rb added in their place. The result is copied to the register ra.
(op<4..0>=28) : R [ra]<31..0 > ← R [rb] <31-n..0> © (n α 0),
For op-code 28, the contents of the register rb are shifted left n bits times, similar to the shift right instruction. The result is copied to the register ra.

(op<4..0>=29) : R [ra]<31..0 > ← R [rb] <31-n..0> © R [rb]<31..32-n >,

The instruction corresponding to op-code 29 is the shift circular instruction. The contents of the register rb are shifted left n times, however, the bits that move out of the register in the shift process are not discarded; instead, these are shifted in from the other end (a circular shifting). The result is stored in register ra.

where

n := (

(c3<4..0>=0) : R [rc],
(c3<4..0>!=0) : c3 <4..0>),

Notation:

α means replication

© Means concatenation

Miscellaneous instructions

(op<4..0>= 0) , No operation (nop)

If the op-code is 0, no operation is carried out for that clock period. This instruction is used as a stall in pipelining.

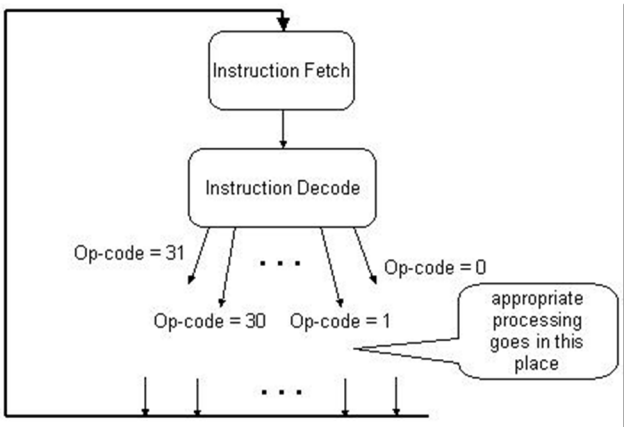
(op<4..0>= 31) : Run ← 0, Halt the processor (Stop)
); iF);

If the op-code is 31, run is set to 0, that is, the processor is halted.

After one of these disjoint instructions is executed, iF, i.e. instruction Fetch is carried out once again, and so the fetch-execute cycle continues.

Flow diagram

Flow diagram is the symbolic representation of Fetch-Execute cycle. Its top block indicates instruction fetch and then next block shows the instruction decode by looking at the first 5-bits of the fetched instruction which would represent op-code which may be from 0 to 31.Depending upon the contents of this op-code the appropriate processing would take place. After the appropriate processing, we would move back to top block, next instruction is fetched and the



same process is repeated until the instruction with op-code 31 would reach and halt the system.

Note:For SRC Assembler and Simulator consult Appendix.

Lecture No. 6

RTL Using Digital Logic Circuits

Reading Material

Handouts Slides

Summary

- Using Behavioral RTL to Describe the SRC (continued)
- Implementing Register Transfer using Digital Logic Circuits

Using behavioral RTL to Describe the SRC (continued)

Once the instruction is fetched and the PC is incremented, execution of the instruction starts. In the following discussion, we denote instruction fetch by "iF" and instruction execution by "iE".

iE:= (
(op<4..0>= 1) : R [ra] ← M [disp],
(op<4..0>= 2) : R [ra] ← M [rel],
. . .
. . .
(op<4..0>=31) : Run ← 0,); iF);

As shown above, instruction execution can be described by using a long list of conditional operations, which are inherently "disjoint". Only one of these statements is executed, depending on the condition met, and then the instruction fetch statement (iF) is invoked again at the end of the list of concurrent statements. Thus, instruction fetch (iF) and instruction execution statements invoke each other in a loop. This is the fetch-execute cycle of the SRC.

Concurrent Statements
The long list of concurrent, disjoint instructions of the instruction execution (iE) is basically the complete instruction set of the processor. A brief overview of these instructions is given below:

Load-Store Instructions
(op<4..0>= 1) : R [ra] ← M [disp], load register (ld)
This instruction is to load a register using a displacement address specified by the instruction, i.e., the contents of the memory at the address 'disp' are placed in the register R [ra].

(op<4..0>= 2) : R [ra] ← M [rel], load register relative (ldr)
If the operation field 'op' of the instruction decoded is 2, the instruction that is executed is loading a register (target address of this register is specified by the field ra) with memory contents at a relative address, 'rel'. The relative address calculation has been explained in this section earlier.
(op<4..0>= 3) : M [disp] ← R [ra], store register (st)

If the op-code is 3, the contents of the register specified by address ra, are stored back to the memory, at a displacement location 'disp'.

(op<4..0>= 4) : M[rel] ← R[ra], store register relative (str)

If the op-code is 4, the contents of the register specified by the target register address ra, are stored back to the memory, at a relative address location 'rel'.

(op<4..0>= 5) : R [ra] ← disp, load displacement address (la)

For op-code 5, the displacement address disp is loaded to the register R (specified by the target register address ra).

(op<4..0>= 6) : R [ra] ← rel, load relative address (lar)

For op-code 6, the relative address rel is loaded to the register R (specified by the target register address ra).

Branch Instructions

(op<4..0>= 8) : (cond : PC ← R [rb]), conditional branch (br)

If the op-code is 8, a conditional branch is taken, that is, the program counter is set to the target instruction address specified by rb, if the condition 'cond' is true.

(op<4..0>= 9) : (R [ra] ← PC,
cond : (PC ← R [rb])), branch and link (brl)

If the op field is 9, branch and link instruction is executed, i.e. the contents of the program counter are stored in a register specified by ra field, (so control can be returned to it later), and then the conditional branch is taken to a branch target address specified by rb. The branch and link instruction is useful for returning control to the calling program after a procedure call returns.

The conditions that these 'conditional' branches depend on, are specified by the field c3 that has 3 bits. This simply means that when c3<2..0> is equal to one of these six values, we substitute the expression on the right hand side of the : in place of cond. These conditions are explained here briefly.

cond := (

c3<2..0>=0 : 0, never

If the c3 field is 0, the branch is never taken.

c3<2..0>=1 : 1, always

If the field is 1, branch is taken

c3<2..0>=2 : R [rc]=0, if register is zero

If c3 = 2, a branch is taken if the register rc = 0.

c3<2..0>=3 : R [rc] ≠ 0, if register is nonzero

If c3 = 3, a branch is taken if the register rc is not equal to 0.

c3<2..0>=4 : R [rc]<31>=0 if positive or zero

If c3 is 4, a branch is taken if the register value in the register specified by rc is greater than or equal to 0.

c3<2..0>=5 : R [rc]<31>=1), if negative

If c3 = 5, a branch is taken if the value stored in the register specified by rc is negative.

Arithmetic and Logical instructions

(op<4..0>=12) : R [ra] ← R [rb] + R [rc],

If the op-code is 12, the contents of the registers rb and rc are added and the result is stored in the register ra.

(op<4..0>=13) : R [ra] ← R [rb] + c2<16..0> {sign extended},

If the op-code is 13, the content of the register rb is added with the immediate data in the field c2, and the result is stored in the register ra.

(op<4..0>=14) : R [ra] ← R [rb] – R [rc],

If the op-code is 14, the content of the register rc is subtracted from that of rb, and the result is stored in ra.

(op<4..0>=15) : R [ra] ← -R [rc],

If the op-code is 15, the content of the register rc is negated, and the result is stored in ra.

(op<4..0>=20) : R [ra] ← R [rb] & R [rc],

If the op field equals 20, logical AND of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=21) : R [ra] ← R [rb] & c2<16..0> {sign extended},

If the op field equals 21, logical AND of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=22) : R [ra] ← R [rb] ~ R [rc],

If the op field equals 22, logical OR of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=23) : R [ra] ← R [rb] ~ c2<16..0> {sign extended},

If the op field equals 23, logical OR of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=24) : R [ra] ← !R [rc],

If the op-code equals 24, the content of the logical NOT of the register rc is obtained, and the result is stored in ra.

Shift instructions

(op<4..0>=26): R [ra]<31..0 > ← (n α 0) © R [rb] <31..n>,

If the op-code is 26, the contents of the register rb are shifted right n bits times. The bits that are shifted out of the register are discarded. 0s are added in their place, i.e. n number of 0s is added (or concatenated) with the register contents. The result is copied to the register ra.

(op<4..0>=27) : R [ra]<31..0 > ← (n α R [rb] <31>) © R [rb] <31..n>,

For op-code 27, shift arithmetic operation is carried out. In this operation, the contents of the register rb are shifted right n times, with the most significant bit, i.e., bit 31, of the register rb added in their place. The result is copied to the register ra.

(op<4..0>=28) : R [ra]<31..0 > ← R [rb] <31-n..0> © (n α 0),

For op-code 28, the contents of the register rb are shifted left n bits times, similar to the shift right instruction. The result is copied to the register ra.

(op<4..0>=29) : R [ra]<31..0 > ← R [rb] <31-n..0> © R [rb]<31..32-n >,

The instruction corresponding to op-code 29 is the shift circular instruction. The contents of the register rb are shifted left n times, however, the bits that move out of the register in the shift process are not discarded; instead, these are shifted in from the other end (a circular shifting). The result is stored in register ra.

where

n := (

(c3<4..0>=0) : R [rc],

(c3<4..0>!=0) : c3 <4..0>),

Notation:

α means replication

© means concatenation

Miscellaneous instructions

(op<4..0>= 0) , No operation (nop)

If the op-code is 0, no operation is carried out for that clock period. This instruction is used as a stall in pipelining.

(op<4..0>= 31) : Run ← 0, Halt the processor (Stop)

); iF);

If the op-code is 31, run is set to 0, that is, the processor stops execution.

After one of these disjoint instructions is executed, iF, i.e. instruction Fetch is carried out once again, and so the fetch-execute cycle continues.

Implementing Register Transfers using Digital Logic Circuits

We have studied the register transfers in the previous sections, and how they help in implementing assembly language. In this section we will review how the basic digital logic circuits are used to implement instructions register transfers. The topics we will cover in this section include:

1. A brief (and necessary) review of logic circuits
2. Implementing simple register transfers
3. Register file implementation using a bus
4. Implementing register transfers with mathematical operations
5. The Barrel Shifter
6. Implementing shift operations

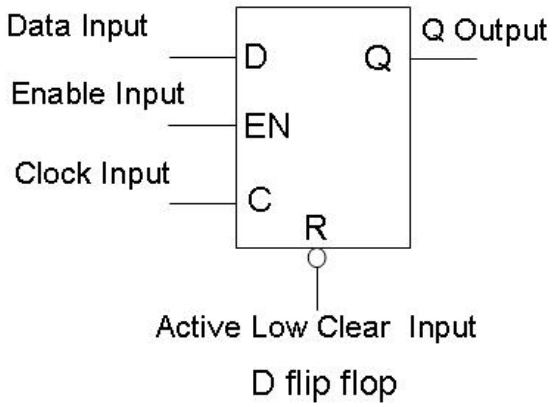
Review of logic circuits

Before we study the implementation of register transfers using logic circuits, a brief overview of some of the important logic circuits will prove helpful. The topics we review in this section include

1. The basic D flip flop
2. The n-bit register
3. The n-to-1 multiplexer
4. Tri-state buffers

The basic D flip flop

A flip-flop is a bi-stable device, capable of storing one bit of Information. Therefore, flip-flops are used as the building blocks of a computer's memory as well as CPU registers.



D flip flop

There are various types of flip-flops; most common type, the D flip-flop is shown in the figure given. The given truth table for this positive-edge triggered D flip-flop shows that the flip-flop is set (i.e. stores a 1) when the data input is high on the leading (also called the positive) edge of the

clock; it is reset (i.e., the flip-flop stores a 0) when the data input is 0 on the leading edge of the clock. The clear input will reset the flip-flop on a low input.

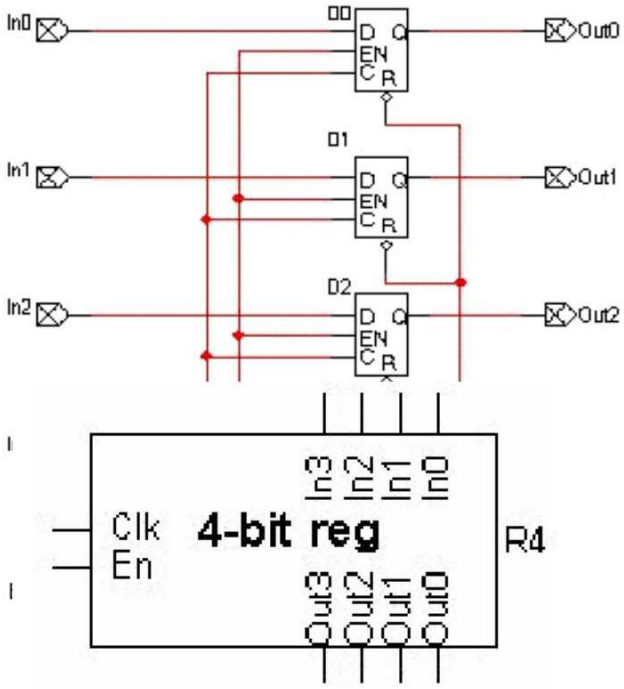
EN	D	Q
0	X	X
1	0	0
1	1	1

Truth table: D Flip Flop

The n-bit register

An n-bit register can be formed by grouping n flip-flops together. So a register is a device in which a group of flip-flops operate synchronously.

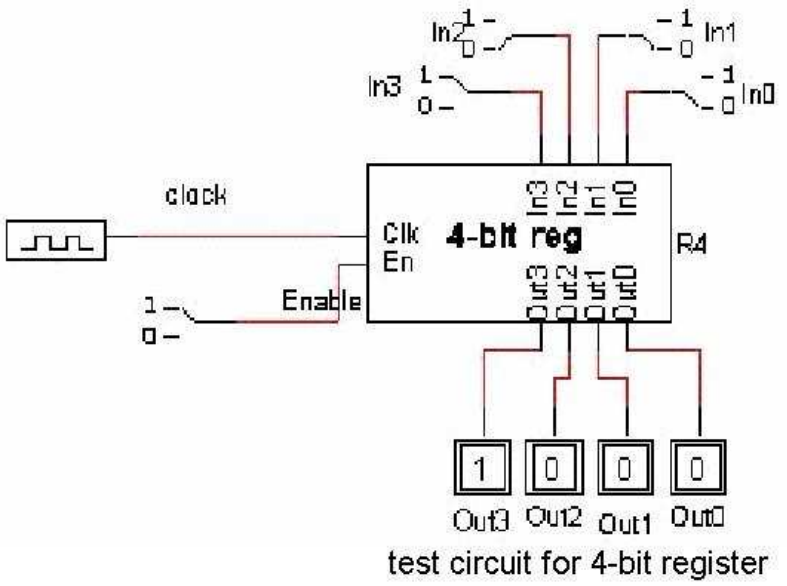
A register is useful for storing binary data, as each flip-flop can store one bit. The clock input of the flip-flops is grouped together, as is the enable input. As shown in the figure, using the input lines a binary number can be stored in the register by applying the corresponding logic level to each of the flip-flops simultaneously at the positive edge of the clock.



4-bit Register Symbol

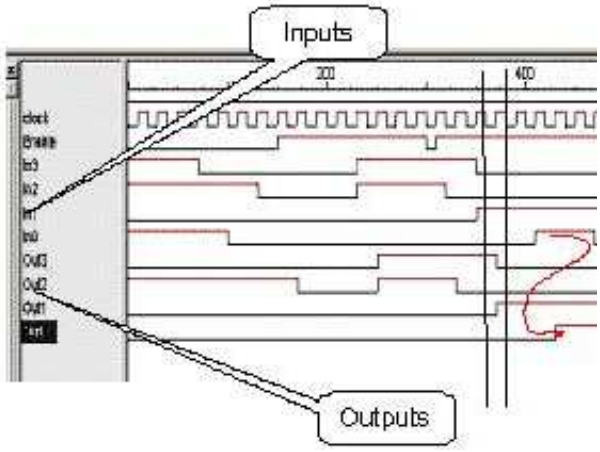
The next figure shows the symbol of a 4-bit register used for an integrated circuit. In0 through In3 are the four input lines, Out0 through Out3 are the four output lines, Clk is the clock input, and En is the enable line. To get a better understanding of this register, consider the situation where we want to store the binary number 1000 in the register. We will apply the number to the input lines, as shown in the figure given.

On the leading edge of the clock, the number will be stored in the register. The enable input has to be high if the number is to be stored into the register.



test circuit for 4-bit register

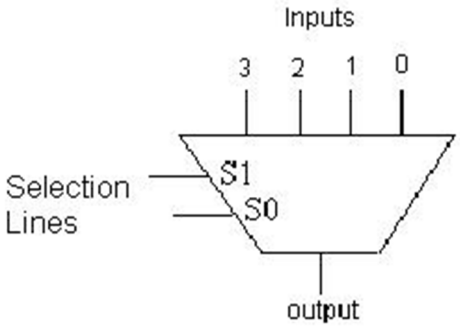
Waveform/Timing diagram



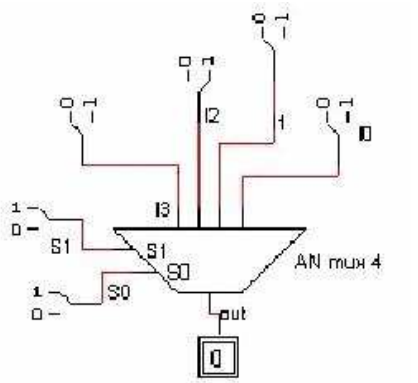
The n-to-1 multiplexer

A multiplexer is a device, constructed through combinational logic, which takes n inputs and transfers one of them as the output at a time. The input that is selected as the output depends on the selection lines, also called the control input lines. For an n-to-1 multiplexer, there are n input lines, $\log_2 n$ control lines, and 1 output line. The given figure shows a 4-to-1 multiplexer. There are 4 input lines; we number these lines as line 0 through line 3. Subsequently, there are 2 select lines (as $\log_2 4 = 2$).

For a better understanding, let us consider a case where we want to transfer the input of line 3 to the output of the multiplexer. We will need to apply the binary number 11 on the select lines (as the binary number 11 represents the decimal number 3). By doing so, the output of the multiplexer will be the input on line 3, as shown in the test circuit given.

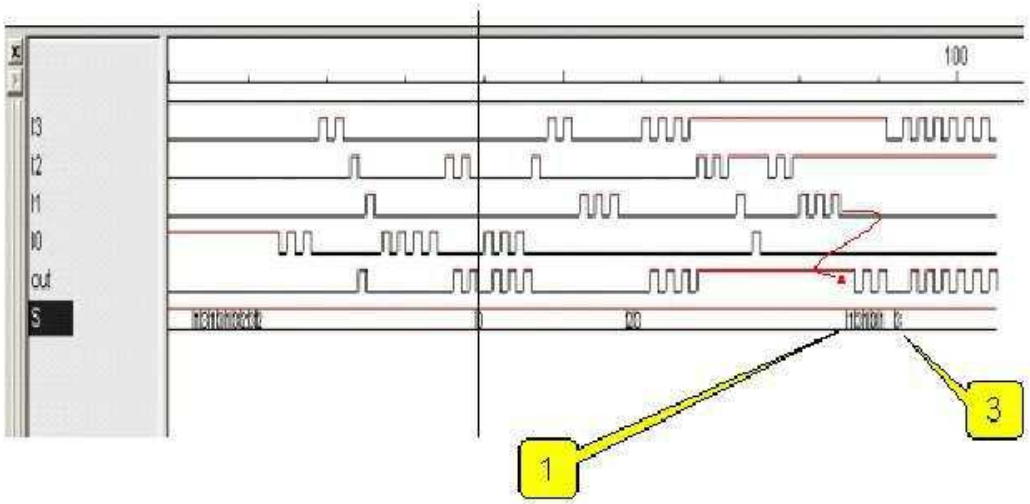


A 4-to-1 MUX



test circuit for 4-to-1 MUX

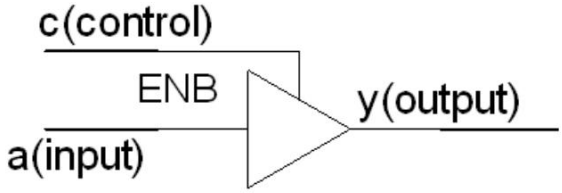
Timing waveform



Timing Waveform for MUX

Tri-state buffers

The tri-state buffer, also called the three-state buffer, is another important component in the digital logic domain. It has a single input, a single output, and an enable line. The input is concatenated to the output only if it is enabled through the enable line, otherwise it gives a high impedance output, i.e. it is tri-stated, or electrically disconnected from the input These buffers are available both



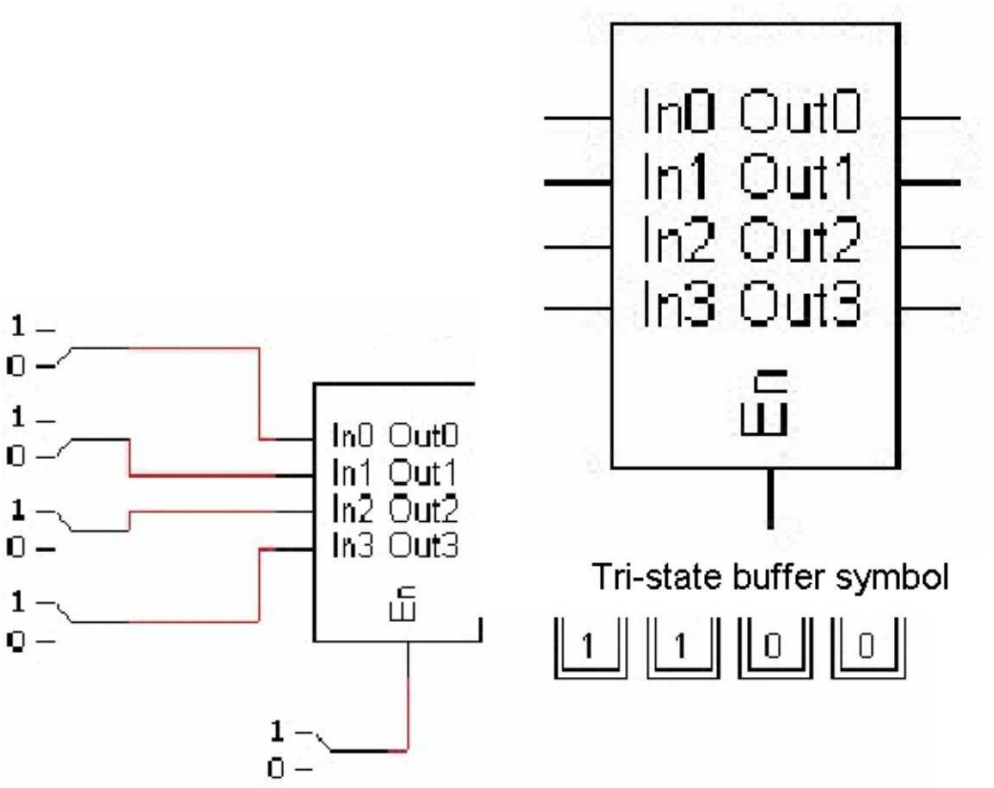
in the inverting and the non-inverting form. The inverting tri-state buffers output the 'inverted' input when they are enabled, as opposed to their non-inverting counterparts that simply output the input when enabled. The circuit symbol of the tri-state buffers is shown.

c	a	y
0	0	Z
0	1	Z
1	0	0
1	1	1

Truth table :Tri-state buffer

The truth table further clarifies the working of a non-inverting tri-state buffer.

We can see that when the enable input (or the control input) c is low (0), the output is high impedance Z. The symbol of a 4-bit tri-state buffer unit is shown in the figure. There are four input lines, an equal number of output lines, and an enable line in this unit. If we apply a high on the input 3 and 2, and a low on input 1 and 0, we get the output 1100, only when the enable input is high, as shown in the given figure.



Test circuit for Tri-state buffer

Implementing simple register transfers

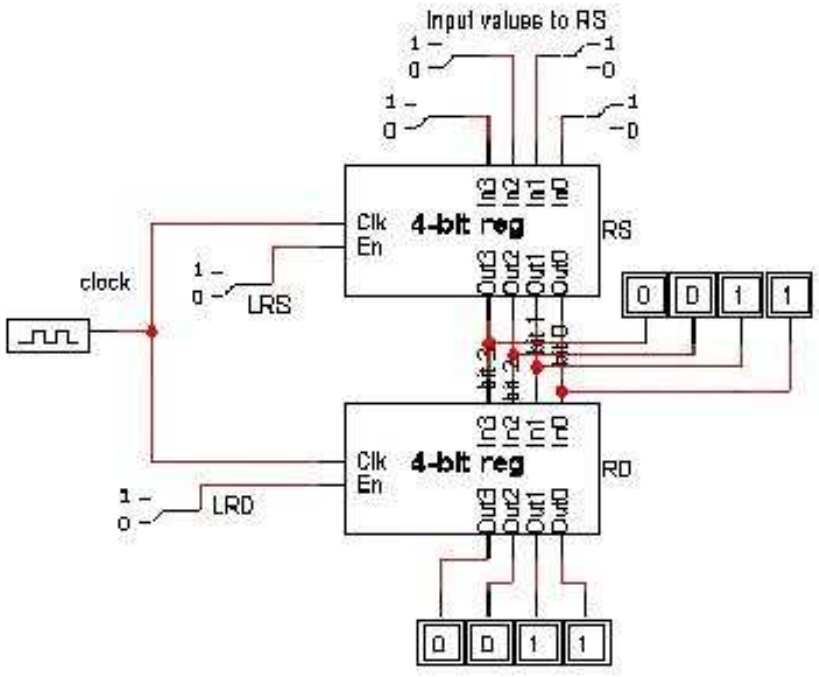
We now build on our knowledge of the primitive logic circuits to understand how register transfers are implemented. In this section we will study the implementation of the following

- Simple conditional transfer
- Concept of control signals
- Two-way transfers
- Connecting multiple registers
- Buses
- Bus implementations

Simple conditional transfer

In a simple conditional transfer, a condition is checked, and if it is true, the register transfer takes place. Formally, a conditional transfer is represented as

Cond: RD ← RS



Conditional Trasnfer

This means that if the condition 'Cond' is true, the contents of the register named RS (the source register) are copied to the register RD (the destination register). The following figure shows how the registers may be interconnected to achieve a conditional transfer. In this circuit, the output of the source register RS is connected to the input of the destination registers RD. However, notice that the transfer will not take place unless the enable input of the destination register is activated. We may say that the 'transfer' is being controlled by the enable line (or the control signal). Now, we are able to control the transfer by selectively enabling the control signal, through the use of other combinational logic that may be the equivalent of our condition.
The condition is, in general, a Boolean expression, and in this example, the condition is equivalent to LRD =1.

Two-way transfers

In the above example, only one-way transfer was possible, i.e., we could only copy the contents of RS to RD if the condition was met. In order to be able to achieve two-way transfers, we must also provide a path from the output of the register RD to input of register RS. This will enable us to implement
Cond1: RD ← RS
Cond2: RS ← RD

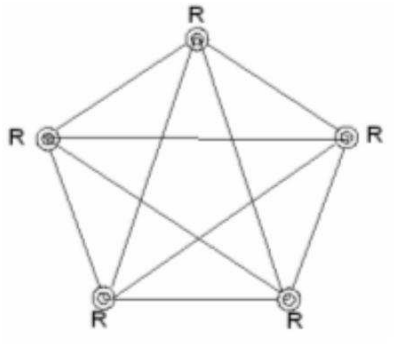
Connecting multiple registers

We have seen how two registers can be connected. However, in a computer we need to connect more than just two registers. In order to connect these registers, one may argue that a connection between the input and output of each be provided. This solution is shown for a scenario where there are 5 registers that need to be interconnected.

We can see that in this solution, an m-bit register requires two connections of m-wires each. Hence five m-bit registers in a "point-to-point" scheme require 20 connections; each with m wires. In general, n registers in a point to point scheme require n (n-1) connections. It is quite obvious that this solution is not going to scale well for a large number of registers, as is the case in real machines. The solution to this problem is the use of a bus architecture, which is explained in the following sections.

Buses

A bus is a device that provides a shared data path to a number of devices that are connected to it, via a 'set of wires' or a 'set of conductors'. The modern computer systems extensively employ the bus architecture. Control signals are needed to decide which two entities communicate using the shared medium, i.e. the bus, at any given time. This control signals can be open collector gate based, tri-state buffer based, or they can be implemented using multiplexers.

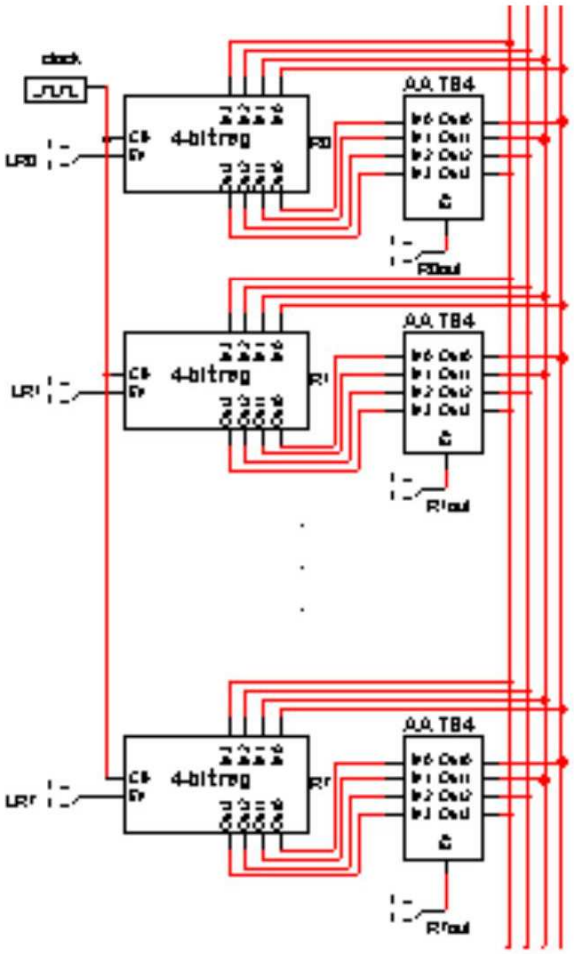


Multiple register connections

Register file implementation using the bus architecture

A number of registers can be inter-connected to form a register file, through the use of a bus. The given diagram shows eight 4-bit registers (R0, R1, …, R7) interconnected through a 4-bit bus using 4-bit tri-state buffer units (labeled AA_TS4). The contents of a particular register can be transferred onto the bus by applying a logical high input on the enable of the corresponding tri-state buffer. For instance, R1out can be used to enable the tri-state buffers of the register R1, and in turn transfer the contents of the register on the bus.

Once the contents of a particular register are on the bus, the contents may be transferred, or read into any other register. More than one register may be written in this manner; however, only one register can write its value on the bus at a given time.

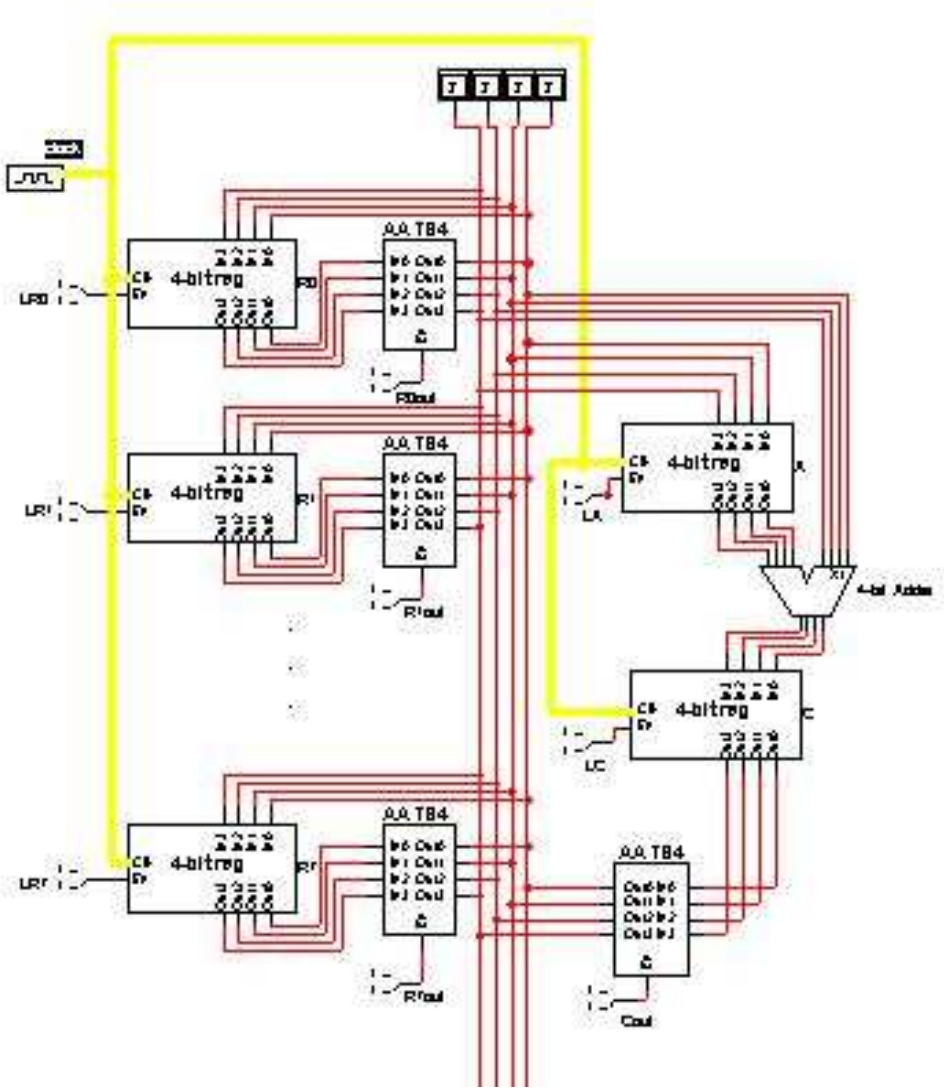


Register File

Implementing register transfers with mathematical operations

We have studied the implementation of simple register transfers; however, we frequently encounter register transfers with mathematical operations. An example is (opc=1): R4← R3 + R2;

These mathematical operations may be achieved by introducing appropriate combinational logic; the above operation can be implemented in hardware by including a 4-bit adder with the register files connected through the bus. There are two more registers in this configuration, one for holding one of the operands, and the other for holding the result before it is transferred to the destination register. This is shown in the figure below.



We now take a look at the steps taken for the (conditional, mathematical) transfer (opc=1): R4← R3 + R2. First of all, if the condition opc = 1 is met, the contents of the first operand register, R3, are transferred to the temporary registerA through the bus. This is done by activating R3out.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	A ← R3	LA, R3out
2	C ← A + R2	LC, R2out
3	R4 ← C	LR4, Cout

Structural RTL: add operation

It lets the contents of the register R3 to be loaded on the bus. At the same time, applying a logical high input to LA enables the load for the register A. This lets the binary number on the bus (the contents of register R3) to be loaded into the register A. The next step is to enable R2out to load the contents of the register R2 onto the bus. As can be observed from the figure, the output of the register A is one of the inputs to the 4-bit adder; the other input to the adder is the bus itself. Therefore, as the contents of register R2 are loaded onto the bus, both the operands are available

to the adder. The output can then be stored to the register RC by enabling its write. So a high input is applied to LC to store the result in register RC.

The third and final step is to store (transfer) the resultant number in the destination register R4. This is done by enabling Cout, which writes the number onto the bus, and then enabling the read of the register R4 by activating the control signal to LR4. These steps are summarized in the given table.

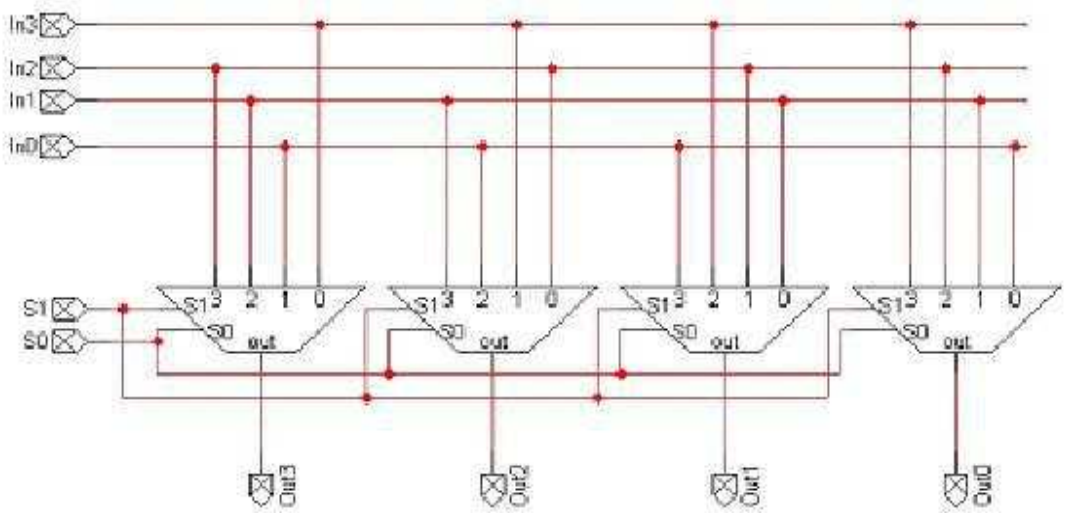
The barrel shifter

Shift operations are frequently used operations, as shifts can be used for the implementation of multiplication and division etc. A bi-directional shift register with a parallel load capability can be used to perform shift operations. However, the delays in such structures are dependent on the number of shifts that are to be performed, e.g., a 9 bit shift requires nine clock periods, as one shift is performed per clock cycle. This is not an optimal solution. The barrel shifter is an alternative, with any number of shifts accomplished during a single clock period. Barrel shifters are constructed by using multiplexers. An n-bit barrel shifter is a combinational circuit implemented using n multiplexers. The barrel provides a shifted copy of the input data at its output. Control inputs are provided to specify the number of times the input data is to be shifted. The shift process can be a simple one with 0s used as fillers, or it can be a rotation of the input data. The corresponding figure shows a barrel shifter that shifts right the input data; the number of shifts depends on the bit pattern applied on the control inputs S0, S1.

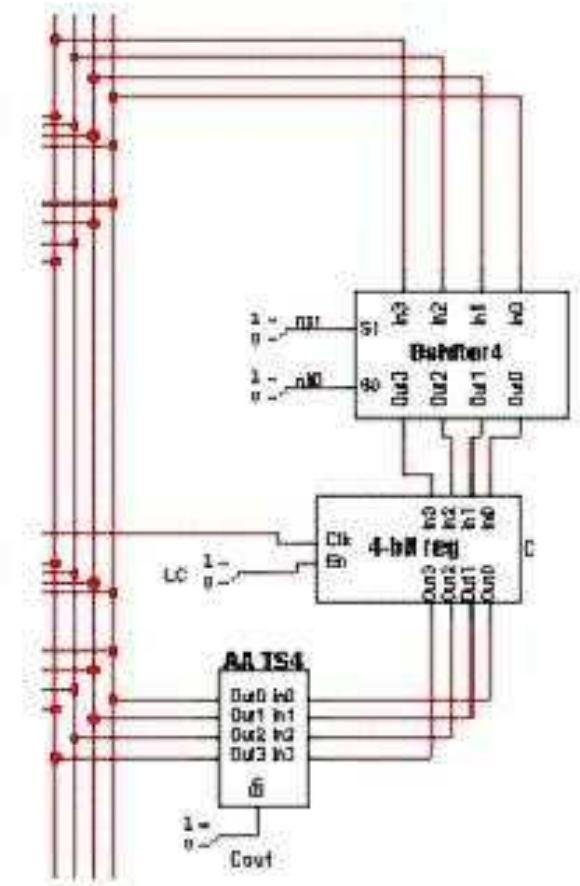
The function table for the barrel shifter is given. We see from the table that in order to apply single shift to the input number, the control signal is 01 on (S1, S0), which is the binary equivalent of the decimal number 1. Similarly, to apply 2 shifts, control signal 10 (on S1, S0) is applied; 10 is the binary equivalent of the decimal number 2. A control input of 11 shifts the number 3 places to the right.

Now we take a look at an example of the shift operation being implemented through the use of the barrel shifter: R4← ror R3 (2 times);

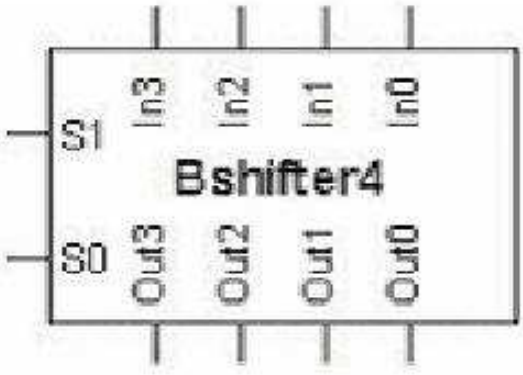
The shift functionality can be incorporated into the register file circuit with the bus architecture we have been building, by introducing the barrel shifter, as shown in the given figure.



Barrel Shifter



Shift operation using Barrel Shifter



Barrel Shifter Symbol

S1	S0	Output in terms of the inputs
0	0	In3 In2 In1 In0
0	1	In0 In3 In2 In1
1	0	In1 In0 In3 In2
1	1	In2 In1 In0 In3

Function table: Barrel shifter

To perform the operation, R4← ror R3 (2 times),

The first step is to activate R3out, nb1 and LC. Activating R3out will load the contents of the register R3 onto the bus. Since the bus is directly connected to the input of the barrel shifter, this number is applied to the input side. nb1 and nb0 are the barrel shifter's control lines for specifying the number of shifts to be applied. Applying a high input to nb1 and a low input to nb0 will shift the number two places to the right. Activating LC will load the shifted output of the barrel shifter into the register C. The second step is to transfer the contents of the register C to the register R4. This is done by activating the control Cout, which will load the contents of register C onto the data bus, and by activating the control LR4, which will let the contents of the bus be written to the register R4. This will complete the conditional shift-and-store operation. These steps are summarized in the table shown below.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	C ← R3 (after rotating right twice)	R3out, nb1, LC
2	R4 ← C	LR4, Cout

Structural RTL: Shift operation

Lecture No. 7

Design Process for ISA of FALCON-A

Reading Material

Handouts Slides

Summary

- Outline of the thinking process for ISA Design
- Introduction to the ISA of FALCON-A

Instruction Set Architecture (ISA) Design: Outline of the thinking process

In this module we will learn to appreciate, understand and apply the approach adopted in designing an instruction set architecture. We do this by designing an ISA for a new processor. We have named our processor FALCON-A, which is an acronym for First Architecture for Learning Computer Organization and Networks (version A). The term Organization is intended to include Architecture and Design in this acronym.

Elements of the ISA

Before we go onto designing the instruction set architecture for our processor FALCON-A, we need to take a closer look at the defining components of an ISA. The following three key components define any instruction set architecture.

1. The operations the processor can execute
2. Data access mode for use as operands in the operations defined
3. Representation of the operations in memory

We take a look at all three of the components in more detail, and wherever appropriate, apply these steps to the design of our sample processor, the FALCON-A. This will help us better understand the approach to be adopted for the ISA design of a processor. A more detailed introduction to the FALCON-A will be presented later.

The operations the processor can execute

All processors need to support at least three categories (or functional groups) of instructions
– Arithmetic, Logic, Shift
– Data Transfer
– Control

ISA Design Steps – Step 1

We need to think of all the instructions of each type that ought to be supported by our processor, the FALCON-A. The following are the instructions that we will include in the ISA for our processor.

Arithmetic:
add, addi (and with an immediate operand), subtract, subtract-immediate, multiply, divide
Logic:
and, and-immediate, or, or-immediate, not
Shift:
shift left, shift right, arithmetic shift right
Data Transfer:
Data transfer between registers, moving constants to registers, load operands from memory to registers, store from registers to memory and the movement of data between registers and input/output devices

Control:
Jump instructions with various conditions, call and return from subroutines, instructions for handling interrupts
Miscellaneous instructions:
Instructions to clear all registers, the capability to stop the processor, ability to "do nothing", etc.

ISA Design Steps – Step 2

Once we have decided on the instructions that we want to add support for in our processor, the second step of the ISA design process is to select suitable mnemonics for these instructions. The following mnemonics have been selected to represent these operations.
Arithmetic:
add, addi, sub ,subi ,mul ,div
Logic:
and, andi, or, ori, not
Shift:
shiftl, shiftr, asr
Data Transfer:
load, store, in, out, mov, movi
Control:
jpl, jmi, jnz, jz, jump, call, ret, int.iret
Miscellaneous instructions:
nop, reset, halt

ISA Design Steps – Step 3

The next step of the ISA design is to decide upon the number of bits to be reserved for the op-code part of the instructions. Since we have 32 instructions in the instruction set, 5 bits will suffice (as 2^5 =32) to encode these op-codes.

ISA Design Steps – Step 4

The fourth step is to assign op-codes to these instructions. The assigned op-codes are shown below.
Arithmetic:
add (0), addi (1), sub (2), subi (3), mul (4),div (5)
Logic:
and (8), andi (9), or (10), ori (11), not (14)

Shift:
shiftl (12), shiftr (13), asr (15)
Data Transfer:
load (29), store (28), in (24), out (25), mov (6), movi (7)
Control:
jpl (16), jmi (17), jnz (18), jz (19), jump (20), call (22), ret (23), int (26), iret (27)

Miscellaneous instructions:
nop (21), reset (30), halt (31)

Now we list these instructions with their op-codes in the binary form, as they would appear in the machine instructions of the FALCON-A.

00000	add	01000	and	10000	jpl	11000	in
00001	addi	01001	andi	10001	jmi	11001	out
00010	sub	01010	or	10010	jnz	11010	int
00011	subi	01011	ori	10011	jz	11011	iret
00100	mul	01100	shiftl	10100	jump	11100	store
00101	div	01101	shiftr	10101	nop	11101	load
00110	mov	01110	not	10110	call	11110	reset
00111	movi	01111	asr	10111	ret	11111	halt

Data access mode for operations

As mentioned earlier, the instruction set architecture of a processor defines a number of things besides the instructions implemented; the resources each instruction can access,

the number of registers available to the processor, the number of registers each instruction can access, the instructions that are allowed to access memory, any special registers, constants and any alternatives to the general-purpose registers. With this in mind, we go on to the next steps of our ISA design.

ISA Design Steps – Step 5

We now need to select the number and types of operands for various instructions that we have selected for the FALCON-A ISA.

ALU instructions may have 2 to 3 registers as operands. In case of 2 operands, a constant (an immediate operand) may be included in the instruction.

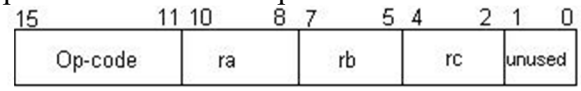
For the load/store type instructions, we require a register to hold the data that is to be loaded from the memory, or stored back to the memory. Another register is required to hold the base address for the memory access. In addition to these two registers, a field is required in the instruction to specify the constant that is the displacement to the base address.

In jump instructions; we require a field for specifying the register that holds the value that is to be compared as the condition for the branch, as well as a destination address, which is specified as a constant.

Registers	Encoding
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Once we have decided on the number and types of operands that will be required in each of the instruction types, we need to address the issue of assigning specific bit-fields in the instruction for each of these operands. The number of bits required to represent each of these operands will eventually determine the instruction word size. In our example processor, the FALCON-A, we reserve eight general-purpose registers. To encode a register in the instructions, 3 bits are required (as 2^3 =8). The registers are encoded in the binary as shown in the given table.

Therefore, the instructions that we will add support for FALCON-A processor will have the given general format. The instructions in the FALCON-A processor are going to be variations of this format, with four different formats in all. The exact format is dependent on the actual number of operands in a particular instruction.



ISA Design Steps – Step 6

The next step towards completely defining the instruction set architecture of our processor is the design of memory and its organization. The number of the memory cells that we may have in the organization depends on the size of the Program Counter register (PC), and the size of the address bus. This is because the size of the program counter and the size of the address bus put a limitation on the number of memory cells that can be referred to for loading an instruction for execution. Additionally, the size of the data bus puts a limitation on the size of the memory word that can be referred to in a single clock cycle.

ISA Design Steps – Step 7

Now we need to specify which instructions will be allowed to access the memory. Since the FALCON-A is intended to be a RISC-like machine; only the load/ store instructions will be allowed to access the memory.

Addressing Mode	Format	Example
direct	[constant or label]	[10] or [a]
displacement	[register + constant or label]	[R1 + 8] or [r2 + a]
register indirect	[register]	[R3]

ISA Design Steps – Step 8

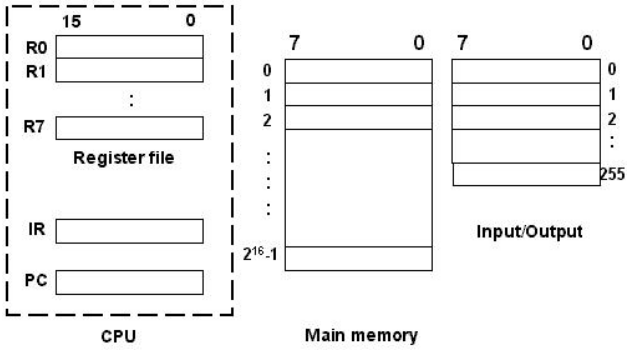
Next we need to select the memory-addressing modes. The given table lists the types of addressing modes that will be supported for the load/store instructions.

FALCON-A: Introduction

FALCON stands for First Architecture for Learning Computer Organization and Networks. It is a 'RISC-like' general-purpose processor that will be used as a teaching aid for this course. Although the FALCON-A is a simple machine, it is powerful enough to explain a variety of fundamental concepts in the field of Computer Architecture.

Programmer's view of the FALCON-A

FALCON-A, an example of a GPR (General Purpose Register) computer, is the first version of the FALCON processor. The programmer's view of the FALCON-A is given in the figure shown. As it is clear from the figure, the CPU contains a register file of 8 registers, named R0 through R7. Each of these registers is 16 bits in length. Aside from these registers, there are two special-purpose registers, the Program Counter (PC), and the Instruction Register (IR). The main memory is organized as 2^{16} x 8 bits, i.e. 2^{16} cells of 1 byte each. The memory word size is 2 bytes (or 16 bits).



The input/output space is 256 bytes (8 bit I/O ports). The storage in these registers and memory is in the big-endian format.

Lecture No. 8

ISA of the FALCON-A

Reading Material
Handouts Slides

Summary

- **Introduction to the ISA of the FALCON-A**
- **Examples for the FALCON-A**

Introduction to the ISA of the FALCON-A

We take a look at the notation that we are going to employ when studying the FALCON-A. We will refer to the contents of a register by enclosing in square brackets the name of the register, for instance, R [3] refers to the contents of the register 3. Memory contents are to be referred to in a similar fashion; for instance, M [8] refers to the contents of memory at location 8 (or the 8th memory cell).

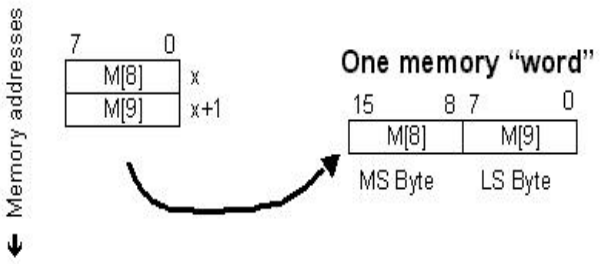


Fig. Big- Endian Notation

Since memory is organized into cells of 1 byte, whereas the memory word size is 2 bytes, two adjacent memory cells together make up a memory word. So, memory word at the memory address 8 would be defined as 1 byte at address 8 and 1 byte at address 9. To refer to 16-bit memory words, we make use of a special notation, the concatenation of two memory locations. Therefore, to refer to the 16-bit memory word at location 8, we would write M[8]©M[9]. As we employ the big-endian format,
M [8]<15…0>:=M[8]©M[9]
So in our notation © is used to represent concatenation.

Little endian puts the smallest numbered byte at the least-significant position in a word, whereas in big endian, we place the largest numbered byte at the most significant position. Note that in our case, we use the big-endian convention of ordering bytes. However, within each byte itself, the ordering of the bits is little endian.

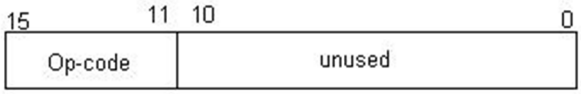
FALCON-A Features

The FALCON-A processor has fixed-length instructions, each 16 bits (2 bytes) long. Addressing modes supported are limited, and memory is accessed through the load/store instructions only.

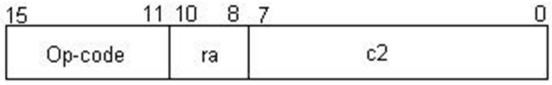
FALCON-A Instruction Formats

Three categories of instructions are going to be supported by the FALCON-A processor; arithmetic, control, and data transfer instructions. Arithmetic instructions enable mathematical computations. Control instructions help change the flow of the program as and when required. Data transfer operations move data between the processor and memory. The arithmetic category also includes the logical instructions. Four different types of instruction formats are used to specify these instructions. A brief overview of the various fields in these instructions formats follows.

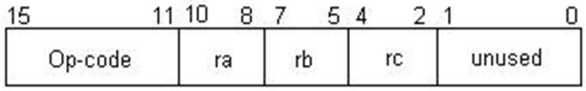
Type I instruction format is shown in the given figure. In it, 5 bits are reserved for the op-code (bits 11 through 15). The rest of the bits are unused in this instruction type, which means they are not considered.



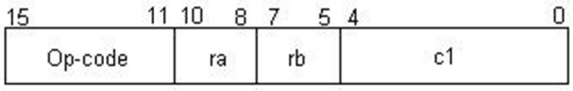
Type II instruction shown in the given figure, has a 5-bit op-code field, a 3-bit register field, and an 8-bit constant (or immediate operand) field.



Type III instructions contain the 5-bit op-code field, two 3-bit register fields for source and destination registers, and an immediate operand field of length 5 bits.



Type IV instructions contain the op-code field, two 3-bit register fields, a constant filed on length 3 bits as well as two unused bits. This format is shown in the given figure.



Encoding of registers

We have a register file comprising of eight general-purpose registers in the CPU. To encode these registers in the binary, so they can be referred to in various instructions, we require $\log_2(8) = 3$ bits. Therefore, we have already allocated three bits per register in the instructions, as seen in the various instruction formats. The encoding of registers in the binary format is shown in the given table.

It is important to note here that the register R0 has special usage in some cases. For instance, in load/ store operations, if register R0 is used as a second operand, its value is considered to be zero. R0 has special usage in the multiply and divide (mul & div) instructions as well.

Registers	Encoding
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Fig. Register Encodings

Instructions and instruction formats

We return to our discussion of instruction formats in this section. We will now classify which instructions belong to what instruction format types.

Type I

Five of the instructions included in the instruction set of FALCON-A belong to type I instruction format. These are

1. nop (op-code = 21)
   This instruction is to instruct the processor to 'do nothing', or, in other words, do 'no operation'. This instruction is generally useful in pipelining. We will study pipelining later in the course.
2. reset (op-code = 30)
3. halt (op-code=31)
4. int (opcode= 26)

5.iret (op-code= 27)

All of these instructions take no operands, therefore, besides the 5 bits used for the op-code, the rest of the bits are unused.

Type II

There are nine FALCON-A instructions that belong to this type. These are listed below.

1. movi (op-code = 7)
   The movi instruction loads a register with the constant (or the immediate value) specified as the second operand. An example is
   movi R3, 56 R[3] ← 56
   This means that the register R3 will have the value 56 stored in it as this instruction is executed.

2. in (op-code = 24)
   This instruction is to load the specified register from input device. An example and its interpretation in register transfer language are
   in R3, 57 R [3] ← IO [57]

3. out (op-code = 25)
   The 'out' instruction will move data from the register to the output device specified in the instruction, as the example demonstrates:
   out R7, 34 IO [34] ← R [7]

4. ret (op-code=23)
   This instruction is to return control from a subroutine. This is done using a register, where the return address is stored. As shown in the example, to return control, the program counter is assigned the contents of the register.
   ret R3 PC ← R [3]

5. jz (op-code= 19)
   When this instruction is executed, the value of the register specified in the field ra is checked, and if it is equal to zero, the Program Counter is advanced by the jump(value) specified in the instruction.
   jz r3, [4] (R[3]=0): PC← PC+ 4;
   In this example, register r3's value is checked, and if found to be zero, PC is advanced by 4.

6. jnz (op-code= 18) This instruction is the reverse of the jz instruction, i.e., the jump (or the branch) is taken, if the contents of the register specified are not equal to zero.jnz r4, [variable] (R[4]≠0): PC← PC+ variable;

7. jpl (op-code= 16) In this instruction, the value contained in the register specified in the field ra is checked, and if it is positive, the jump is taken.
   jpl r3, [label] (R[3]≥0): PC ← PC+ (label-PC);

8. jmi (op-code= 17) In this case, PC is advanced (jump/branch is taken) if the register value is negative
   jmi r7, [address] (R[7]<0): PC← PC+ address;

Note that, in all the instructions for jump, the jump can be specified by a constant, a variable, a label or an address (that holds the value by which the PC is to be advanced). A variable can be defined through the use of the '.equ' directive. An address (of data) can be specified using the directive '.db' or '.dw'. A label can be specified with any instruction. In its usage, we follow the label by a colon ':' before the instruction itself. For example, the following is an instruction that has a label 'alfa' attached to it alfa: movi r3 r4

Labels implement relative jumps, 128 locations backwards or 127 locations forward (relative to the current position of program control, i.e. the value in the program counter). The compiler handles the interpretation of the field c2 as a constant/ variable/ label/ address. The machine code just contains an 8-bit constant that is added to the program counter at run-time.

9. jump (op-code= 20)

   This instruction instructs the processor to advance the program counter by the displacement specified, unconditionally (an unconditional jump). The assembler allows the displacement (or the jump) to be specified in any of the following ways

   jump [ra + constant]
   jump [ra + variable]
   jump [ra + address]
   jump [ra + label]

The types of unconditional jumps that are possible are

- Direct
- Indirect
- PC relative (a 'near' jump)
- Register relative (a 'far' jump)

The c2 field may be a constant, variable, an address or a label.
A direct jump is specified by a PC-label.
An indirect jump is implemented by using the C2 field as a variable.
In all of the above instructions, if the value of the register ra is zero, then the Program Counter is incremented (or decremented) by the sign-extended value of the constant specified in the instruction. This is called the PC-relative jump, or the 'near' jump. It is denoted in RTL as:

(ra = 0): PC← PC + (8αC2<7>) © C2<7..0>;

If the register ra field is non-zero, then the Program Counter is assigned the sum of the sign-extended constant and the value of register specified in the field ra. This is known as the register-relative, or the 'far' jump. In RTL, this is denoted as: (ra≠0):PC← R[ra]+(8αC2<7>)©C2<7..0>;
Note that C2 is computed by sign extending the constant, variable, address, or (label – PC). Since we have 8 bits available for the C2 field (which can be a constant, variable, address or a PC-label), the range for the field is -128 to + 127. Also note that the compiler does not allow an instruction with a negative sign before the register name, such as 'jump [-r2]'. If the C2 field is being used as an address, it should always be an even value for the jump instruction. This is because our instruction word size is 16 bits, whereas in instruction memory, the instruction memory cells are of 8 bits each. Two consecutive cells together make an instruction.

Type III

There are nine instructions of the FALCON-A that belong to Type III. These are:

1. andi (op-code = 9)
   The andi instruction bit-wise 'ands' the constant specified in the instruction with the value stored in the register specified in the second operand register and stores the result in the destination register. An example is:
   andi r4, r3, 5
   This instruction will bit-wise and the constant 5 and R[3], and assign the value thus obtained to the register R[4], as given .
   R [4] ← R [3] & 5
2. addi (op-code = 1)
   This instruction is to add a constant value to a register; the result is stored in a

destination register. An example:
addi r4, r3,4 R [4] ← R [3] + 4

3. subi (op-code = 3)
   The subi instruction will subtract the specified constant from the value stored in a source register, and store to the destination register. An example follows.
   subi r5, r7, 9 R [5] ← R [7] – 9
4. ori (op-code= 11)
   Similar to the andi instruction, the ori instruction bit-wise 'ors' a constant with a value stored in the source register, and assigns it to the destination register. The following instruction is an example.
   ori r4, r7, 3 R[4] ← R[7] ~ 3
5. shiftl (op-code = 12)
   This instruction shifts the value stored in the source register (which is the second operand), and shifts the bits left as many times as is specified by the third operand, the constant value. For instance, in the instruction
   shiftl r4, r3, 7
   The contents of the register are shifted left 7 times, and the resulting number is assigned to the register r4.
6. shiftr (op-code = 13)
   This instruction shifts to the right the value stored in a register. An example is:
   shiftr r4, r3,9
7. asr (op-code = 15)
   An arithmetic shift right is an operation that shifts a signed binary number stored in the source register (which is specified by the second operand), to the right, while leaving the sign-bit unchanged. A single shift has the effect of dividing the number by 2. As the number is shifted as many times as is specified in the instruction through the constant value, the binary number of the source register gets divided by the constant value times 2.
   An example is asr r1, r2, 5

   This instruction, when executed, will divide the value stored in r2 by 10, and assign the result to the register r1.

8. load (op-code= 29)
   This instruction is to load a register from the memory. For instance, the Instruction
   load r1, [r4 +15]
   will add the constant 15 to the value stored in the register r4, access the memory location that corresponds to the number thus resulting, and assign the memory contents of this location to the register r1; this is denoted in RTL by:
   R[1] ← M[R[4]+15]

9. store (op-code= 28)
   This instruction is to store a value in the register to a particular memory location. In the example:
   store r6, [r7+13]
   The contents of the register r6 are being stored to the memory location that corresponds to the sum of the constant 13 and the value stored in the register r7.
   M[R[7]+13] ← R[6]

Type III Modified

There are 3 instructions in the modified form of the Type III instructions. In the modified Type III instructions, the field c1 is unused.

1. mov (op-code = 6)
   This instruction will move (copy) data of a source register to a destination register. For instance, in the following example, the contents of the register r3 are copied to the register r4.
   mov r4, r3

In RTL, this can be represented as

R[4] ← R[3]

2. not (op-code = 14)
   This instruction inverts the contents of the source register, and assigns the value thus obtained to the destination register. In the following example, the contents of register r2 are inverted and assigned to register r4.
   not r4, r2
   In RTL:

R[4] ← !R[2]

3. call (op-code = 22)
   Procedure calls are often encountered in programming languages. To add support for procedure (or subroutine) calls, the instruction call is used. This instruction first stores the return address in a register and then assigns the Program Counter a new value (that specifies the address of the subroutine). Following is an example of the call instruction
   call r4, r3
   This instruction saves the current contents (the return address) of the Program Counter into the register r4 and assigns the new value to the PC from register r3.

R[4] ← PC, PC ← R[3]

Type IV

Six instructions belong to the instruction format Type IV. These are

1. add (op-code = 0)
   This instruction adds contents of a register to those of another register, and assigns to the destination register. An example:

   and r4, r3, r5

   R[4] ← R[3] +R[5]

2. sub (op-code = 2)
   This instruction subtracts value of a register from another the value stored in another register, and assigns to the destination register. For example,

   sub r4, r3, r5

   In RTL, this is denoted by

   R[4] ← R[3] – R[5]

3. mul (op-code = 4)
   The multiply instruction will store the product of two register values, and stores in the destination register. An example is

   mul r5, r7, r1

   The RTL notation for this instruction will be

   R[0] © R[5] ← R[7]*R[1]

4. div (op-code= 5)
   This instruction will divide the value of the register that is the second operand, by the number in the register specified by the third operand, and assign the result to the destination register.
   div r4, r7, r2 R[4]←R[0] ©R[7]/R[2],R[0]←R[0] ©R[7]%R[2]

5. and (op-code= 8)

This 'and' instruction will obtain a bit-wise 'and' of the values of two registers and assigns it to a destination register. For instance, in the following example, contents of register r4 and r5 are bit-wise 'anded' and the result is assigned to the register r1.
and r1, r4, r5
In RTL we may write this as
R[1] ← R[4] & R[5]

6. or (op-code= 10)
To bit-wise 'or' the contents of two registers, this instruction is used. For instance,
or r6, r7,r2
In RTL this is denoted as
R[6] ← R[7] ~ R[2]

FALCON-A: Instruction Set Summary

We have looked at the various types of instruction formats for the FALCON-A, as well as the instructions that belong to each of these instruction format types. In this section, we have simply listed the instructions on the basis of their functional groups; this means that the instructions that perform similar class of operations have been listed together.

Data Transfer Instructions	Mnemonic	opcode
move	mov	00110 (6)
Move immediate	movi	00111 (7)
Input to register	in	11000 (24)
Output from register	out	11001 (25)
Load from memory	load	11101 (29)
Store into memory	store	11100 (28)

Fig. Data Transfer Instructions

jump instruction	Mnemonic	opcode
jump if positive	jpl	10000 (16)
jump if negative	jmi	10001 (17)
jump if not zero	jnz	10010 (18)
jump if zero	jz	10011 (19)
jump	jump	10100 (20)

Fig. Jump Instructions

Control Instruction	Mnemonic	opcode
No operation	nop	10101 (21)
call	call	10110 (22)
return	ret	10111 (23)
interrupt	int	11010 (26)
Interupt return	iret	11011 (27)
reset	reset	11110 (30)
halt	halt	11111 (31)

Fig. Control Instructions

Examples for FALCON-A

In this section we take up a few sample problems related to the FALCON-A processor. This will enhance our understanding of the FALCON-A processor, as well as of the general concepts related to general processors and their instruction set architectures. The problems we will look at include

1. Identification of the instruction types and operands
2. Addressing modes and RTL description
3. Branch condition and status of the PC
4. Binary encoding for instructions

Example 1:

Identify the types of given FALCON-A instructions and specify the values in the fields

Instruction	Type	ra	rb	rc	c1	c2
movi r1, 2						
add r1,r2,r3						
nop						
load r2,[r5 + 6]						
jz r0, [3]						

Fig. Example 1

Solution

The solution to this problem is quite straightforward. The types of these instructions, as well as the fields, have already been discussed in the preceding sections.

Instruction	Type	ra	rb	rc	c1	c2
movi r1, 2	II	r1	-	-	-	2
add r1,r2,r3	IV	r1	r2	r3	-	-
nop	I	-	-	-	-	-
load r2,[r5 + 6]	III	r2	r5	-	6	-
jz r0, [3]	II	r0	-	-	-	3

Fig. Solution 1

We can also find the machine code for these instructions. The machine code (in the hexadecimal representation) is given for these instructions in the given table.

Instruction	Machine Code	ra	rb	rc	c1	c2
movi r1, 2	3902h	r1	-	-	-	2
add r1,r2,r3	014Ch	r1	r2	r3	-	-
nop	A800h	-	-	-	-	-
load r2,[r5 + 6]	EAA6h	r2	r5	-	6	-
jz r0, [3]	9803h	r0	-	-	-	3

Fig. Machine Code

Example 2:
Identify the addressing modes and Register Transfer Language (RTL) description (meaning) for the given FALCON-A instructions

Instruction	Addressing mode	RTL description (meaning)
load r2,[r4 + 8]		
jnz r1,[54]		
shiftl r1,r2,4		
addi r3,r6,2		
sub r1, r7,r2		

Fig. Example 2

Solution

Addressing modes relate to the way architectures specify the address of the objects they access. These objects may be constants and registers, in addition to memory locations.

Instruction	Addressing mode	RTL description (meaning)
load r2,[r4 + 8]	Displacement	R[2]←M[R[4]+8]
jnz r1, [54]	Relative	(R[1]≠0): PC←PC+54
shiftl r1,r2,4	Immediate	Shift r2 left 4 times and store in r1
addi r3,r6,2	Immediate	R[3]←R[6]+2
sub r1, r7,r2	Register	R[1]←R[7]-R[2]

Fig. Solution 2

Example 3: Specify the condition for the branch instruction and the status of the PC after the branch instruction executes with a true branch condition

Instruction	Condition	PC status
jz r2,[35]		
jump [12]		
jnz r6, [3]		
jpl r1, [45]		
jmi r2, [20]		

Fig. Example 3

Solution

We have looked at the various jump instructions in our study of the FALCON-A. Using that knowledge, this problem can be solved easily.

Instruction	Condition	PC status
jz r2,[35]	If R[2]==0	PC ← PC+35
jump [12]	always	PC ← PC+12
jnz r6, [3]	If R[6] ≠ 0	PC ← PC+3
jpl r1, [45]	If R[1] ≥ 0	PC ← PC+45
jmi r2, [20]	If R[2] < 0	PC ← PC+20

Fig. Solution 3

Example 4: Specify the binary encoding of the different fields in the given FALCON-A instructions.

Instruction	TYPE	opcode	ra	rb	rc	C1(5 bits) OR C2(8 bits)
store r4, [r1+8]						
sub r3,r6,r5						
shiftr r4,r6,9						
jump [10]						
halt						

Fig. Example 4

Solution

We can solve this problem by referring back to our discussion of the instruction format types. The op-codes for each of the instructions can also be looked up from the tables. ra, rb and rc (where applicable) registers' values are obtained from the register encoding table we looked at. The constants C1 and C2 are there in instruction type III and II respectively. The immediate constant specified in the instruction can also be simply converted to binary, as shown.

Instruction	TYPE	opcode	ra	rb	rc	C1(5 bits) OR C2(8 bits)
store r4, [r1+8]	III	11100	100	001	-	01000
sub r3,r6,r5	IV	00010	011	110	101	-
shiftr r4,r6,9	III	01101	100	110	-	01001
jump [10]	II	10100	-	-	-	0000 1010
halt	I	11111	-	-	-	-

Fig. Solution 4

Lecture No. 9

Description of FALCON-A and EAGLE using RTL

Reading Material

Handouts Slides

Summary

- Use of Behavioral Register Transfer Language (RTL) to describe the FALCON-A
- The EAGLE
- The Modified EAGLE

Use of Behavioral Register Transfer Language (RTL) to describe the FALCON-A

The use of RTL (an acronym for the Register Transfer Language) to describe the FALCON-A is discussed in this section. FALCON-A is the sample machine we are building in order to enhance our understanding of processors and their architecture.

Behavior vs. Structure

Computer design involves various levels of abstraction. The behavioral description of a machine is a higher level of abstraction, as compared with the structural description. Top-down approach is adopted in computer design. Designing a computer typically starts with defining the behavior of the overall system. This is then broken down into the behavior of the different modules. The process continues, till we are able to define, design and implement the structure of the individual modules.

As mentioned earlier, we are interested in the behavioral description of our machine, the FALCON-A, in this section.

Register Transfer Language

The RTL is a formal way of expressing the behavior and structure of a computer.

Behavioral RTL

Behavioral Register Transfer Language is used to describe what a machine does, i.e. it is used to define the functionality the machine provides. Basically, the behavioral architecture describes the algorithms used in a machine, written as a set of process statements. These statements may be sequential statements or concurrent statements, including signal assignment statements and wait statements.

Structural RTL

Structural RTL is used to describe the hardware implementation of the machine. The structural architecture of a machine is the logic circuit implementation (components and their interconnections), that facilitates a certain behavior (and hence functionality) for that machine.

Using RTL to describe the static properties of the FALCON-A

We can employ the RTL for the description of various properties of the FALCON-A that we have already discussed.

Specifying Registers

In RTL, we will refer to a register by its abbreviated, alphanumeric name, followed by the number of bits in the register enclosed in angle brackets '< >'. For instance, the instruction register (IR), of 16 bits (numbered 0 to 15), will be referred to as, IR<15..0>

Naming of the Fields in a Register

We can name the different fields of a register using the := notation. For example, to name the most significant bits of the instruction register as the operation code (or simply op), we may write:

op<4..0> := IR<15..11>

Note that using this notation to name registers or register fields will not create a new copy of the data or the register fields; it is simply an alias for an already existing register, or part of a register.

Fields in the FALCON-A Instructions

We now use the RTL naming operator to name the various fields of the RTL instructions. Naming the fields appropriately helps us make the study of the behavior of a processor more readable.

op<4..0>:= IR<15..11>:	**operation code field**
ra<2..0> := IR<10..8>:	**target register field**
rb<2..0> := IR<7..5>:	**operand or address index**
rc<2..0> := IR<4..2>:	**second operand**
c1<4..0> := IR<4..0>:	**short displacement field**
c2<7..0> := IR<7..0>:	**long displacement or the immediate field**

We are already familiar with these fields, and their usage in the various instruction formats of the RTL.

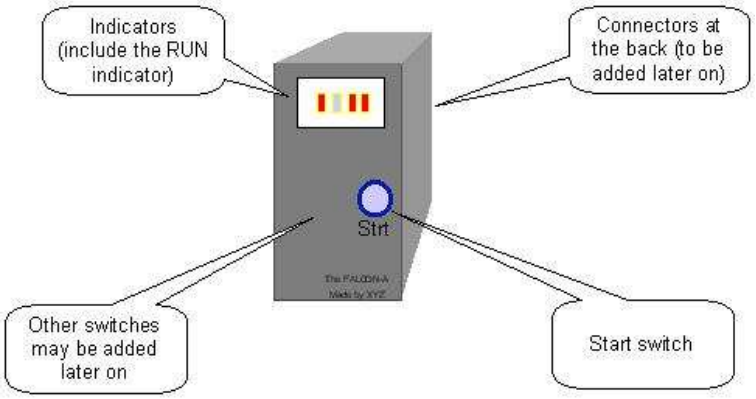
Describing the Processor State using RTL

The processor state defines the contents of all the register internal to the CPU at a given time. Maintaining or restoring the machine or processor state is important to many operations, especially procedure calls and interrupts; the processor state needs to be restored after a procedure call or an interrupt so normal operation can continue. Our processor state consists of the following:

PC<15..0>:	**program counter (the PC holds the memory address of the next instruction)**
IR<15..0>:	**instruction register (used to hold the current instruction)**
Run:	**one bit run/halt indicator**
Strt:	**start signal**
R [0..7]<15..0>:	**8 general purpose registers, each consisting of 16 bits**

FALCON-A in a black box

The given figure shows what a processor appears as to a user. We see a start button that is basically used to start up the processor, and a run indicator that turns on when the processor is in the running state.



There may be several other indicators as well. The start button as well as the run indicator can be observed on many machines.

Using RTL to describe the dynamic properties of the FALCON-A

We have just described some of the static properties of the FALCON-A. The RTL can also be employed to describe the dynamic behavior of the processor in terms of instruction interpretation and execution.

Conditional expressions can be specified using the RTL. For instance, we may specify a conditional subtraction operation employing RTL as

(op=2) : R[ra] ← R[rb] - R[rc];

This instruction means that "if" the operation code of the instruction equals 2 (00010 in binary), then subtract the value stored in register rc from that of register rb, and store the resulting value in register ra.

Effective address calculations in RTL (performed at runtime)

The operand or the destination address may not be specified directly in an instruction, and it may be required to compute the effective address at run-time. Displacement and relative addressing modes are instances of such situations. RTL can be used to describe these effective address calculations.

Displacement address

A displacement address is calculated, as shown:

disp<15..0> := (R[rb]+ (11α c1<4>)© c1<4..0>);

This means that the address is being calculated by adding the constant value specified by the field c1 (which is first sign extended), to the value specified by the register rb.

Relative address

A relative address is calculated by adding the displacement to the contents of the program counter register (that holds the instruction to be executed next in a program flow). The constant is first sign-extended. In RTL this is represented as, rel<15..0>:=PC+(8αc2<7>)©c2<7..0>;

Range of memory addresses

Using the displacement or the relative addressing modes, there is a specific range of memory addresses that can be accessed.

- Range of addresses when using direct addressing mode (displacement with rb=0)
  - If c1<4>=0 (positive displacement) absolute addresses range: 00000b to 01111b (0 to +15)
  - If c1<4>=1 (negative displacement) absolute addresses range: 11111b to 10000b (-1 to -16)
- Address range in case of relative addressing
  - The largest positive value that can be specified using 8 bits (since we have only 8 bits available in c2<7..0>), is 27-1, and the most negative value that can be represented using the same is 27. Therefore, the range of addresses or locations that can be referred to using this addressing mode is 127 locations forward or 128 locations backward from the Program Counter (PC).

Instruction Fetch Operation (using RTL)

We will now employ the notation that we have learnt to understand the fetch-execute cycle of the FALCON-A processor.

The RTL notation for the instruction fetch process is

```
instruction_Fetch := (
        !Run&Strt : Run ← 1,
        Run : (IR ← M[PC], PC ← PC + 2;
                instruction_Execution) );
```

This is how the instruction-fetch phase of the fetch-execute cycle for FALCON-A can be represented using RTL. Recall that ":=' is the naming operator, "!" implies a logical NOT, "&" implies a logical AND, "←" represents a transfer operation, ";" is used to separate sequential statements, and concurrent statements are separated by ",". We can observe that in the instruction_Fetch phase, if the machine is not in the running state and the start bit has been set, then the run bit is also set to true. Concurrently, an instruction is fetched from the instruction memory; the program counter (PC) holds the next instruction address, so it is used to refer to the memory location from where the instruction is to be fetched. Simultaneously, the PC is incremented by 2 so it will point to the next instruction. (Recall that our instruction word is 2 bytes long, and the instruction memory is organized into 1-byte cells). The next step is the instruction execution phase. Difference between "," and ";" in RTL

We again highlight the difference between the "," and ";". Statements separated by a "**,**" take place during the same clock pulse. In other words, the order of execution of statements separated by "**,**" does not matter.

On the other hand, statements separated by a "**;**" take place on successive clock pulses. In other words, if statements are separated by "**;**" the one on the left must complete before the one on the right starts. However, some things written with one RTL statement can take several clocks to complete.

We return to our discussion of the instruction-fetch phase. The statement

!Run&Strt : Run ← 1

is executed when 'Run' is 0, and 'Strt' is 1, that is, Strt has been set. It is used to set the Run bit. No action takes place when both 'Run' and 'Strt' are 0.

The following two concurrent register transfers are performed when 'Run' is set to 1, (as ':' is a conditional operator; if the condition is met, the specified action is taken).

IR ← M[PC]
PC ← PC + 2

Since these instructions appear concurrent, and one of the instructions is using the value of PC that the other instruction is updating, a question arises; which of the two values of the PC is used in the memory access? As a rule, all right hand sides of the register transfers are evaluated before the left hand side is evaluated/updated. In case of simultaneous register transfers (separated by a ","), all the right hand side expressions are evaluated in the same clock-cycle, before they are assigned. Therefore, the old, un-incremented value of the PC is used in the memory access, and the incremented value is assigned to the PC afterwards. This corresponds to "master-slave" flip-flop operation in logic circuits.

This makes the PC point to the next instruction in the instruction memory. Once the instruction has been fetched, the instruction execution starts. We can also use i.F for

instruction_Fetch and i.E for instruction_Execution. This will make the Fetch operation easy to write.

```
iF := ( !Run&Strt : Run ← 1, Run : (IR ← M[PC], PC ← PC + 2; iE ) );
```

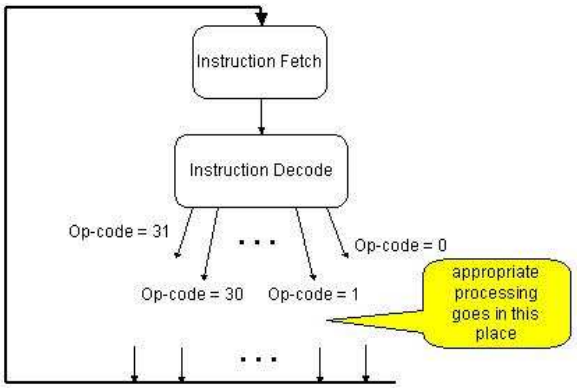
Instruction Execution (Describing the Execute operation using RTL)

Once an instruction has been fetched from the instruction memory, and the program counter has been incremented to point to the next instruction in the memory, instruction execution commences. In the instruction fetch-execute cycle we showed in the preceding discussion, the

entire instruction execution code was aliased iE (or instruction_Execution), through the assignment operator ":=". Now we look at the instruction execution in detail.

iE := (
(op<4..0>= 1) : R[ra] ← R[rb]+ (11α c1<4>)© c1<4..0>,
(op<4..0>= 2) : R[ra] ← R[rb]-R[rc],
. . .
. . .
(op<4..0>=31) : Run ← 0,); iF);

As we can see, the instruction execution can be described in RTL by using a long list of concurrent, conditional operators that are inherently 'disjoint'. Being inherently disjointed implies that at any instance, only one of the conditions can be met; hence one of the statements is executed. The long list of statements is basically all of the instructions that are a part of the FALCON-A instruction set, and the condition for their execution is related to the operation code of the instruction fetched. We will take a closer look at the entire list in our subsequent discussion. Notice that in the instruction execute phase, besides the long list of concurrent, disjoint instructions, there is also the instruction fetch or iF sequenced at the end. This implies that once one of the instructions from the list is executed, the instruction fetch is called to fetch the next instruction. As shown before, the instruction fetch will call the instruction execute after fetching a certain instruction, hence the instruction fetch-execute cycle continues.



The instruction fetch-execute cycle is shown schematically in the above given figure. We now see how the various instructions in the execute code of the fetch-execute cycle of FALCON-A, are represented using the RTL. These instructions form the instruction set of the FALCON-A.

Jump instructions

Some of the instructions listed for the instruction execution phase are jump instruction, as shown. (Note '. . .' implies that more instructions may precede or follow, depending on whether it is placed before the instructions shown, or after).

iE := (
. . .
. . .

If op-code is 20, the branch is taken unconditionally (the jump instruction).

(op<4..0>=20) : (cond : (PC ← R[ra]+C2(sign extended)),

If the op-code is 16, the condition for branching is checked, and if the condition is being met, the branch is taken; otherwise it remains untaken, and normal program flow will continue.

(op<4..0>= 16) : (cond : (PC ← PC+C2 (sign extended))
. . .
. . .

Arithmetic and Logical Instructions

Several instructions provide arithmetic and logical operations functionality. Amongst the list of concurrent instructions of the iE phase, the instructions belonging to this category are highlighted:

iE := (
. . .

. . .

If op-code is 0, the instruction is 'add'. The values in register rb and rc are added and the result is stored in register ra

(op<4..0>=0) : R[ra] ← R[rb] + R[rc],

Similarly, if op-code is 1, the instruction is addi; the immediate constant specified by the constant field C1 is sign extended and added to the value in register rb. The result is stored in the register ra.

(op<4..0>=1) : R[ra] ←R[rb] + (11α C1<4>)© C1<4..0>,

For op-code 2, value stored in register rc is subtracted from the value stored in register rb, and the result is stored in register ra.

(op<4..0>=2) : R[ra] ← R[rb] - R[rc],

If op-code is 3, the immediate constant C1 is sign-extended, and subtracted from the value stored in rb. Result is stored in ra.

(op<4..0>=3) : R[ra] ← R[rb]- (11α C1<4>)© C1<4..0>,

For op-code 4, values of rb and rc register are multiplied and result is stored in the destination register.

(op<4..0>=4) : R[ra] ← R[rb] * R[rc],

If the op-code is 5, contents of register rb are divided by the value stored in rc, result is concatenated with 0s, and stored in ra. The remainder is stored in R0.

(op<4..0>=5) : R[ra] ← R[0] ©R[rb]/R[rc],
R[0] ← R[0] ©R[rb]%R[rc],

If op-code equals 8, bit-wise logical AND of rb and rc register contents is assigned to ra.

(op<4..0>=8) : R[ra] ← R[rb] & R[rc],

If op-code equals 8, bit-wise logical OR of rb and rc register contents is assigned to ra.

(op<4..0>=10) : R[ra] ← R[rb] ~ R[c],

For op-code 14, the contents of register specified by field rc are inverted (logical NOT is taken), and the resulting value is stored in register ra.

(op<4..0>=14) : R[ra] ← ! R[rc],

. . .
. . .

Shift Instructions

The shift instructions are also a part of the instruction set for FALCON-A, and these are listed in the instruction execute phase in the RTL as shown.

iE := (
. . .
. . .

If the op-code is 12, the contents of the register rb are shifted right N bits. N is the number specified in the constant field. The space that has been created due to the shift out of bits is filled with 0s through concatenation. In RTL, this is shown as:

(op<4..0>=12) : R[ra]<15..0> ← R [rb]<(15-N)..0>©(Nα0),

If op-code is 13, rb value is shifted left, and 0s are inserted in place of shifted out contents at the right side of the value. The result is stored in ra.

(op<4..0>=13) : R[ra]<15..0> ← (Nα0)©R [rb]<(15)..N>,

For op-code 15, arithmetic shift right operation is carried out on the value stored in rb. The arithmetic shift right shifts a signed binary number stored in the source register to the right, while leaving the sign-bit unchanged. Note that α means replication, and © means concatenation.

(op<4..0>=15) : R[ra]<15..0> ← Nα(R [rb]<15>)© (R [rb]<15..N>),

. . .
. . .

Data transfer instructions

Several of the instructions belong to the data transfer category.

iE := (

. . .

. . .

Op-code 29 specifies the load instruction, i.e. a memory location is referenced and the value stored in the memory location is copied to the destination register. The effective address of the memory location to be referenced is calculated by sign extending the immediate field, and adding it to the value specified by register rb.

(op<4..0>=29) : R[ra]← M[R[rb]+ (11α C1<4>)© C1<4..0>],

A value is stored back to memory from a register using the op-code 28. The effective address in memory where the value is to be stored is calculated in a similar fashion as the load instruction.

(op<4..0>=28) : M[R[rb]+ (11α C1<4>)© C1<4..0>] ← R [ra],

The move instruction has the op-code 6. The contents of one register are copied to another register through this instruction.

(op<4..0>=6) : R[ra] ← R[rb],

To store an immediate value (specified by the field C2 of the instruction) in a register, the op-code 7 is employed. The constant is first sign-extended.

(op<4..0>=7) : R[ra] ← (8αC2<7>)©C2<7..0>,

If the op-code is 24, an input is obtained from a certain input device, and the input word is stored into register ra. The input device is selected by specifying its address through the constant C2.

(op<4..0>=24) : R[ra] ← IO[C2],

Unconditional branch (jump)If the op-code is 25, an output (the register ra value) is sent to an output device (where the address of the output device is specified by the constant C2).

(op<4..0>=25) : IO[C2] ← R[ra],

. . .

. . .

Miscellaneous instructions

Some more instruction included in the FALCON-A are

iE := (

. . .

. . .

The no-operation (nop) instruction, if the op-code is 21. This instructs the processor to do nothing.

(op<4..0>= 21) : ,

If the op-code is 31, setting the run bit to 0 halts the processor.

(op<4..0>= 31) : Run ← 0, Halt the processor (halt)

At the end of this concurrent list of instructions, there is an instruction i.F (the instruction fetch). Hence when an instruction is executed, the next instruction is fetched, and the cycle continues, unless the processor is halted.

); iF);

Note: For Assembler and Simulator Consult Appendix.

The EAGLE

(Original version)

Another processor that we are going to study is the EAGLE. We have developed two versions of it, an original version, and a modified version that takes care of the limitations in the original

version. The study of multiple processors is going to help us get thoroughly familiar with the processor design, and the various possible designs for the processor. However, note that these machines are simplified versions of what a real machine might look like.

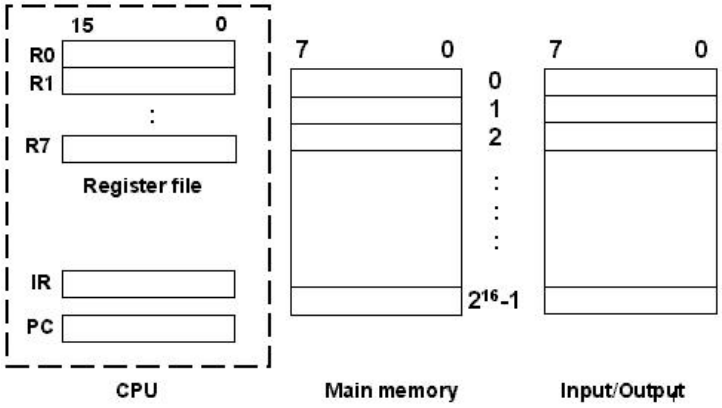
Introduction

The EAGLE is an accumulator-based machine. It is a simple processor that will help us in our understanding of the processor design process. EAGLE is characterized by the following:

- Eight General Purpose Registers of the CPU. These are named R0, R1…R7. Each register is 16-bits in length.
- Two 16-bit system registers transparent to the programmer are the Program Counter (PC) and the Instruction Register (IR). (Being transparent to the programmer implies the programmer may not directly manipulate the values to these registers. Their usage is the same as in any other processor)
- Memory word size is 16 bits
- The available memory space size is 2^{16} bytes
- Memory organization is 2^{16} x 8 bits. This means that there are 2^{16} memory cells, each one byte long.
- Memory is accessed in 16 bit words (i.e., 2 byte chunks)
- Little-endian byte storage is employed.

Programmer's View of the EAGLE

The programmer's view of the EAGLE processor is shown by means of the given figure.

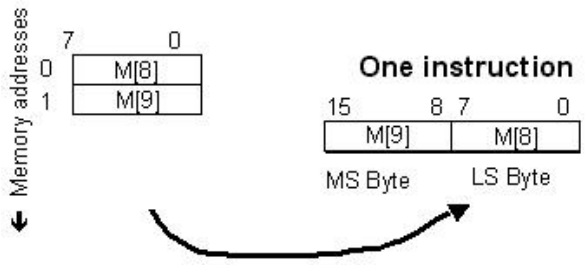


EAGLE: Notation

Let us take a look at the notation that will be employed for the study of the EAGLE.

Enclosing the register name in square brackets refers to register contents; for instance, R[3] means contents of register R3.

Enclosing the location address in square brackets, preceded by 'M', lets us refer to **memory contents**. Hence M [8] means contents of memory location 8.



As little endian storage is employed, a **memory word** at address x is defined as the 16 bits at address x +1 and x. For instance, the bits at memory location 9,8 define the memory word at location 8. So employing the special notation for 16-bit memory words, we have

M [8]<15…0>:=M [9]©M [8]

Where © is used to represent concatenation

EAGLE Features

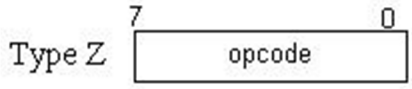
The following features characterize the EAGLE.

- Instruction length is variable. Instructions are either 8 bits or 16 long, i.e., instruction size is either 8-bits or 16-bits.
- The instructions may have either one or two operands.
- The only way to access memory is through load and store instructions.
- Limited addressing modes are supported

EAGLE: Instruction Formats
There are five instruction formats for the EAGLE. These are

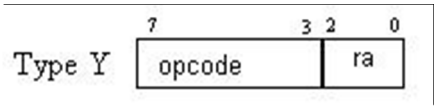
Type Z Instruction Format

The Z format instructions are half-word (1 byte) instructions, containing just the op-code field of 8 bits, as shown



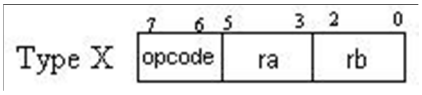
Type Y Instruction Format

The type Y instructions are also half-word. There is an op-code field of 5 bits, and a register operand field ra.

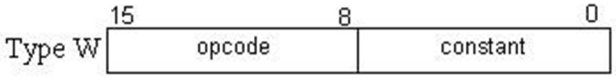


Type X Instruction Format

Type X instructions are also half-word instructions, with a 2-bit op-code field, and two 3-bit operand register fields, as shown. **Type W instruction format**

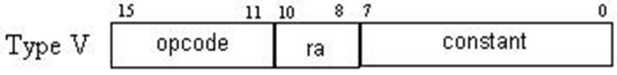


The instructions in this type are 1-word (16-bit) in length. 8 bits are reserved for the op-code, while the remaining 8 bits form the constant (immediate value) field.



Type V instruction format

Type V instructions are also 1-word



instructions, containing an op-code field of 5 bits, an operand register field of 3 bits, and 8bits for a specifying a constant.

Encoding of the General Purpose Registers

The encoding for the eight GPRs is shown in the table. These binary codes are to be used in place of the 'place-holders' ra, rb in the actual instructions of the processor EAGLE.

Register	Code	Register	Code
R0	000	R4	100
R1	001	R5	101
R2	010	R6	110
R3	011	R7	111

Listing of EAGLE instructions with respect to instruction formats

The following is a brief introduction to the various instructions of the processor EAGLE, categorized with respect to the instruction formats.

Type Z

There are four type Z instructions,

- halt(op-code=250)
  This instruction halts the processor
- nop(op-code=249)
  nop, or the no-operation instruction stalls the processor for the time of execution of a single instruction. It is useful in pipelining.
- init(op-code=251)
  This instruction is used to initialize all the registers, by setting them to 0
- reset(op-code=248)

This instruction is used to initialize the processor to a known state.In this instruction the control step counter is set to zero so that the operation begins at the start of the instruction fetch and besides this PC is also set to a known value so that machine operation begins at a known instruction.

Type Y

Seven instructions of the processor are of type Y. These are

- add(op-code=11)
  The type Y add instruction adds register ra's contents to register R0. For example, add r1
  In the behavioral RTL, we show this as
  R[0] ← R[1]+R[0]

- and(op-code=19)
  This instruction obtains the logical AND of the value stored in register specified by field ra and the register R0, and assigns the result to R0, as shown in the example:
  and r5
  which is represented in RTL as R[0] ← R[1]&R[0]

- div(op-code=16)
  This instruction divides the contents of register R0 by the value stored in the register ra, and assigns result to R0. The remainder is stored in the divisor register, as shown in example,
  div r6
  In RTL, this is
  R[0] ← R[0]/R[6]
  R[6] ← R[0]%R[6]

- mul (op-code = 15)
  This instruction multiplies the values stored in register R0 and the operand register, and assigns the result to R0). For example,
  mul r4
  In RTL, we specify this as R[0] ← R[0]*R[4]

- not (op-code = 23)
  The not instruction inverts the operand register's value and assigns it back to the same register, as shown in the example
  not r6
  R[6] ← ! R[6]

- or (op-code=21)
  The or instruction obtains the bit-wise OR of the operand register's and R0's value, and assigns it back to R0. An example,
  or r5
  R[0] ← R[0] ~ R[5]

- sub (op-code=12)
  The sub instruction subtracts the value of the operand register from R0 value, assigning it back to register R0. Example:
  sub r7
  In RTL: R[0] ← R[0] – R[7]

Type X

Only one instruction falls under this type. It is the 'mov' instruction that is useful for register transfers

- mov (op-code = 0)

  The contents of one register are copied to the destination register ra.
  Example: mov r5, r1
  RTL Notation: R[5]← R[1]

Type W

Again, only one instruction belongs to this type. It is the branch instruction

- br (op-code = 252)

  This is the unconditional branch instruction, and the branch target is specified by the 8-bit immediate field. The branch is taken by incrementing the PC with the new value. Hence it is a 'near' jump. For instance,
  br 14
  PC ← PC+14

Most of the instructions of the processor EAGLE are of the format type V. These are

- addi (op-code = 13)

  The addi instruction adds the immediate value to the register ra, by first sign-extending the immediate value. The result is also stored in the register ra. For example,
  addi r4, 31
  In behavioral RTL, this is
  R[4] ← R[4]+(8αc<7>)©c<7…0>;
- andi (op-code = 20)

  Logical 'AND' of the immediate value and register ra value is obtained when this instruction is executed, and the result is assigned back to register ra. An example, andi r6, 1
  R[6] ← R[6] &1
- in (op-code=29)

  This instruction is to read in a word from an IO device at the address specified by the immediate field, and store it in the register ra. For instance,
  in r1, 45
  In RTL this is
  R[1] ← IO[45]
- load (op-code=8)

  The load instruction is to load the memory word into the register ra. The immediate field specifies the location of the memory word to be read. For instance,
  load r3, 6
  R[3] ← M[6]
- brn (op-code = 28)

  Upon the brn instruction execution, the value stored in register ra is checked, and if it is negative, branch is taken by incrementing the PC by the immediate field value. An example is
  brn r4, 3
  In RTL, this may be written as if R[4]<0, PC ← PC+3
- brnz (op-code = 25)

  For a brnz instruction, the value of register ra is checked, and if found non-zero, the PC-relative branch is taken, as shown in the example,
  brnz r6, 12 Which, in RTL is
  if R[6]!=0, PC ← PC+12

- brp (op-code=27)
  brp is the 'branch if positive'. Again, ra value is checked and if found positive, the PC-relative near jump is taken, as shown in the example:
  brp r1, 45
  In RTL this is
  if R[1]>0, PC ← PC+45
- brz (op-code=8)
  In this instruction, the value of register ra is checked, and if it equals zero, PC-relative branch is taken, as shown,
  brz r5, 8
  In RTL:
  if R[5]=0, PC ← PC+8
- loadi (op-code=9)
  The loadi instruction loads the immediate constant into the register ra, for instance,
  loadi r5,54 R[5] ← 54
- ori (op-code=22)
  The ori instruction obtains the logical 'OR' of the immediate value with the ra register value, and assigns it back to the register ra, as shown,
  ori r7, 11 In RTL,
  R[7] ← R[7]~11
- out (op-code=30)
  The out instruction is used to write a register word to an IO device, the address of which is specified by the immediate constant. For instance,
  out 32, r5
  In RTL, this is represented by IO[32] ← R[5]
- shiftl (op-code=17)
  This instruction shifts left the contents of the register ra, as many times as is specified through the immediate constant of the instruction. For example: shiftl r1, 6
- shiftr(op-code=18)
  This instruction shifts right the contents of the register ra, as many times as is specified through the immediate constant of the instruction. For example: shiftr r2, 5
- store (op-code=10)
  The store instruction stores the value of the ra register to a memory location specified by the immediate constant. An example is,
  store r4, 34
  RTL description of this instruction is M[34] ← R[4]
- subi (op-code=14)
  The subi instruction subtracts the immediate constant from the value of register ra, assigning back the result to the register ra. For instance,
  subi r3, 13

  RTL description of the instruction
  R[3] ← R[3]-13

(ORIGINAL) ISA for the EAGLE

(16-bit registers, 16-bit PC and IR, 8-bit memory)

mnemonic	opcode	operand1 3 bits	operand2 3 bits	constant 8 bits	Format	Behavioral RTL
add	01011	ra	-	-	Y	R [0] ← R [ra]+R [0];
addi	01101	ra	-	c	V	R [ra] ← R [ra]+(8αc<7>)©c;
and	10011	ra	-	-	Y	R[0] ← R[ra]&R[0];
andi	10100	ra	-	c	V	R [ra] ← R [ra]& (8αc<7>)©c;
br	11111100	-	-	c	W	PC ← PC+(8αc<7>)©c;
brnv	11100	ra	-	c	V	(R [ra]<0): PC ← PC+(8αc<7>)©c;
brnz	11001	ra	-	c	V	(R [ra]<>0): PC ← PC+(8αc<7>)©c;
brpl	11011	ra	-	c	V	(R [ra]>0): PC ← PC+(8αc<7>)©c;
brzr	11010	ra	-	c	V	(R [ra]=0): PC ← PC+(8αc<7>)©c;
div	10000	ra	-	-	Y	R [0] ← R [0]/R [a], R [ra] ←R [0]%R [ra],
halt	11111010	-	-	-	Z	RUN← 0;
in	11101	ra	-	c	V	R [ra] ←IO[c];
init	11111011	-	-	-	Z	R [7…0] ← 0;
load	01000	ra	-	c	V	R [ra] ←M[c];
loadi	01001	ra	-	c	V	R [ra] ← (8αc<7>)©c;
mov	00	ra	rb	-	X	R [ra] ← R [rb];
mul	01111	ra	-	-	Y	R [ra] © R [r0] ← R [ra]*R [0];
nop	11111001	-	-	-	Z	;
not	10111	ra	-	-	Y	R [ra] ←! (R [ra]);
or	10101	ra	-	-	Y	R [0] ← R [ra]~R [0];
ori	10110	ra	-	c	V	R [ra] ← R [ra]~ (8αc<7>)©c;
out	11110	ra	-	c	V	IO[c] ←R [ra];
reset	11111000	-	-	-	Z	TBD;
shiftl	10001	ra	-	c	V	R [ra] ← R [ra]<(7-n)..0>©(nα0);
shiftr	10010	ra	-	c	V	R [ra] ← (nα0)©R [ra]<7...n>;
store	01010	ra	-	c	V	M[c]← R [ra];
sub	01100	ra	-	-	Y	R [0] ← R [0]-R [a];
subi	01110	ra	-	c	V	R [ra] ← R [ra]- (8αc<7>)©c;

Symbol	Meaning	Symbol	Meaning
α	Replication	%	Remainder after integer division
©	Concatenation	&	Logical AND
:	Conditional constructs (IF-THEN)	~	Logical OR
;	Sequential constructs	!	Logical NOT or complement
,	Concurrent constructs	←	LOAD or assignment operator

Limitations of the ORIGINAL EAGLE ISA

The original 16-bit ISA of EAGLE has severe limitations, as outlined below.

1. Use of R0 as accumulator

In most cases, the register R0 is being used as one of the source operands as well as the destination operand. Thus, R0 has essentially become the accumulator. However, this will require some additional instructions for use with the accumulator. That should not be a problem since there are some unused op-codes available in the ISA. Unequal and inefficient op-code assignment

The designer has apparently tried to extend the number of operations in the ISA by op-code extension. Op-code 11111 combine three additional bits of the instruction for five instructions: unconditional branch, nop, halt, reset and init.while there is a possibility of including three more instructions in this scheme, notice that op-code 00 for register to register mov is causing a "loss" of eight "slots" in the original 5-bit op-code assignment. (The mov instruction is, in effect, using eight op-codes). A better way would be to assign

a 5-bit op-code to mov and use the remaining op-codes for other instructions. Number of the operands

Looking at the mov instruction again, it can be noted that this is the only instruction that uses two operands, and thus requires a separate format (Format#1) for instruction enoding. If the job of this instruction is given to two instructions (copy register to accumulator, and copy accumulator to register), the number of instruction formats can be reduced thereby, simplifying the assembler and the compiler needed for this ISA.

2. Use of registers for branch conditions

Note that one of the GPRs is being used to hold the branch condition. This would require that the result from the accumulator be copied to the particular GPR before the branch instruction. Including flags with the ALSU can eliminate this restriction

The Modified EAGLE

The modified EAGLE is an improved version of the processor EAGLE. As we have already discussed, there were several limitations in EAGLE, and these have been remedied in the modified EAGLE processor.

Introduction

The modified EAGLE is also an accumulator-based processor. It is a simple, yet complex enough to illustrate the various concepts of a processor design. The modified EAGLE is characterized by

- A special purpose register, the 16-bit accumulator: ACC
- 8 General Purpose Registers of the CPU: R0, R1, …, R7; 16-bits each
- Two 16-bit system registers transparent to the programmer are the Program Counter (PC) and the Instruction Register (IR).
- Memory word size: 16 bits
- Memory space size: 2^{16} bytes
- Memory organization: 2^{16} x 8 bits
- Memory is accessed in 16 bit words (i.e., 2 byte chunks)
- Little-endian byte storage is employed

Programmer's View of the Modified EAGLE

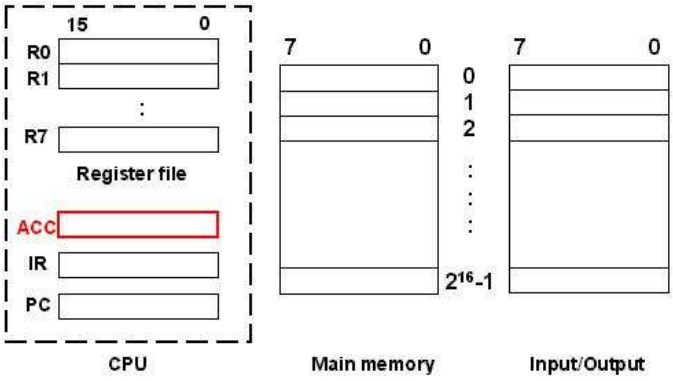
The given figure is the programmer's view of the modified EAGLE processor.

Notation

The notation that is employed for the study of the modified EAGLE is the same as the original EAGLE processor. Recall that we know that:

Enclosing the register name in square brackets refers to register contents; for instance, R [3] means contents of register R3.

Enclosing the location address in square brackets, preceded by 'M', lets us refer to **memory contents**. Hence M [8] means contents of memory location 8.

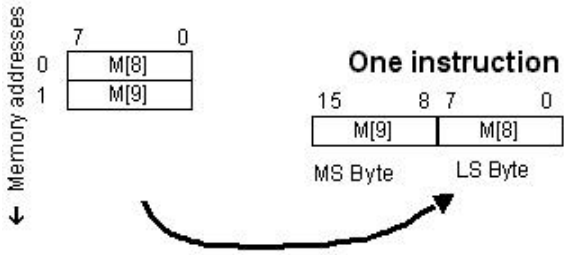


As little endian storage is employed, a **memory word** at address x is defined as the 16 bits at address x+1 and x. For instance, the bits at memory location 9,8 define the memory word at location 8. So employing the special notation for 16-bit memory words, we have

M[8]<15…0>:=M[9]©M[8]

Where © is used to represent concatenation

The memory word access and copy to a register is shown in the figure.

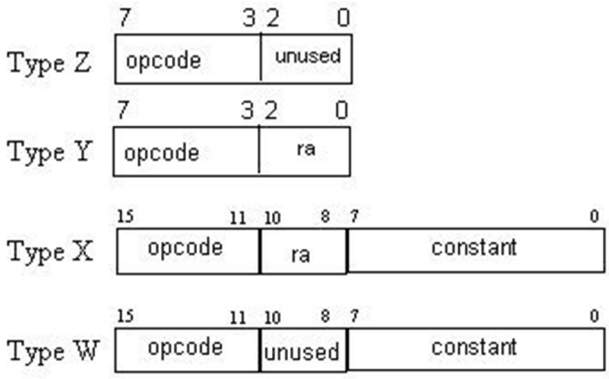


Features

The following features characterize the modified EAGLE processor.

- Instruction length is variable. Instructions are either 8 bits or 16 long, i.e., instruction size is either half a word or 1 word.
- The instructions may have either one or two operands.
- The only way to access memory is through load and store instructions
- Limited addressing modes are supported

Note that these properties are the same as the original EAGLE processor



Instruction formats

There are four instruction format types in the modified EAGLE processor as well. These are

Register	Code	Register	Code
R0	000	R4	100
R1	001	R5	101
R2	010	R6	110
R3	011	R7	111

Encoding of the General Purpose Registers

The encoding for the eight GPRs is shown in the table. These are binary codes assigned to the registers that will be used in place of the ra, rb in the actual instructions of the modified processor EAGLE.

ISA for the Modified EAGLE

(16-bit registers, 16-bit ACC, PC and IR, 8-bit wide memory, 256 I/O ports)

Mnemonic	Op-code	Operand 3bits	Constant 8 bits	Format	Behavioral RTL
Unused	00111				
addi	00100	ra	C1	X	ACC ← R[ra] +(8αC1<7>)©C1;
subi	00101	ra	C1	X	ACC ← R[ra] - (8αC1<7>)©C1;
shiftl	01010	ra	C1	X	R[ra] ← R[ra]<(15-n)..0>©(nα0);
shiftr	01011	ra	C1	X	R[ra] ← (nα0)©R[ra]<15...n>;
andi	01100	ra	C1	X	ACC ← R[ra] & (8αC1<7>)©C1;
ori	01101	ra	C1	X	ACC ← R[ra] ~ (8αC1<7>)©C1;
asr	01110	ra	C1	X	R[ra] ← (nαR[ra}<15>)©R[ra]<15...n>;
in	10001	ra	C1	X	R[ra] ←IO[C1];
ldacc	10010	ra	C1	X	ACC ←M[R[ra] +(8αC1<7>)©C1];
movir	10100	ra	C1	X	R[ra] ← (8αC1<7>)©C1;
out	10101	ra	C1	X	IO[C1] ←R[ra];
stacc	10111	ra	C1	X	M[R[ra] +(8αC1<7>)©C1]← ACC;
movia	10011		C1	W	ACC ← (8αC1<7>)©C1;
br	11000	-	C1	W	PC ← PC + 8αC1<7>)©C1;
brn	11001		C1	W	(S=1): PC ← PC+(8αC1<7>)©C1;
brnz	11010		C1	W	(Z=0): PC ← PC+(8αC1<7>)©C1;
brp	11011		C1	W	(S=0): PC ← PC+(8αC1<7>)©C1;
brz	11100		C1	W	(Z=1): PC ← PC+(8αC1<7>)©C1;
add	00000	ra	-	Y	ACC ← ACC + R[ra];
sub	00001	ra	-	Y	ACC ← ACC - R[a];
div	00010	ra	-	Y	ACC ← (R[ra] ©ACC)/R[a], R[ra] ← (R[ra] ©ACC)%R[a];
mul	00011	ra	-	Y	R[ra] © ACC ← R[ra]*ACC;
and	01000	ra	-	Y	ACC ← ACC & R[ra];
or	01001	ra	-	Y	ACC ← ACC ~ R[ra];
not	01111	ra	-	Y	ACC ← !(R[ra]);
a2r	10000	ra	-	Y	R[ra] ← ACC
r2a	10110	ra		Y	ACC ← R[ra]
cla	00110			Z	ACC ← 0;
halt	11101	-	-	Z	RUN← 0;
nop	11110	-	-	Z	;
reset	11111	-	-	Z	TBD;

Symbol	Meaning	Symbol	Meaning
α	Replication	%	Remainder after integer division
©	Concatenation	&	Logical AND
:	Conditional constructs (IF-THEN)	~	Logical OR
;	Sequential constructs	!	Logical NOT or complement
,	Concurrent constructs	←	LOAD or assignment operator

Lecture No. 10

The FALCON-E and ISA Comparison

Reading Material

Handouts Slides

Summary

- The FALCON-E
- Instruction Set Architecture Comparison

THE FALCON-E

Introduction

FALCON stands for First Architecture for Learning Computer Organization and Networks. We are already familiar with our example processor, the FALCON-A, which was the first version of the FALCON processor. In this section we will develop a new version of the processor. Like its predecessor, the FALCON-E is a General-Purpose Register machine that is simple, yet is able to elucidate the fundamentals of computer design and architecture.

The FALCON-E is characterized by the following

- Eight General Purpose Registers (GPRs), named R0, R1…R7. Each registers is 4 bytes long (32-bit registers).
- Two special purposes registers, named BP and SP. These registers are also 32-bit in length.
- Two special registers, the Program Counter (PC) and the Instruction Register (IR). PC points to the next instruction to be executed, and the IR holds the current instruction.

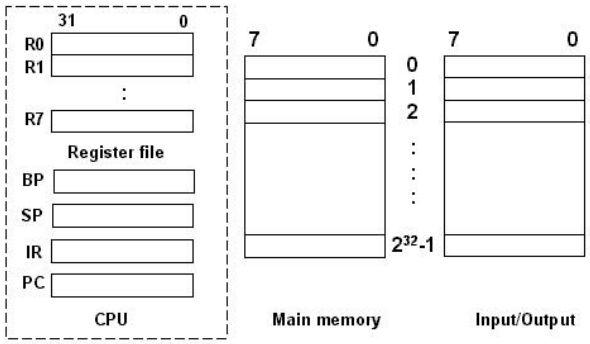


Fig. Programmer's View

- Memory word size is 32 bits (4 bytes).
- Memory space is 2^{32} bytes
- Memory is organized as 1-byte cells, and hence it is 2^{32} x 8 bits.
- Memory is accessed in 32-bit words (4-byte chunks, or 4 consecutive cells)
- Byte storage format is little endian.

Programmer's view of the FALCON-E

The programmer's view of the FALCON-E is shown in the given figure.

FALCON-E Notation

We take a brief look at the notation that we will employ for the FACLON-E.

Register contents are referred to in a similar fashion as the FALCON-A, i.e. the register name in square brackets. So R[3] means contents of register R3.

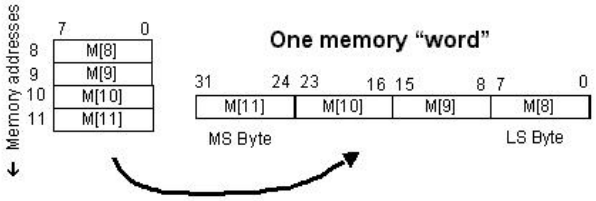


Fig. FALCON-E Notation

Memory contents (or the memory location) can be referred to in a similar way. Therefore, M[8] means contents of memory location 8.

A **memory word** is stored in the memory in the little endian format. This means that the least significant byte is stored first (or the little end comes first!). For instance, a memory word at address 8 is defined as the 32 bits at addresses 11, 10, 9, and 8 (little-endian). So we can employ a special notation to refer to the memory words. Again, we will employ © as the concatenation operator. In our notation for the FALCON-E, the memory word stored at address 8 is represented as:

M[8]<31…0>:=M[11]©M[10]©M[9]©M[8]

The shown figure will make this easier to understand.

FALCON-E Features

The following features characterize the FALCON-E

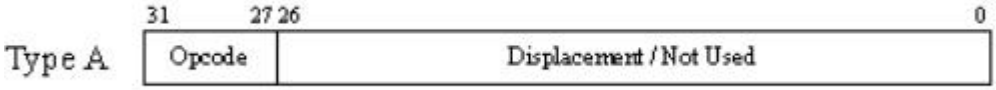
- Fixed instruction size, which is 32 bits. So the instruction size is 1 word.
- All ALU instructions have three operands
- Memory access is possible only through the load and store instructions. Also, only a limited addressing modes are supported by the FALCON-E

FALCON-E Instruction Formats

Four different instruction formats are supported by the FALCON-E. These are

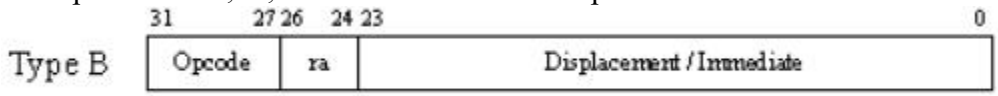
Type A instructions

The type A instructions have 5 bits reserved for the operation code (abbreviated op-code), and the rest of the bits are either not used or specify a displacement.



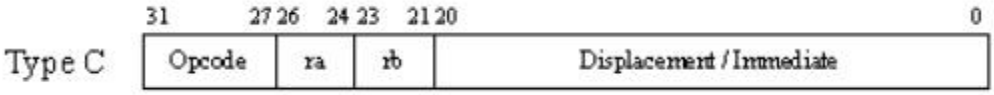
Type B instructions

The type B instructions also have 5 bits (27 through 31) reserved for the op-code. There is a register operand field, ra, and an immediate or displacement field in addition to the op-code field.



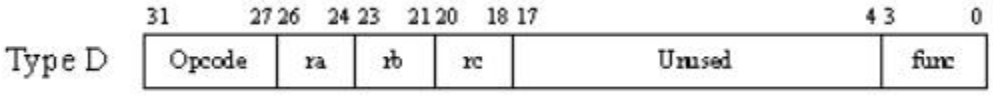
Type C instructions

Type C instructions have the 5-bit op-code field, two 3-bit operand registers (rb is the source register, ra is the destination register), a 17-bit immediate or displacement field, as well as a 3-bit function field. The function field is used to differentiate between instructions that may have the same op-code, but different operations.



Type D instructions

Type D instructions have the 5-bit op-code field, three 3-bit operand registers, 14 bits are unused, and a 3-bit function field.



Encoding for the General Purpose Registers (GPRs)

In the instruction formats discussed above, we used register operands ra, rb and rc. It is important to know that these are merely placeholders, and not the real register names. In an actual instruction, any one of the 8 registers of our general-purpose register file may be used. We need

to encode our registers so we can refer to them in an instruction. Note that we have reserved 3 bits for each of the register field. This is because we have 8 registers to represent, and they can be completely represented by 3 bits, since $2^3 = 8$. The following table shows the binary encoding of the general-purpose registers.

Register	Code	Register	Code
R0	000	R4	100
R1	001	R5	101
R2	010	R6	110
R3	011	R7	111

Fig. Encoding of the GPRs

There are two more special registers that we need to represent; the SP and the BP. We will use these registers in place of the operand register rb in the **load** and **store** instructions only, and therefore, we may encode these as

Register	Code
SP	000
BP	001

Fig. Special Registers Encoding

Instructions, Instruction Formats

The following is a brief introduction to the various instructions of the FALCON-E, categorized with respect to the instruction formats.

Type A instructions

Four instructions of the FALCON-E belong to type A. These are

- **nop** (op-code = 0)
  This instruction instructs the processor to do nothing. It is generally useful in pipelining. We will study more on pipelining later in the course.
- **ret** (op-code = 15)
  The return instruction is used to return control to the normal flow of a program after an interrupt or a procedure call concludes
- **iret** (op-code = 17)
  The **iret** instruction instructs the processor to return control to the address specified by the immediate field of the instruction. Setting the program counter to the specified address returns control.
- **near jmp** (op-code = 18)

A near jump is a PC-relative jump. The PC value is incremented (or decremented) by the immediate field value to take the jump.

Type B instructions

Five instructions belong to the type B format of instructions. These are:

- **push** (op-code = 8)
  This instruction is used to push the contents of a register onto the stack. For instance, the instruction,
  push R4
  will push the contents of register R4 on top of the stack
- **pop** (op-code = 9)
  The pop instruction is used to pop a value from the top of the stack, and the value is read into a register. For example, the instruction
  pop R7
  will pop the upper-most element of the stack and store the value in register R7
- **ld** (op-code = 10)
  This instruction with op-code (10) loads a memory word from the address specified by the immediate filed value. This word is brought into the operand register ra. For example, the instruction,
  ld R7, 1254h
  will load the contents of the memory at the address 1254h into the register R7.

- **st** (op-code = 12)
  The store instruction of (opcode 12) stores a value contained in the register operand into the memory location specified by the immediate operand field. For example, in
  st R7, 1254h
  the contents of register R7 are saved to the memory location 1254h.

Type C instructions

There are four data transfer instructions, as well as nine ALU instructions that belong to type C instruction format of the FALCON-E. The data transfer instructions are

- **lds** (op-code = 4)
  The load instruction with op-code (4)loads a register from the memory, after calculating the address of the memory location that is to be accessed. The effective address of the memory location to be read is calculated by adding the immediate value to the value stored by the register rb. For instance, in the example below, the immediate value 56 is added to the value stored by the register R4, and the resultant value is the address of the memory location which is read
  lds R3, R4(56)
  In RTL, this can be shown as
  R [3] ← M[R [4]+56]
- **sts** (op-code = 5)
  This instruction is used to store the register contents to the memory location, by first calculating the effective memory address. The address calculation is similar to the lds instruction. An example:
  sts R3, R4 (56)
  In RTL, this is shown as
  M[R [4]+56] ← R [3]
- **in** (op-code = 6)

This instruction is to load a register from an input/output device. The effective address of the I/O device has to be calculated before it is accessed to read the word into the destination register ra, as shown in the example:
in R5, R4(100)
In RTL:
R[5] ← IO[R[4]+100]

- **out** (op-code = 7)
  This instruction is used to write / store the register contents into an input/output device. Again, the effective address calculation has to be carried out to evaluate the destination I/O address before the write can take place. For example,
  out R8, R6 (36)
  RTL representation of this is IO[R [6]+36] ← R [8]

Three of the ALU instructions that belong to type C format are

- **addi** (op-code = 2)
  The addi instruction is to add a constant to the value of operand register rb, and assign the result to the destination register ra. For example, in the following instruction, 56 is added to the value of register R4, and result is assigned to the register R3.
  addi R3, R4, 56
  In RTL this can be shown as R[3] ← R[4]+56
  Note that if the immediate constant specified was a negative number, then this would become a subtract operation.
- **andi** (op-code = 2)
  This instruction is to calculate the logical AND of the immediate value and the rb register value. The result is assigned to destination register ra. For instance
  andi R3, R4, 56
  R[3] ← R[4]&56
  Note that the logical AND is represented by the symbol '&'
- **ori** (op-code = 2)
  This instruction calculates the logical OR of the immediate field and the value in operand register rb. The result is assigned to the destination register ra. Following
  is an example:
  ori R3, R4, 56
  The RTL representation of this instruction:
  R [3] ← R [4]~56
  Note that the symbol '~' is used to represent logical OR.

Type D Instructions

Four of the instructions that belong to this instruction format type are the ALU instructions shown below. There are other instructions of this type as well, listed in the tables at the end of this section.

- **add** (op-code = 1)
  This instruction is used to add two numbers. The numbers are stored in the registers specified by rb and rc. Result is stored into register ra. For instance, the instruction, add R3, R5, R6
  adds the numbers in register R5, R6, storing the result in R3. In RTL, this is given by R [3] ← R [5] + R [6]
- **sub** (op-code = 1)
  This instruction is used to carry out 2's complement subtraction. Again, register addressing mode is used, as shown in the example instruction

sub R3, R5, R6

RTL representation of this is R[3] ← R[5] - R[6]

- **and** (op-code = 1)

  For carrying out logical AND operation on the values stored in registers, this instruction is employed. For instance

  and R8, R3, R4

  In RTL, we can write this as R [8] ← R [3] & R [4]

- **or** (op-code = 1)

  For evaluating logical OR of values stored in two registers, we use this instruction. An example is

  or R8, R3, R4

  In RTL, this is

  R [8] ← R [3] ~ R [4]

Falcon-E Instruction Summary

The following are the tables that list the instructions that form the instruction set of the FALCON-E. These instructions have been grouped with respect to the functionality they provide.

Control Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
No Operation	nop	0	00000	-	

Fig. Control Instructions

Arithmetic Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Add	add	1	00001	0	0000
Add Immediate	addi	2	00010	0	0000
Subtract	sub	1	00001	1	0001
Subtract Immediate	subi	2	00010	1	0001
Multiply	mul	1	00001	2	0010
Multiply Immediate	muli	2	00010	2	0010
Divide	div	1	00001	3	0011
Divide Immediate	divi	2	00010	3	0011

Fig. Arithmetic Instructions

Data Transfer Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Move Immediate to GPR	movi	3	00011	-	
Load Special Purpose Register from GPR	lds	4	00100	-	
Store Special Purpose Register to GPR	sts	5	00101	-	
Load Register from IO	in	6	00110	-	
Store Register to IO	out	7	00111	-	
Push GPR to Stack	push	8	01000	-	
Pop GPR from Stack	pop	9	01001	-	
Load GPR from Memory (Direct Addressing)	ld	10	01010	-	
Load GPR from Memory (Displacement Addressing)	ld	11	01011	-	
Store GPR to Memory (Direct Addressing)	st	12	01100	-	
Store GPR to Memory (Displacement Addressing)	st	13	01101	-	

Fig. Data Transfer Instructions

Procedure Calls/Interrupts	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Call	call	14	01110	-	
Return	ret	15	01111	-	
Interrupt	int	16	10000	-	
Interrupt Return	iret	17	10001	-	

Fig. Procedure Calls/Interrupts

Branch Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Near Jump (Relative)	jmp	18	10010	-	
Far Jump (Direct)	jmp	19	10011	-	
Branch If Equal (Relative)	bre	20	10100	0	0000
Branch If Equal (Direct)	bre	21	10101	0	0000
Branch If Not Equal (Relative)	bne	20	10100	1	0001
Branch If Not Equal (Direct)	bne	21	10101	1	0001
Branch If Less (Relative)	bl	20	10100	2	0010
Branch If Less (Direct)	bl	21	10101	2	0010
Branch If Greater (Relative)	bg	20	10100	3	0011
Branch If Greater (Direct)	bg	21	10101	3	0011

Fig. Branch Instructions

Instruction Set Architecture Comparison

In this lecture, we compare the instruction set architectures of the various processors we have described/ designed up till now. These processors are:

- EAGLE
- FALCON-A
- FALCON-E
- SRC

Classifying Instruction Set Architectures

In the design of the ISA, the choice of some of the parameters can critically affect the code density (which is the number of instructions required to complete a given task), cycles per instruction (as some instructions may take more than one clock cycle, and the number of cycles per instruction varies from instruction to instruction, architecture to architecture), and cycle time (the total cycle time to execute a given piece of code). Classification of different architectures is based on the following parameters.

Operand storage in CPU	Where are they stored other than memory?
Number of explicit operands in an instruction	One, two or three operands?
Addressing Modes	How the effective address for operands is calculated?
Operations	What operation are possible and what are the choices for the opcodes?
Type and size of operands.	How the size is specified for operands?

Fig. ISA Comparison Parameters

Instruction Length

With reference to the instruction lengths in a particular ISA, there are two decisions to be made; whether the instruction will be fixed in length or variable, and what will be the instruction length or the range (in case of variable instruction lengths).

Fixed versus variable

Fixed instruction lengths are desirable when simplicity of design is a goal. It provides ease of implementation for assembling and pipelining. However, fixed instruction length can be wasteful in terms of code density. All the RISC machines use fixed instruction length format

Instruction Length

The required instruction length mainly depends on the number of instruction required to be in the instruction set of a processor (the greater the number of instructions supported, the more bits are required to encode the operation code), the size of the register file (greater the number of registers in the register file, more is the number of bits required to encode these in an instruction), the number of operands supported in instructions (as obviously, it will require more bits to encode a greater number of operands in an instruction), the size of immediate operand field (the greater the size, the more the range of values that can be specified by the immediate operand) and finally, the code density (which implies how many instructions can be encoded in a given number of bits). A summary of the instruction lengths of our processors is given in the table below.

EAGLE	FALCON-A	FALCON-E	SRC
Variable 8 bits or 16 bits	Fixed 16 bits	Fixed 32 bits	Fixed 32 bits

Fig. Instruction Length

Instruction types and sub-types

The given table summarizes the number of instruction types and sub-types of the processors we have studied. We have already studied these instruction types, and their sub-types in detail in the related sections.

	EAGLE	FALCON-A	FALCON-E	SRC
Types	4	4	4	4
Sub-types	-	2	4	3

Number of operands in the instructions

The number of operands that may be required in an instruction depends on the type of operation to be performed by that instruction; some instruction may have no operands, other may have up to 3. But a limit on the maximum number of operands for the instruction set of a processor needs to be defined explicitly, as it affects the instruction length and code density. The maximum number of operands supported by the instruction set of each processor under study is given in the given table. So FALCON-A, FALCON-E and the SRC processors may have 3, 2, 1 or no operands, depending on the instruction. EAGLE has a maximum number of 2 operands; it may have one operand or no operands in an instruction.

EAGLE	FALCON-A	FALCON-E	SRC
2	3	3	3

Fig. Number of Operands per instructions

Explicit operand specification in an instruction gives flexibility in storage. Implicit operands like an accumulator or a stack reduces the instruction size, as they need not be coded into the instruction. Instructions of the processor EAGLE have implicit operands, and we saw that the result is automatically stored in the accumulator, without the accumulator being specified as a destination operand in the instruction.

Number and Size of General Purpose Registers

While designing a processor, another decision that has to be made is about the number of registers present in the register file, and the size of the registers.

Increasing the number of registers in the register file of the CPU will decrease the memory traffic, which is a desirable attribute, as memory accesses take relatively much longer time than register access. Memory traffic decreases as the number of registers is increased, as variables are copied into the registers and these do not have to be accessed from memory over and over again. If there is a small number of registers, the values stored previously will have to be saved back to memory to bring in the new values; more registers will solve the problem of swapping in, swapping out. However, a very large register file is not feasible, as it will require more bits of the instruction to encode these registers. The size of the registers affects the range of values that can be stored in the registers.

The number of registers in the register file, along with the size of the registers, for each of the processors under study, is in the given table.

EAGLE	FALCON-A	FALCON-E	SRC
Eight registers, 16 bit wide	Eight registers, 16 bit wide	Eight registers, 32 bit wide	Thirty-two registers 32 bit wide

Fig. Number and size of GPRS

Memory specifications

Memory design is an integral part of the processor design. We need to decide on the memory space that will be available to the processor, how the memory will be organized, memory word size, memory access bus width, and the storage format used to store words in memory. The memory specifications for the processor under comparison are:

Memory Specs.	EAGLE	FALCON-A	FALCON-E	SRC
Memory Space	2^{16}	2^{16}	2^{32}	2^{32}
Memory Organization	$2^{16} * 8$	$2^{16} * 8$	$2^{32} * 8$	$2^{32} * 8$
Memory Word Size	16 bit	16 bit	32 bit	32 bit
Memory Access	16 bits	16 bits	32 bits	32 bits
Memory Storage	Little-Endian	Big Endian	Little-Endian	Big Endian

Fig. Memory Specifications

Data transfer instructions

Data needs to be transferred between storage devices for processing. Data transfers may include loading, storing back or copying of the data. The different ways in which data transfers may take place have their related advantages and disadvantages. These are listed in the given table.

Data Transfer	Advantage	Disadvantage
Register to Register	Simple, faster, constant CPI, Easier to pipeline.	Higher instruction count, longer program codes
Register to Memory	Separate load instruction eliminated, good code density	Variable CPI due to different operand locations
Memory to Memory	Most compact, small number of registers required	Variable CPI, variable instruction size, memory bottleneck.

Fig. Data Transfer Modes

Following are the data transfer instructions included in the instruction sets of our processors.

Register to register transfers

As we can see from the given table on the next page, in the processor EAGLE, register to register transfers are of two types only: register to accumulator, or accumulator to register. Accumulator is a special-purpose register.

FALCON-A has a **mov** instruction, which can be used to move data of any register to any other register. FALCON-E has the instructions 'lds' and 'sts' which are used to load/store a register from/to memory after effective address calculation.

SRC does not provide any instruction for data movement between general-purpose registers. However, this can be accomplished indirectly, by adopting either of the following two approaches:

- A register's contents can be loaded into another register via memory. First storing the content of a register to a particular memory location, and then reading the contents of the memory from that location into the register we want to copy the value to can achieve this. However, this method is very inefficient, as it requires memory accesses, which are inherently slow operations.
- A better method is to use the addi instruction with the constant set to 0.

Data Transfer Instructions

Instructions	EAGLE	FALCON-A	FALCON-E	SRC
Register to Register	a2r, r2a	mov	lds, sts	lar (only from PC)
Register to Memory	ldacc, stacc	load, store	ld, st	ld, st
Memory to Memory	-	-	-	-

Register to memory

EAGLE has instructions to load values from memory to the special purpose register, names the accumulator, as well as saving values from the accumulator to memory. Other register to memory transfers is not possible in the EAGLE processor. FALCON-A, FALOCN-E and the SRC have simple load, store instructions and all register-memory transfers are supported.

Memory to memory

In any of the processors under study, memory-to-memory transfers are not supported.
However, in other processors, these may be a possibility.

Control Flow Instructions

All processors have instructions to control the flow of programs in execution. The general control flow instructions available in most processors are:

- Branches (conditional)
- Jumps (unconditional)
- Calls (procedure calls)
- Returns (procedure returns)

Conditional Branches

Whereas jumps, calls and call returns changes the control flow in a specific order, branches depend on some conditions; if the conditions are met, the branch may be taken, otherwise the program flow may continue linearly. The branch conditions may be specified by any of the following methods:

- Condition codes
- Condition register
- Comparison and branching

Condition codes

The ALU may contain some special bits (also called flags), which may have been set (or raised) under some special circumstances. For instance, a flag may be raised if there is an overflow in the addition results of two register values, or if a number is negative. An instruction can then be ordered in the program that may change the flow depending on any of these flag's values. The EAGLE processor uses these condition codes for branch condition evaluation.

Condition register

A special register is required to act as a branch register, and any other arbitrary register (that is specified in the branch instruction), is compared against that register, and the branching decision is based on the comparison result of these two registers. None of the processors under our study use this mode of conditional branching.

Compare and branch

In this mode of conditional branching, comparison is made part of the branching instruction. Therefore, it is somewhat more complex than the other two modes. All the processors we are studying use this mode of conditional branching.

Size of jumps

Jumps are deviations from the linear program flow by a specified constant. All our processors, except the SRC, support PC-relative jumps. The displacement (or the jump) relative to the PC is specified by the constant field in the instruction. If the constant field is wider (i.e. there are more

bits reserved for the constant field in the instruction), the jump can be of a larger magnitude. Shown table specifies the displacement size for various processors.

Processor	Displacement size
EAGLE	8 bits for both conditional and unconditional.
FALCON-A	8 bits for both conditional and unconditional.
FALCON-E	27 bits (unconditional jump), 21 or 32 bits (conditional jumps)
SRC	32 bits for both conditional and unconditional jumps.

Fig. Size of Jumps

Addressing Modes

All processors support a variety of addressing modes. An addressing mode is the method by which architectures specify the address of an object they will access. The object may be a constant, a register or a location in memory.
Common addressing modes are

- **Immediate**
  An immediate field may be provided in instructions, and a constant value may be given in this immediate field, e.g. **123** is an immediate value.
- **Register**
  A register may contain the value we refer to in an instruction, for instance, register **R4** may contain the value being referred to.
- **Direct**
  By direct addressing mode, we mean the constant field may specify the location of the memory we want to refer to. For instance, **[123]** will directly refer to the memory location 123's contents.
- **Register Indirect**
  A register may contain the address of memory location to which we want to refer to, for example, **M [R3]**.
- **Displacement**
  In this addressing mode, the constant value specified by the immediate field is added to the register value, and the resultant is the index of memory location that is referred to, e.g. **M [R3+123]**
- **Relative**
  Relative addressing mode implies PC-relative addressing, for example, **[PC+123]** will refer to the memory location that is 123 words farther than the memory index currently stored in the program counter.
- **Indexed or scaled**
  The values contained in two registers are added and the resultant value is the index to the memory location we refer to, in the indexed addressing mode. For example, **M [[R1]+[R2]]**. In the scaled addressing mode, a register value may be scaled as it is added to the value of the other register to obtain the index of memory location to be referred to.
- **Auto increment/ decrement**
  In the auto increment mode, the value held in a register is used as the index to memory location that holds the value of operand. After the operand's value is retrieved, the

register value is automatically increased by 1 (or by any specified constant). e.g. **M [R4]+**, or **M [R4]+d**. In the auto decrement mode, the register value is first decremented and then used as a reference to the memory location that referred to in the instruction, e.g. **-M [R4].**

As may be obvious to the reader, some of these addressing modes are quite simple, others are relatively complex. The complex addressing modes (such as the indexed) reduce the instruction count (thus improving code density), at the cost of more complex implementation.

The given table lists the addressing modes supported by the processors we are studying. Note that the register-addressing mode is a special case of the relative addressing mode, with the constant equal to 0, and only the PC can be used as a source. Also note that, in the shown table, relative implies PC-relative.

EAGLE	FALCON-A	FALCON-E	SRC
Immediate	Immediate	Immediate	Immediate
-	-	Direct	Direct
Register	Register	Register	Register *
Register Indirect	Register Indirect	Register Indirect	Register Indirect
-	-	-	Relative**
Displacement	Displacement	Displacement	Displacement

Fig. Addressing Modes Comparison

Displacement addressing mode

We have already talked about the displacement-addressing mode. We look at this addressing mode at length now.

The displacement-addressing mode is the most common of the addressing mode used in general

Size of displacement field

Processor	Number of bits in displacement field
SRC	17 or 22 bits depending on the instruction type.
FALCON-E	21 or 24 bits depending on the instruction type.
FALCON-A	5 bits for load and store instruction
EAGLE	8 bits for ldacc and stacc instructions

purpose processors. Some other modes such as the indexed based plus index, scaled and register indirect are all slightly modified forms of the displacement-addressing mode. The size of

displacement plays a key role in efficient address calculation. The following table specifies the size of the displacement field in different processors under study. The given table lists the size of the immediate field in our processors.

Processor	Number of bits in the immediate field
EAGLE	8 bits
FALCON-A	5 or 8 bits
FALCON-E	17 or 24 bits depending on the instruction
SRC	17 or 22 bits

Fig. Immediate Field Bits Comparison

Instructions common to all Instruction Set Architectures

In this section we have listed the instructions that are common to the Instruction Set Architectures of all the processors under our study.

- **Arithmetic Instructions**
  add, addi & sub.
- **Logic Instructions**
  and, andi, or, ori, not.
- **Shift Instructions.**
  Right shift, left shift & arithmetic right shift.
- **Data movement Instructions.**
  Load and store instructions.
- **Control Instructions**
  Conditional and unconditional branches, nop & reset.

The following tables list the assembly language instruction codes of these common instructions for all the processors under comparison.

Common Arithmetic Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Add	add	add	add	add
Add Immediate	addi	addi	addi	addi
Subtract	sub	sub	sub	sub
Subtract Immediate	subi	subi	subi	-
Multiply	mul	mul	mul	-
Divide	div	div	div	-

Common data movement Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Load	ldacc	load	ld	ld
Store	stacc	store	st	st
Move	mov	mov	-	-
Move immediate	movi	movi	movi	la
In	in	in	in	-
Out	out	out	out	-

Common Logical Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
And	and	and	and	and
And Immediate	andi	andi	andi	andi
Or	or	or	or	or
Or Immediate	ori	ori	ori	ori
Not	not	not	not	not
Neg	neg	neg	-	-

Common Shift Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Shift right	shiftr	shiftr	-	shr
Shift right immediate	-	-	srai	shr
Circular shift	-	-	rol	shc
Shift left	shiftl	shiftl	-	shl
Shift right arithmetic	asr	asr	sra	shra

Common Branch Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Unconditional branch	br	jump	jmp	br
Branch if zero	brz	jz	-	brzr
Branch if non zero	brnz	jnz	-	brnz
Branch if positive	brp	jpl	-	brpl
Branch if negative	brn	jmi	-	brmi

Common Call and Interrupt Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Procedure call	-	call	call	brl
Interrupt	-	int	int	?
Interrupt return	-	iret	iret	?

Common Control Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
No operation	nop	nop	nop	nop
Halt	halt	halt	-	stop
Reset	reset	reset	-	-

Instructions unique to each processor

Now we take a look at the instructions that are unique to each of the processors we are studying.

EAGLE

The EAGLE processor has a minimal instruction set. Following are the instructions that are unique only to the EAGLE processor. Note that these instructions are unique only with reference to the processor set under our study; some other processors may have these instructions.

- movia
  This instruction is for moving the immediate value to the accumulator (the special purpose register)
- a2r
  This instruction is for moving the contents of the accumulator to a register
- r2a
  For moving register contents to the accumulator
- cla
  For clearing (setting to zero) the value in the accumulator

FALCON-A

There is only one instruction unique to the FALCON-A processor;

- ret
  This instruction is used to return control to a calling procedure. The calling procedure may save the PC value in a register ra, and when this instruction is called, the PC value is restored. In RTL, we write this as
  PC R [ra];

FALCON-E

The instructions unique to the FALCON-E processor are listed:

- push
  To push the contents of a specified general purpose register to the stack
- pop
  To pop the value that is at the top of the stack
- ldr
  To load a register with memory contents using displacement addressing mode
- str
  To store a register value into memory, using displacement addressing mode
- bl
  To branch if source operand is less than target address
- bg
  To branch if source operand is greater than target address
- muli
  To multiply an immediate value with a value stored in a register
- divi
  To divide a register value by the immediate value
- xor, xori
  To evaluate logical ‘exclusive or’
- ror, rori

SRC

Following are the instructions that are unique to the SRC processor, among of the processors under study

- ldr

To load register from memory using PC-relative address

- lar
  To load a register with a word from memory using relative address
- str
  To store register value to memory using relative address
- brlnv
  This instruction is to tell the processor to 'never branch' at that point in program. The instruction saves the program counter's contents to the register specified
- brlpl
  This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register's value is positive. Return address is saved before branching.
- brlmi
  This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register's value is negative. Return address is saved before branching.
- brlzr
  This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register's value equals zero. Return address is saved before branching.
- brlnz
  This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register's value does not equal zero. Return address is saved before branching.

Problem Comparison

Given is the code for a simple C statement:

a=(b-2)+4c

The given table gives its implementation in all the four processors under comparison. Note that this table highlights the code density for each of the processors; EAGLE, which has relatively fewer specialized instructions, and so it takes more instructions to carry out this operation as compared with the rest of the processors

EAGLE	FALCON-A	FALCON-E	SRC
.org 100 a: .dw 1	.org 100 a: .dw 1	.org 100 a: .dw 1	.org 100 a: .dw 1

.org 200
ldacc b
a2r r1
subi r1,2
a2r r1
ldacc c
a2r r2
shl r2, 2
r2a r2
add r1
stacc a
	.org 200
	load r1, b
	subi r2, r1, 2
	load r3, c
	shiftl r3,r3,2
	add r4,r2,r3
	store r4, a
		.org 200
		ld r1, b
		subi r2, r1,2
		ld r3, c
		muli r3,r3, 4
		add r4, r3,r2
		store r4,a
			.org 200
			ld r1, b
			addi r2,r1,-2
			ld r3, c
			shl r3, r3, 2
			add r4,r2,r3
			st r4, a

Fig. Problem Comparison

Lecture No. 11

CISC and RISC

Reading Material

Vincent P. Heuring & Harry F. Jordan	Chapter 3
Computer Systems Design and Architecture	3.3, 3.4

Summary

- A CISC microprocessor:The Motorola MC68000
- A RISC Architecture:The SPARC

Material of this Lecture is included in the above-mentioned sections of Chapter 3.

Lecture No. 12

CPU Design

Reading Material

Vincent P. Heuring & Harry F. Jordan	Chapter 4
Computer Systems Design and Architecture	4.1, 4.2, 4.3

Summary

- The design process
- A Uni-Bus implementation for the SRC
- Structural RTL for the SRC instructions

Central Processing Unit Design

This module will explore the design of the central processing unit from the logic designer's view. A unibus implementation of the SRC is discussed in detail along with the Data Path Design and the Control Unit Design. The topics covered in this module are outlined below:

- The Design Process
- Unibus Implementation of the SRC
- Structural RTL for the SRC
- Logic Design for one bus SRC
- The Control Unit
- 2-bus and 3-bus designs
- The machine reset
- The machine exceptions

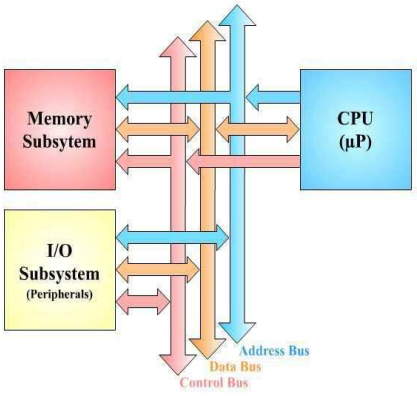
As we progress through this list of topics, we will learn how to convert the earlier specified behavioral RTL into a concrete structural RTL. We will also learn how to interconnect various programmer visible registers to get a complete data path and how to incorporate various control signals into it. Finally, we will add the machine reset and exception capability to our processor.

The design process

The design process of a processor starts with the specification of the behavioral RTL for its instruction set. This abstract description is then converted into structural RTL which shows the actual implementation details. Since the processor can be divided into two main sub-systems, the data path and the control unit, we can split the design procedure into two phases.

1. The data path design
2. The control unit design

It is important that the design activity of these important components of the processor be carried out with the pros and cons of adopting different approaches in mind.



Block Diagram of Computer System

As we know, the execution time is dependent on the following three factors. ET = IC x CPI x T

During the design procedure we specify the implementation details at an advanced level. These details can affect the clock cycle per instruction and the clock cycle time. Hence following things should be kept in mind during the design phase.

- Effect on overall performance
- Amount of control hardware
- Development time

Processor Design

Let us take a look at the steps involved in the processor design procedure.

1. ISA Design

The first step in designing a processor is the specification of the instruction set of the processor. ISA design includes decisions involving number and size of instructions, formats, addressing modes, memory organization and the programmer's view of the CPU i.e. the number and size of general and special purpose registers.

2. Behavioral RTL Description

In this step, the behavior of processor in response to the specific instructions is described in register transfer language. This abstract description is not bound to any specific implementation of the processor. It presents only those static (registers) and dynamic aspects (operations) of the machine that are necessary to understand its functionality. The unit of activity here is the instruction execution unlike the clock cycle in actual case. The functionality of all the instructions is described here in special register transfer notation.

3. Implementation of the Data Path

The data path design involves decisions like the placement and interconnection of various registers, the type of flip-flops to be used and the number and kind of the interconnection buses. All these decisions affect the number and speed of register transfers during an operation. The structure of the ALU and the design of the memory-to-CPU interface also need to be decided at this stage. Then there are the control signals that form the interface between the data path and the control unit. These control signals move data onto buses, enable and disable flip-flops, specify the ALU functions and control the buses and memory operations. Hence an integral part of the data path design is the seamless embedding of the control signals into it.

4. Structural RTL Description

In accordance with the chosen data path implementation, the structural RTL for every instruction is described in this step. The structural RTL is formed according to the proposed micro-architecture which includes many hidden temporary registers necessary for instruction execution. Since the structural RTL shows the actual implementation steps, it should satisfy the time and space requirements of the CPU as specified by the clocking interval and the number of registers and buses in the data path.

5. Control Unit Design

The control unit design is a rather tricky process as it involves timing and synchronization issues besides the usual combinational logic used in the data path design. Additionally, there are two different approaches to the control unit design; it can be either hard-wired or micro-programmed. However, the task can be made simpler by dividing the design procedure into smaller steps as follows.

a) Analyze the structural RTL and prepare a list of control signals to be activated during the execution of each RTL statement.
b) Develop logic circuits necessary to generate the control signals
c) Tie everything together to complete the design of the control unit.

Processor Design

A Uni-bus Data Path Implementation for the SRC

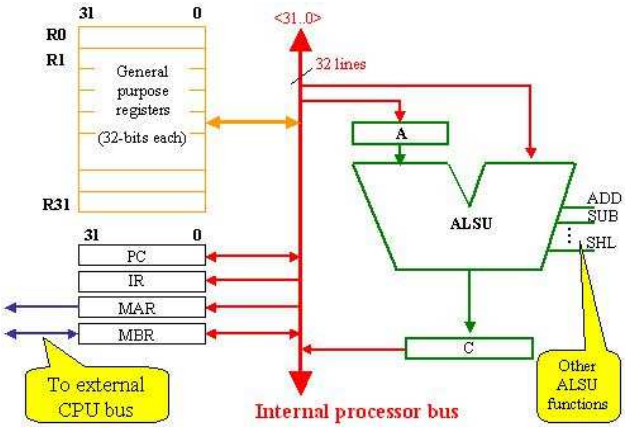
In this section, we will discuss the uni-bus implementation of the data path for the SRC. But before we go onto the design phase, we will discuss what a data path is. After the discussion of the data path design, we will discuss the timing step generation, which makes possible the synchronization of the data path functions.

The Data Path

The data path is the arithmetic portion of the Von Neumann architecture. It consists of registers, internal buses, arithmetic units and shifters. We have already discussed the decisions involved in designing the data path. Now we shall have an overview of the 1-Bus SRC data path design. As the name suggests, this implementation employs a single bus for data flow. After that we develop each of its blocks in greater detail and present the gate level implementation.

Overview of the Unibus SRC Data Path

The 1-bus implementation of the SRC data path is shown in the figure given. The control signals are omitted here for the sake simplicity. Following units are present in the SRC data path.



of

1. The Register File

The general-purpose register file includes 32 registers R0 to R31 each 32 bit wide. These registers communicate with other components via the internal processor bus.

2. MAR

The Memory Address Register takes input from the ALSU as the address of the memory location to be accessed and transfers the memory contents on that location onto the memory sub-system.

3. MBR

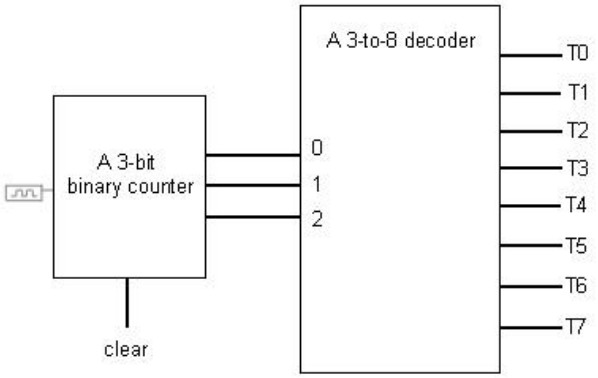
The Memory Buffer Register has a bi-directional connection with both the memory sub-system and the registers and ALSU. It holds the data during its transmission to and from memory.

4. PC

The Program Counter holds the address of the next instruction to be executed. Its value is incremented after loading of each instruction. The value in PC can also be changed based on a branch decision in ALSU. Therefore, it has a bi-directional connection with the internal processor bus.

5. IR

The Instruction Register holds the instruction that is being executed. The instruction fields are extracted from the IR and transferred to the appropriate registers according to the external circuitry (not shown in this diagram).



6. Registers A and C

The registers A and C are required to hold an operand or result value while the bus is busy transmitting some other value. Both these registers are programmer invisible.

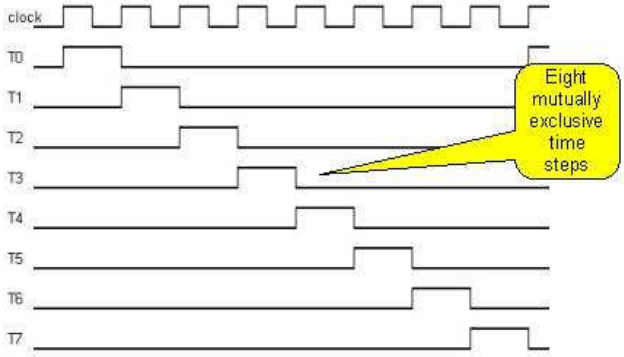
7. ALSU

There is a 32-bit Arithmetic Logic Shift Unit, as shown in the diagram. It takes input from memory or registers via the bus, computes the result according to the control signals applied to it, and places it in the register C, from where it is finally transferred to its destination.

Timing Step Generator To ensure the correct and controlled execution of instructions in a program, and all the related operations, a timing device is required. This is to ensure that the operations of essentially different instructions do not mix up in time. There exists a 'timing step generator' that provides mutually exclusive and sequential timing intervals. This is analogous to the clock cycles in the actual processor. A possible implementation of the timing step generator is shown in the figure.

Each mutually exclusive step is carried out in one timing interval. The timing intervals can be named T0, T1…T7. The given figure is helpful in understanding the 'mutual exclusiveness in time' of these timing intervals.

Processor design



Structural RTL descriptions of selected SRC instructions

Structural RTL for the SRC

The structural RTL describes how a particular operation is performed using a specific hardware implementation. In order to present the structural RTL we assume that there exists a "timing step generator", which provides mutually exclusive and sequential timing intervals, analogous to the clock cycles in actual processor.

Structural RTL for Instruction Fetch

	Step	RTL
Instruction Fetch	T0	MAR ← PC, C ← PC + 4;
	T1	MBR ← M[MAR], PC ← C;
	T2	IR ← MBR;

The instruction fetch procedure takes three time steps as shown in the table. During the first time step, T0, address of the instruction is moved to the Memory Address Register (MAR) and value of PC is incremented. In T1 the instruction is brought from the memory into the Memory Buffer Register (MBR), and the incremented PC is updated. In the third and final time-step of the instruction fetch phase, the instruction from the memory buffer register is written into the IR for execution. What follows the instruction fetch phase, is the instruction execution phase. The number of timing steps taken by the execution phase generally depends on the type and function of instruction. The more complex the instruction and its implementation, the more timing steps it will require to complete execution. In the following discussion, we will take a look at various types of instructions, related timing steps requirements and data path implementations of these in terms of the structural RTL.

Structural RTL for Arithmetic/Logic Instructions

The arithmetic/logic instructions come in two formats, one with the immediate operand and the other with register operand. Examples of both are shown in the following tables.

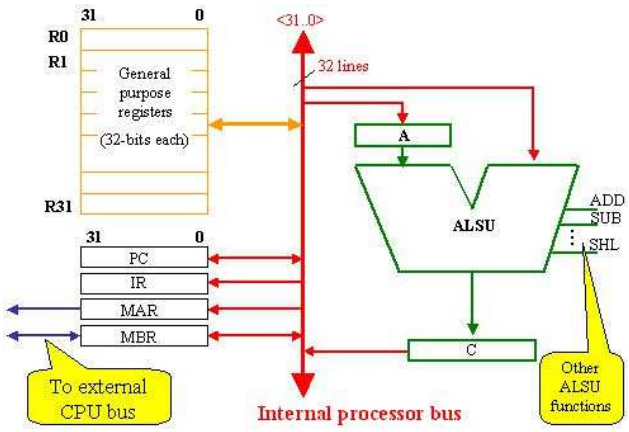
Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A - R[rc];
T5	R[ra] ← C;

Register-to-Register sub

Register-to-register subtract (or sub) will take three timing steps to complete execution, as shown in the table. Here we have assumed that the instruction given is:

sub ra, rb, rc

Here we assume that the instruction fetch process has taken up the first three timing steps. In step T3 the internal register A receives the contents of the register rb. In the next timing step, the value of register rc is subtracted (since the op-code is sub) from A. In the final step, this result is transferred into the destination register ra. This concludes the instruction fetch-execute cycle and at the end of it, the timing step generator is initialized to T0. The given figure refreshes our knowledge of the data path. Notice that we can visualize how the steps that we have just outlined can be carried out, if appropriate control signals are applied at the appropriate timing.



As will be obvious, control signals need to be applied to the ALSU, based on the decoding of the op-code field of an instruction. The given table lists these control signals:
Note that we have used uppercase alphabets for naming the ALSU functions. This is to differentiate these control signals from the actual operation-code mnemonics we have been using for the instructions.

The SHL, SHR, SHC and the SHRA functions are listed assuming that a barrel shifter is available to the processor with signals to differentiate between the various types of shifts that are to be performed.

ALSU Function	Needed for the following instructions/operations
ADD	add, addi, address calculation for disp and rel
SUB	sub
NEG	neg; applies to the B input of the ALSU
AND	and, andi
OR	or, ori
NOT	not; applies to the B input of the ALSU
SHL	shl
SHR	shr
SHC	shc
SHRA	shra
C=B	to load from the bus directly into C
INC4	to increment the PC by 4; applies to the B input;

assuming a barrel shifter with five n<4..0> signals available as well

Use uppercase for control signals, because lowercase was used for mnemonics

Structural RTL for Register-to-Register add

To enhance our understanding of the instruction execution phase implementation, we will now take a look at some more instructions of the SRC. The structural RTL for a simple add instruction **add ra, rb, rc** is given in table. The first three instruction fetch steps are common to all instructions. Execution of instruction starts from step T3 where the first operand is moved to register A. The second step involves computation of the sum and result is transferred to the destination in step T5. Hence the complete execution of the add instruction takes 6 time steps. Other arithmetic/logic instructions having the similar structural RTL are "**sub", "and"** and **"or".** The only difference is in the T4 step where the sign changes to (-), (^), or (~) according to the opcode.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A + R[rc];
T5	R[ra] ← C;

Structural RTL for the not instruction

The first three steps T0 to T2 are used up in fetching the instruction as usual. In step T3, the value of the operand specified by the register is brought into the ALSU, which will use the control function NOT, negate the value (i.e. invert it), and the result moves to the register C. In the time step R4, this result is assigned to the destination register through the internal bus. Note that we need control signals to coordinate all of this; a control signal to allow reading of the instruction-specified source register in T3, control signal for the selection of appropriate function to be

Step	RTL
T0-T2	Instruction fetch
T3	C ← !(R[rb]);
T4	R[ra] ← C;

carried out at the ALSU, and control signal to allow only the instruction-specified destination register to read the result value from the data bus.

The table shown outlines these steps for the instruction: **not ra, rb**

Structural RTL for the addi instruction

Again, the first three time steps are for the instruction fetch. Next, the first operand is brought into ALSU in step T3 through register A. The step T4 is of interest here as the second operand c2 is extracted from the instruction in IR register, sign extended to 32 bits, added to the first operand and written into the result register C. The execution of instruction completes in step T5 when the result is written into the destination register. The sign extension is assumed to be carried out in the ALSU as no separate extension unit is provided.

Sign extension for 17-bit c2 is the same as:(15αIR<16> ©IR<16..0>)

Sign extension for 22-bit c1 is the same as:(10αIR<21> ©IR<21..0>)

The given table outlines the time steps for the instruction addi:

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A + c2(sign extend);

Other instructions that have the same structural RTL are **subi**, **andi** and **ori.**

RTL for the load (ld) and store (st) instructions

The syntax of load instructions is:

ld ra, c2(rb)

And the syntax of store instructions is:

st ra, c2(rb)

Step	RTL for ld	RTL for st
T0-T2	Instruction fetch	Instruction fetch
T3	A ← ((rb = 0) : 0, (rb ≠ 0): R[rb]);	A ← ((rb = 0): 0, (rb ≠ 0): R[rb]);
T4	C ← A + (15αIR<16> ©IR<16..0>);	C ← A + (15αIR<16> ©IR<16..0>);
T5	MAR ← C;	MAR ← C;
T6	MBR ← M[MAR];	MBR ← R [ra];
T7	R[ra] ← MBR;	M[MAR] ← MBR;

sign extension

The given table outlines the time steps in fetching and executing a load and a store instruction. Note that the first 6 time steps (T0 to T5) for both the instructions are the same.

The first three steps are those of instruction fetch. Next, the register A gets the value of register rb, in case it is not zero. In time step T4, the constant is sign-extended, and added to the value of register A using the ALSU. The result is assigned to register C. Note that in the RTL outlined above; we are sign extending a field of the Instruction Register (32-bit). It is so because this field is the constant field in the instruction, and the Instruction Register holds the instruction in execution. In step T5, the value in C is transferred to the Memory Address Register (MAR). This completes the effective address calculation of the memory location to be accessed for the load/ store operation. If it is a load instruction in time step T6, the corresponding memory location is accessed and result is stored in Memory Buffer Register (MBR). In step T7, the result is transferred to the destination register ra using the data bus. If the instruction is to store the value of a register, the time step T6 is used to store the value of the register to the MBR. In the next and final step, the value stored in MBR is stored in the memory location indexed by the MAR. We can look at the data-path figure and visualize how all these steps can take place by applying appropriate control signals. Note that, if more time steps are required, then a counter with more bits and a larger decoder can be used, e.g., a 4-bit counter along with a 4-to-16 decoder can produce up to 16 time steps.

Lecture No. 13

Structural RTL Description of the FALCON-A

Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture — Chapter 4, 4.2.2, slides

Summary

- Structural RTL Description of the SRC (continued…)
- Structural RTL Description of the FALCON-A

This lecture is a continuation of the previous lecture.

Structural RTL for branch instructions

Let us take a look at the structural RTL for branch instructions. We know that there are several variations of the branch instructions including unconditional branch and different conditional branches. We look at the RTL for 'branch if zero' (brzr) and 'branch and link if zero' brlzr' conditional branches.

The syntax for the branch if zero (brzr) is:

brzr rb, rc

As you may recall, this instruction instructs the processor to branch to the instruction at the address held in register rb, if the value stored in register rc is zero. Time steps for this instruction are outlined in the table.

Step	RTL
T0-T2	Instruction Fetch
T3	CON ← cond(R[rc]);
T4	CON: PC ← R[rb];

The first three steps are of the instruction fetch phase. Next, the value of register rc is checked and depending on the result, the condition flag CON is set. In time step T4, the program counter is set to the register rb value, depending on the CON bit (the condition flag). The syntax for the branch and link if zero (brlzr) is:

brlzr ra, rb, rc

This instruction is the same as the instruction **brzr** but additionally the return address is saved (linking procedure). The time steps for this instruction are shown in the table.

Step	RTL
T0-T2	Instruction Fetch
T3	CON ← cond(R[rc]);
T4	CON: R[ra] ← PC;
T5	CON: PC ← R[rb];

Notice that the steps for this instruction are the same as the instruction brzr with an additional step after the condition bit is set; the current value of the program counter is saved to register ra.

Structural RTL for shift instructions

Shift instructions are rather complicated in the sense that they require extra hardware to hold and decrement the count. For an ALSU that can perform only single bit shifts, the data must be repeatedly cycled through the ALSU and the count decremented until it reaches zero. This approach presents some timing problems, which can be overcome by employing multiple-bit shifts using a barrel shifter.

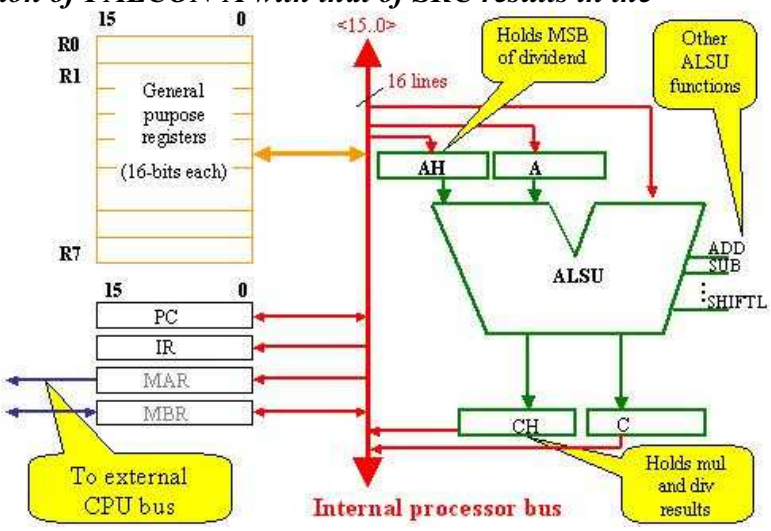
Step	RTL
T0-T2	Instruction fetch
T3	n<4..0> ← IR<4..0>;
T4	(N = 0) : (n<4..0> ← R[rc]<4..0>);
T5	C ← (Nα0) © R[rb]<31..N>;
T6	R[ra] ← C;

The structural RTL for **shr ra, rb, rc** or **shr ra, rb, c3** is given in the corresponding table shown. Here n represents a 5-bit register; IR bits 0 to 4 are copied in to it. N is the decimal value of the number in this register. The actual shifting is being done in step T5. Other instructions that will have similar tables are**: shl, shc, shra** e.g., for shra, T5 will have C← (NαR [rb] <31>) © R[rb] <31...N>;

Structural RTL Description of FALCON-A Instructions

Uni-bus data path implementation

Comparing the uni-bus implementation of FALCON-A with that of SRC results in the following differences:



- ***FALCON-A processor bus has 16 lines or is 16-bits wide while that of SRC is 32-bits wide.***
- All registers of FALCON-A are of 16-bits while in case of SRC all registers are 32-bits.
- Number of registers in FALCON-A are 8 while in SRC the number of registers is 32.
- Special registers i.e. Program Counter (PC) and Instruction Register (IR) are 16-bit registers while
- in SRC these are 32-bits.
- Memory Address Register (MAR) and Memory Buffer Register (MBR) are also of 16-bits while in SRC these are of 32-bits.
- MAR and MBR are dual port registers. At one side they are connected to internal bus and at other side to external memory in order to point to a particular address for reading or writing data from or to the memory and MBR would get the data from the memory.

ALSU functions needed

ALSU of FALCON-A has slightly different functions. These functions are given in the table.

ALSU Function	Needed for the following instructions/operations
ADD	add, addi
SUB	sub, subi
MUL	mul
DIV	div
AND	and, andi
OR	or, ori
NOT	not; applies to the B input of the ALSU
SHIFTL	shiftl
SHIFTR	shiftr
ASR	asr
C=B	to load from the bus directly into C
INC2	to increment the PC by 2; applies to the B input;

assuming a barrel shifter with five n<4..0> signals available as well

Note that **mul** and **div** are two significant instructions in this instruction set. So whenever one of these instructions is activated, the ALSU unit would take the operand from its input and provide the output immediately, if we neglect the propagation delays to its output. In case of FALCON-A, we have two registers A and AH each of 16-bits. AH would contain the higher 16-bits or most significant 16-bits of a 32-bit operand. This means that the ALSU provides the facility of using 32-bit operand in certain instructions. At the output of ALSU we could have a 32-bit result and that cannot be saved in just one register C so we need to have another one that is CH. CH can store the most significant 16-bits of the result.

Why do we need to add AH and CH?

This is because we have mul and div instructions in the instruction set of the FALCON-A. So for that case, we can implement the div instruction in which, at the input, one of the operand which is dividend would be 32-bits or in case of mul instruction the output which is the result of multiplication of two 16-bit numbers, would be 32-bit that could be placed in C and CH. The data in these 2 registers will be concatenated and so would be the input operand in two registers AH and A. Conceptually one could consider the A and AH together to represent 32-bit operand.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A - R[rc];
T5	R[ra] ← C;

Structural RTL for subtract instruction
sub ra, rb, rc

In sub instruction three registers are involved. The first three steps will fetch the sub instruction and in T3, T4, T5 the steps for execution of the sub instruction will be performed.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A + R[rc];
T5	R[ra] ← C;

Structural RTL for addition instruction
add ra, rb, rc

The table of add instruction is almost same as of sub instruction except in timing step T4 we have + sign for addition instead of – sign as in sub instruction. Other instructions that belong to the same group are ‘and’, ‘or’ and ‘sub’.

Structural RTL for multiplication instruction

mul ra, rb, rc

This instruction is only present in this processor and not in SRC. The first three steps are exactly same as of other instructions and would fetch the mul instruction. In step T3 we will bring the contents of register R

[rb] in the buffer register A at the input of ALSU. In step T4 we take the multiplication of A with the contents of R[rc] and put it at the output of the ALSU in two registers C and CH. CH would contain the higher 16-bits while register C would contain the lower 16-bits. Now these two registers cannot transfer the data in one bus cycle to the registers, since the width is 16-bits.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	CH©C ← A * R[rc],
T5	R[0] ← CH;
T6	R[ra] ← C;

So we need to have 2 timing steps, in T5 we transfer the higher byte to register R[0] and in T6 the lower 16-bits are transferred to the placeholder R[a]. As a result of multiplication instruction we need 3 timing steps for Instruction Fetch and 4 timing steps for Instruction Execution and 7 steps altogether.

Structural RTL for division instruction

div ra, rb, rc

In this instruction first three steps are the same. In step T3 the contents of register rb are placed in buffer register A and in step T4 we take the contents of register R[0] in to the register AH. We assume before using the divide instruction that we will place the higher 16-bits of dividend to register R[0]. Now in T5 the actual division takes place in two concurrent operations. We have the dividend at the input of ALSU unit represented by concatenation of AH and A. Now as a result of division instruction, the first operation would take the remainder. This means divide AH concatenated with A with the contents given in register rc and the remainder is placed in register CH at the output of ALSU. The quotient is placed in C. In T6 we take C to the register R[ra] and in T7 remainder available in CH is taken to the default register R[0] through the bus. In divide instruction 5 timing steps are required to execute the instruction while 3 to fetch the instruction.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	AH ← R[0];
T5	CH ← (AH©A)% R[rc], C ← (AH©A) / R[rc];
T6	R[ra] ← C;
T7	R[0] ← CH;

Note: Corresponding to mul and div instruction one should be careful about the additional register R[0] that it should be properly loaded prior to use the instructions e.g. if in the divide instruction we don't have the appropriate data available in R[0] the result of divide instruction would be wrong.

Structural RTL for not instruction

not ra, rb

In this instruction first three steps will fetch the instruction. In T3 we perform the not operation of contents in R[rb] and transfer them in to the buffer register C. It is simply the one's complement changing of 0's to 1's and 1's to 0's. In timing step T4 we take

Step	RTL
T0-T2	Instruction fetch
T3	C ← !(R[rb]);
T4	R[ra] ← C;

the contents of register C and transfer to register R[ra] through the bus as shown in its corresponding table.

Structural RTL for add immediate instruction addi ra, rb, c1

In this instruction c1 is a constant as a part of the instrucion. First three steps are for Instruction Fetch operation. In T3

we take the contents of register R [rb] in to the buffer register A. In

T4 we add up the contents of A with the constant c1 after sign extension and bring it to C.

Step	RTL
T0-T2	Instruction fetch
T3	A ← R[rb];
T4	C ← A + c1(sign extend);
T5	R[ra] ← C;

Sign extension of 5-bit c1 and 8-bit constant c2

Sign extension for 5-bit c1 is: (11αIR<4> ©IR<4.. 0>)

We have immediate constant c1 in the form of lower 5-bits and bit number 4 indicates the sign bit. We just copy it to the left most 11 positions to make it a 16-bit number.

Sign extension for 8-bit c2 is: (8αIR<7> ©IR<7.. 0>)

In the same way for constant c2 we need to place the sign bit to the left most 8 position to make it 16-bit number.

Structural RTL for the load and store instruction

Tables for load and store instructions are same as SRC except a slight difference in the notation. So when we have square brackets [R [rb]+c1], it corresponds to the base address in R[rb] and an offset taken from c1**.**

Step	RTL for ld	RTL for st
T0-T2	Instruction fetch	Instruction fetch
T3	A ← R[rb];	A ← R[rb];
T4	C ← A + (11αIR<4> ©IR<4..0>);	C ← A + (11αIR<4> ©IR<4..0>);
T5	MAR ← C;	MAR ← C;
T6	MBR ← M[MAR];	MBR ← R [ra];
T7	R[ra] ← MBR;	M[MAR] ← MBR;

Structural RTL for conditional jump instructions

jz ra, [c2]

In first three steps of this table, the instruction is fetched. In T3 we set a 1-bit register "CON" to true if the condition is met.

Step	RTL
T0-T2	Instruction Fetch
T3	CON ← cond(R[ra]);
T4	A ← PC;
T5	C ← A + c2(sign extend);
T6	PC ← C;

How do we test the condition?

This is tested by the contents given by the register ra. So condition within square brackets is R[ra]. This means test the data given in register ra. There are different possibilities and so the data could be positive, negative or zero. For this particular instruction it would be tested if the data were zero. If the data were zero, the "CON" would be 1.

In T4 we just take the contents of the PC into the buffer register A. In T5 we add up the contents of A to the constant c2 after sign extension. This addition will give us the effective address to which a jump would be taken. In T6, this value is copied to the PC.

In FALCON-A, the number of conditional jumps is more than in SRC. Some of which are shown below:

- jz (op-code= 19) **jump if zero**
  jz r3, [4] (R[3]=0): PC← PC+ 2;
- jnz (op-code= 18) **jump if not zero**
  jnz r4, [variable] (R[4]≠0): PC← PC+ variable;
- jpl (op-code= 16) **jump if positive**
  jpl r3, [label] (R[3]≥0): PC ← PC+ (label-PC);
- jmi (op-code= 17) **jump if negative**
  jmi r7, [address] (R[7]<0): PC← PC+ address;

The unconditional jump instruction will be explained in the next lecture.

Lecture No. 14

External FALCON-A CPU

Reading Material

Handouts Slides

Summary

- Structural RTL Description of the FALCON-A (continued…)
- External FALCON-A CPU Interface

This lecture is a continuation of the previous lecture.

Un-conditional jump instruction

jump (op-code= 20)

In the un-conditional jump with op-code 20, the op-code is followed by a 3-bit identifier for register ra and then followed by an 8-bit constant c2.

Forms allowed by the assembler to define the jump are as follows:

jump [ra + constant]
jump [ra + variable]
jump [ra + address]
jump [ra + label]

For all the above instructions:

(ra=0):PC← PC+(8αC2<7>)©C2<7..0>,
(ra≠0):PC← R[ra]+(8αC2<7>)©C2<7..0>;[4]

In the case of a constant, variable, an address or (label-PC) the jump ranges from –128 to 127 because of the restriction on 8-bit constant c2. Now, for example if we have jump [r0+a], it means jump to a. On the other hand if we have jump [– r2] that is not allowed by the assembler. The target address should be even because we have each instruction with 2 bytes. So the types available for the un-conditional jumps are either direct, indirect, PC-relative or register relative. In the case of direct jump the constant c2 would define the target address and in the case of indirect jump constant c2 would define the indirect location of memory from where we could find out the address to jump. While in the case of PC-relative if the contents of register ra are zero then we have near jump and the type of jump for this would be PC-relative. If ra is not be zero then we have a far jump and the contents of register ra will be added with the constant c2 after sign-extension to determine the jump address.

[4] c2 is computed by sign extending the constant, variable, address or (label-PC)

Structural RTL description for un-conditional jump instruction

jump [ra+c2]

Step	RTL
T0-T2	Instruction Fetch
T3	(ra=0): A← PC, (ra≠0): A ← R[ra];
T4	C ← A + c2(sign extend);
T5	PC ← C;

In first three steps, T0-T2, we would fetch the jump instruction, while in T3 we would either take the contents of PC and place them in a temporary register A if the condition given in jump instruction is true, that is if the ra field is zero, otherwise we would place the contents of register ra in the temporary register A. Comma ',' indicates that these two instructions are concurrent and only one of them would execute at a time. If the ra field is zero then it would be PC-relative jump otherwise it would be register-relative jump. In step T4 we would add the constant c2 after sign-extension to the contents of temporary register A. As a result we would have the effective address in the buffer register C, to which we need to jump. In step T5 we will take the contents of C and load it in the PC, which would be the required address for the jump.

Structural RTL for the shift instruction

shiftr ra, rb, c1

Step	RTL
T0-T2	Instruction fetch
T3	n<4..0> ← IR<4..0>;
T4	C ← (Nα0) © R[rb]<15..N>;
T5	R[ra] ← C;

First three steps would fetch the shift instruction. c1 is the count field. It is a 5-bit constant and is obtained from the lower 5-bits of the instruction register IR. In step T3 we would load the 5-bit register 'n' from the count field or the lower 5-bits of the IR and then in T4 depending upon the value of 'N' which indicates the decimal value of 'n', we would take the contents of register rb and shift right by N-bits which would indicate how many shifts are to be performed. 'n' indicates the register while 'N' indicates the decimal value of the bits present in the register 'n'. So as a result we need to copy the zeros to the left most bits, this shows that zeros are replicated 'N' times and are concatenated with the shifted bits that are actually 15…N. In T5, we take the contents from C through the bus and feed it to the register ra which is the destination register. Other instructions that would have similar tables are 'shiftl' and 'asr'.

In case of asr, when the data is shifted right, instead of copying zeros on the left side, we would copy the sign bit from the original data to the left-most position.

Other instructions

Other instructions are mov, call and ret. Note that these instructions were not available with the SRC processor.

Structural RTL for the mov instruction

mov ra, rb

Step	RTL
T0-T2	Instruction fetch
T3	C ← R[rb];
T4	R[ra] ← C;

In mov instruction the data in register rb, which is the source register, is to be moved in the register ra, which is the destination register. In first three steps, mov instruction is fetched. In step T3 the contents of register rb are placed in buffer register C through the ALSU unit while in step T4 the buffer register C transfers the data to register ra through internal uni-bus.

Structural RTL for the mov immediate instruction

movi ra, c2

Step	RTL
T0-T2	Instruction fetch
T3	C ← (8αc2<7>) © c2<7..0>;
T4	R[ra] ← C;

In this instruction ra is the destination register and constant c2 is to be moved in the ra. First three steps would fetch the move immediate instruction. In step T3 we would take the constant c2 and place it into the buffer register C. Buffer register C is 16-bit register and c2 is 8-bit constant so we need to concatenate the remaining leftmost bits with the sign bit which is bit '7' shown within angle brackets. This sign bit which is the most significant bit would be '1' if the number is negative and '0' if the number is positive. So depending upon this sign bit the remaining 8-bits are replicated with this sign bit to make a 16-bit constant to be placed in the buffer register C. In step T4 the contents of C are taken to the destination register ra.

In case of FALCON-A, 'in' and 'out' instructions are present which are not present in the SRC processor. So, for this we assume that there would be interconnection with the input and output addresses up to 0..255.

Structural RTL for the in instruction

in ra, c2

Step	RTL
T0-T2	Instruction fetch
T3	C ← IO[c2];
T4	R[ra] ← C

First three steps would fetch the instruction In step T3 we take the IO [c2] which indicates that go to IO address indicated by c2 which is a positive constant in this case and then data would be taken to the buffer register C. In step T4 we would transfer the data from C to the destination register ra.

Structural RTL for the out instruction

out ra, c2

Step	RTL
T0-T2	Instruction fetch
T3	C ← R[ra];
T4	IO[c2] ← C

This instruction is opposite to the 'in' instruction. First three instructions would fetch the instruction. In step T3 the contents of register ra are placed in to the buffer register C and then in Step T4 from C the data is placed at the output port indicated by the c2 constant. So this

instruction is just opposite to the 'in' instruction.

Structural RTL for the call instruction

call ra, rb

In this instruction we need to give the control to the procedure, sub-routine or to another address specified in the program. First three steps would fetch the call instruction. In step T3 we store the present contents of PC in to the buffer register C and then from C we transfer the data to the register ra in step T4. As a result register ra would contain the original contents of PC and this would be a pointer to come back after executing the sub-routine and it would be later used by a return instruction. In step T5 we take the contents of register rb, which would actually indicate to the point where we want to go. So in step T6 the contents of C are placed in PC and as a result PC would indicate the position in the memory from where new execution has to begin.

Step	RTL
T0-T2	Instruction Fetch
T3	C ← PC;
T4	R[ra] ← C;
T5	C ← R[rb];
T6	PC ← C;

Structural RTL for return instruction

ret ra

After instruction fetch in first 3 steps T0-T2, the register data in ra is placed in the buffer register C through ALSU unit. PC is loaded with contents of this buffer register in step T4. Assuming that bus activity is synchronized, appropriate control signals are available to us now.

Step	RTL
T0-T2	Instruction Fetch
T3	C ← R[ra];
T4	PC ← C;

Control signals required at different timing steps of FALCON-A instructions

The following table shows the details of the control signals needed. The first column is the time step, as before. In the second column the structural RTLs for the particular step is given, and the corresponding control signals are shown in the third column. Internal bus is active in step T0, causing the contents of the PC to be placed in the Memory Address register MAR and simultaneously the PC is incremented by 2 and placed it in the buffer register C. Recalling previous lectures, to write data in to a particular register we need to enable the load signal. In case of fetch instruction in step T0, control signal LMAR is enabled to cause the data from internal bus to be written in to the address register. To provide data to the bus through tri-state buffers we need to activate the 'out' control signal named as 'PCout', making contents of the PC available to the ALSU and so control unit provides the increment signal 'INC2' to increment the PC. As the ALSU is the combinational circuit, the PCout signal causes the contents over the 2nd input of ALSU incremented by 2 and so the data is available in buffer register C. Control signal "LC" is required

Step	RTL	Control Signals
T0	MAR ← PC , C ← PC + 2;	PCout, LMAR, INC2, LC
T1	MBR ← M[MAR], PC ← C;	LMBR, MRead, MARout, Cout, LPC
T2	IR ← MBR;	MBRout, LIR
T3	Instruction Execution	

to write data into the buffer register C form the ALSU output. Now note that 'INC2' is one of the ALSU functions and also it is a control signal. So knowing the control signals, which need to be activated at a particular step, is very important.

So, at step T0 the control signal 'PCout' is activated to provide data to the internal bus. Now control signal 'LMAR' causes the data from the bus to be read into the register MAR. The ALSU function 'INC2' increments the PC to 2 and the output are stored in the buffer register C by the control signal 'LC'. The data from memory location addressed by MAR is read into Memory Buffer Register MBR in the next timing step T1. In the mean time there is no activity on the internal bus, the output from the buffer register C (the incremented value of the PC) is placed in the PC through bus. For this the control signal 'LPC' is activated.

To enable tri-state buffer of Memory Address Register MAR, we need control signal 'MARout'. Another control signal is required in step T1 to enable memory read i.e. 'MRead'. In order to enable buffer register C to provide its data to the bus we need 'Cout' control signal and in order to enable the PC to read from C we need to enable its load signal, which is 'LPC'. To read data coming from memory into the Memory Buffer Register MBR, 'LMBR'control signal is enabled. So in T2 we need 5 control signals, as shown.

In T2, the instruction register IR is loaded with data from the MBR, so we need two-control signals,'MBRout' to enable its tri-state buffers and the other signal required is the load signal for IR register 'LIR'. Fetch operation is completed in steps T0-T2 and appropriate control signals are generated. Those control signals, which are not shown, would remain de-activated. All control signals are activated simultaneously so the order of these controls signals is immaterial. Recall that in SRC the fetch operation is implemented in the same way, but 'INC4' is used instead of 'INC2' because the instruction length is 4 bytes.

Now we take a look at other examples for control signals required during execution phase.

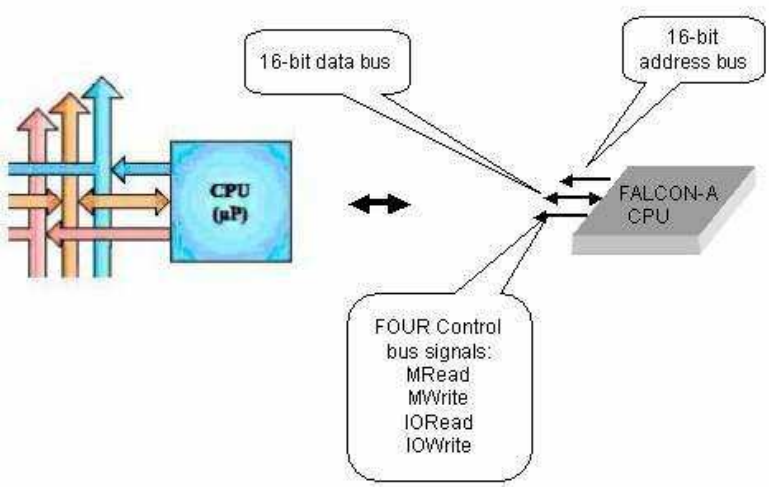
For various instructions, we will define other control signals needed in the execution phase of each instruction but fetch cycle will be the same for all instructions.

Another important fact is the interface of the CPU with an external memory and the I/O depending upon whether the I/O is memory mapped or non-memory mapped. The processor will generate some control signals, used by the memory or I/O to read/write data to/from the I/O devices or from the memory. Another assumption is that the memory read is fast enough. Therefore data from memory must be available to the processor in a fixed time interval, which in this particular example is T2.

For a slow data transfer, the concept of handshaking is used. Some idle states are introduced and buffer is prepared until the data is available. But for simplicity, we will assume that memory is fast enough and data is available in buffer register MBR to the CPU.

External FALCON-A CPU Interface

This figure is a symbolic representation of the FALCON-A in the form of a chip. The external interface consists of a 16-bit address bus, a 16-bit data bus and a control bus on which different control signals like MRead, MWrite, IORead, IOWrite are present.



Example Problem

Instruction	RTL equivalent	Address Bus <15..0>	Data Bus <15..0>	MRead	MWrite
load r7, [12+r5]					
addi r2, r4, 31					
jump [52]					
store r1,[r3+17]					
sub r5, r7, r6					
shiftr r2, r6, 4					
mov r3, r2					
jz r4, [-32]					

Memory Address	Memory Content
0020h	D2h
0021h	96h
0022h	49h
0023h	2Fh
.....	.....
C300h	44h
C301h	23h
C302h	E3h
C303h	D5h

.....	.....
C340h	51h
C341h	CAh
C343h	D5h
C344h	E2h
.....	.....
1240h	07h
1241h	85h
1242h	E5h
1243h	3Dh

(a) What will be the logic levels on the external FALCON-A buses when each of the given FALCON-A instruction is executing on the processor? Complete the table given. All numbers are in the decimal number system, unless noted otherwise.
(b) Specify memory-addressing modes for each of the FALCON-A instructions given.

Assumptions

For this particular example we will assume that all memory contents are properly aligned, i.e. memory addresses start at address divisible by 2.

PC= C348h

This table contains a partial memory map showing the addresses and the corresponding data values.
The next table shows the register map showing the contents of all the CPU registers.
Another important thing to note is that memory storage is big-endian.

Register Name	Content
R[0]	A54Bh
R[1]	4CB8h
R[2]	492Fh
R[3]	C2EFh
R[4]	2301h
R[5]	1234h
R[6]	0020h
R[7]	2D7Fh

Solution:

FALCON-A Instruction	RTL equivalent	Address Bus* <15..0>	Data Bus <15..0>	M R	M W
load r7, [r5+12]	R[7] ← M[12+R[5]]	1240h	0785h	1	0
addi r2, r4, 31	R[2] ← R[4]+31	Unknown	????	?	?
jump [52]	PC ← PC +52	Unknown	????	?	?
store r1, [r3+17]	M[R[3]+17] ← R[1]	C300h	**4CB8h**	0	1
sub r5, r7, r6	R[5] ← R[7]-R[6]	Unknown	????	?	?
shiftr r2, r6, 4	R[2] ← (4α0)©R[6]<15…4>	Unknown	????	?	?
mov r3, r2	R[3] ← R[2]	Unknown	????	?	?
jz r4, [-32]	R[4]=0:PC←PC-32	Unknown	????	?	?

In this table the second column contains the RTL descriptions of the instructions. We have to specify the address bus and data bus contents for each instruction execution. For load instruction the contents of register r5+12 are placed on the address bus. From register map shown in the previous table we can see that the contents of r5 are 1234h. Now contents of r5 are added with displacement value 12 in decimal .In other words the address bus will carry the hexadecimal value 1234h+ Ch = 1240h.Now for load instruction, the contents of memory location at address 1240h will be placed on the data bus. From the memory map shown in the previous table we can see that memory location 1240h contains 785h. Now to read this data from this location, MRead control signal will be activated shown by 1 in the next column and MWrite would be 0.Similarly RTL description is given for the 2nd instruction. In this instruction, only registers are involved so

there is no need to activate external bus. So data bus, address bus and control bus columns will contain '?' or 'unknown'. The next instruction is jump. Here PC is incremented by the jump offset, which is 52 in this case. As before, the external bus will remain inactive and control signals will be zero. The next instruction is store. Its RTL description is given. For store instruction, the register contents have to be placed at memory location addressed by R [3] +17. As this is a memory write operation, the MWrite will be 1 and MRead will be zero. Now the effective address will be determined by adding the contents of R [3] with the displacement value 17 after its conversion to the hexadecimal. The resulting effective address would be C300h. In this way we can complete the table for other instructions.

Addressing Modes

This table lists the addressing mode for each instruction given in the previous example.

FALCON-A Instruction	Addressing Mode
load r7, [r5+12]	Displacement
addi r2, r4, 31	Immediate
jump [52]	Relative
store r1, [r3+17]	Displacement
sub r5, r7, r6	Register
shiftr r2, r6, 4	Register
mov r3, r2	Register
jz r4, [-32]	Relative

Lecture No. 15

Logic Design and Control Signals Generation in SRC

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 4
Computer Systems Design and Architecture — 4.4

Summary

- Logic Design for the Uni-bus SRC
- Control Signals Generation in SRC

Logic Design for the Uni-bus SRC

In the previous sections, we have looked at both the behavioral and structural RTL for the SRC. We saw that there is a need for some control circuitry for ensuring the proper and synchronized functioning of the components of the data path, to enable it to carry out the instructions that are part of the Instruction Set Architecture of the SRC. The control unit components and related signals make up the control path. In this section, we will talk about

- Identifying the control signals required
- The external CPU interface
- Memory Address Register (MAR), and Memory Buffer Register (MBR) circuitry
- Register Connections

We will also take a look at how sign extension is performed. This study will help us understand how the entire framework works together to ensure that the operations of a simple computer like the SRC are carried out in a smooth and consistent fashion.

Identifying control signals

For any of the instructions that are a part of the instruction set of the SRC, there are certain control signals required; these control signals may be to select the appropriate function for the ALU to be performed, to select the appropriate registers, or the appropriate memory location.

Any instruction that is to be executed is first fetched into the CPU. We look at the control signals that are required for the fetch operation.

Control signals for the fetch operation

Table 1 lists the control signals that are needed to ensure the synchronized register transfers in the instruction fetch phase. Note that we use uppercase for control signals as we have been using lowercase for the instruction mnemonics, and we want to distinguish between the two. Also note that control signals during each time slot are activated simultaneously, and

Step	RTL	Control Signals
T0	MAR ← PC, C← PC + 4;	PCout, LMAR, INC4, LC
T1	MBR ← M[MAR], PC ← C;	LMBR, MRead, MARout, Cout, LPC
T2	IR ← MBR;	MBRout, LIR

Table:1

that the control signals for successive time slots are activated in sequence. If a particular control signal is not shown, its value is zero.

As shown in the Table: 1, some control signals are to let register values to be written onto buses, or read from the buses. Similarly, some signals are required to read/ write memory contents onto the bus. The memory is assumed to be fast enough to respond during a given time slot; if that is not true, wait states have to be inserted. We require four control signals to be issued in the time step T0:

PCout: This control signal allows the contents of the Program Counter register to be written onto the internal processor bus.

LMAR: This signal enables write onto the memory address register (MAR), thus the value of PC that is on the bus, is copied into this register

INC4: It lets the PC value to be incremented by 4 in the ALSU, and result to be stored in C. Notice that the value of PC has been received by the ALSU as an operand. This control signal allows the constant 4 to be added to it. The ALSU is assumed to include an INC4 function

LC: This enables the input to the register C for writing the incremented value of PC onto it.

During the time step T1, the following control signals are applied:

LMBR: This enables the "write" for the register MBR. When this signal is activated, whatever value is on the bus, can be written into the MBR.

MRead: Allow memory word to be gated from the external CPU data bus into the MBR.

MARout: This signal enables the tri-state buffers at the output of MAR.

Cout: This will enable writing of the contents of register C onto the processor's internal data bus.

LPC: This will enable the input to the PC for receiving a value that is currently on the internal processor bus. Thus the PC will receive an incremented value.

At the final time step, T2, of the instruction fetch phase, the following control signals are issued:

MBRout: To enable the tri-state buffers with the **MBR**.

LIR: To allow the **IR** read the value from the internal bus. Thus the instruction stored in the **MBR** is read into the Instruction Register (IR).

Uni-bus SRC implementation

The uni-bus implementation of the SRC data path is given in the Fig.1. We can now visualize

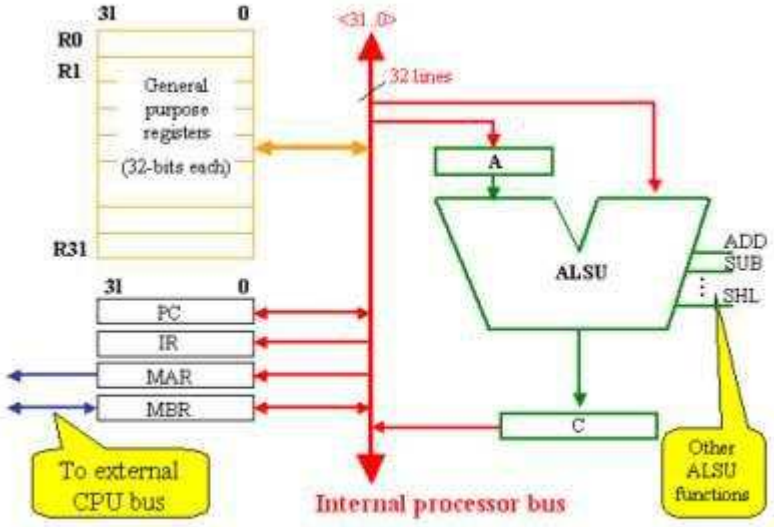


Fig.1

how the control signals in mutually exclusive time steps will allow the coordinated working of instruction fetch cycle.

Similar control signals will allow the instruction execution as well. We have already mentioned the external CPU buses that read from the memory and write back to it. In the given figure, we had not shown these external (address and data buses) in detail. Fig.2 will help us understand this external interface.

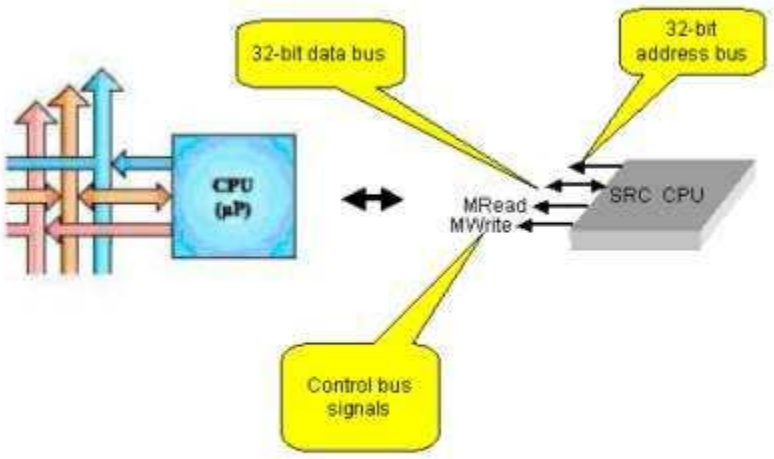


Fig.2

External CPU bus activity

Let us take up a sample problem to further enhance our understanding of the external CPU interface. As mentioned earlier, this interface consists of the data bus/ address bus, and control signals for enabling memory read and write.

Example problem:

(a) What will be the logic levels on the external SRC buses when each of the given SRC instruction is executing on the processor? Complete Table: 2. all numbers are in the decimal number system, unless noted otherwise.

(b) Specify memory addressing modes for each of the SRC instructions given in Table: 2.

Instruction	RTL equivalent	Address Bus <31..0>	Data Bus <31..0>	MRead	MWrite
ld r7, 12(r5)					
ld r2, 32					
la r9, 32					
ldr r12, -72					
lar r3, 0					
st r2, 0(r6)					
str r3, -8					
st r4, 32					

Table 2

Assumptions:

- All memory content is aligned properly.

In other words, all the memory accesses start at addresses divisible by 4. Value in the PC = 000DC348h

(Note that the SRC uses the big-endian storage format).

Memory map with assumed values

Memory Address	Memory Content
00000020h	D2h
00000021h	96h
00000022h	49h
00000023h	2Fh
.....	.....
000DC300h	44h
000DC301h	23h
000DC302h	E3h
000DC303h	D5h

.....	.....
000DC340h	51h
000DC341h	CAh
000DC343h	D5h
000DC344h	E2h
.....	.....
00AB1240h	07h
00AB1241h	85h
00AB1242h	E5h
00AB1243h	3Dh

Fig.3

Register map with assumed values

Register Name	Content
R[0]	0012A54Bh
R[1]	10234CB8h
R[2]	D296492Fh
R[3]	001400CDh
R[4]	B7432301h
R[5]	00AB1234h
R[6]	00000020h
R[7]	01432D7Fh
R[8]	00B94821h
R[9]	00CDA7A3h
R[10]	0031A0F0h
R[11]	0012A246h
R[12]	000FAB17h

Fig.4

Solution Part (a):

Instruction	RTL equivalent	Address Bus <31..0>	Data Bus <31..0>	MRead	MWrite
ld r7, 12(r5)	R[7] ← M[12+R[5]]	00AB1240 h	0785E53D h	1	0
ld r2, 32	R[2] ← M[32]	00000020 h	D296492F h	1	0
la r9, 32	R[9] ← 32	Unknown	Unknown	?	?
ldr r12, -72	R[12] ← M[PC - 72]	000DC300 h	4423E3D5 h	1	0
lar r3, 0	R[3] ← PC	Unknown	Unknown	?	?
st r2, 0(r6)	M[R[6]] ← R[2]	00000020 h	D296492F h	0	1
str r3, -8	M[PC - 8] ← R[3]	000DC340 h	001400CD h	0	1
st r4, 32	M[32] ← R[4]	00000020 h	B7432301 h	0	1

Solution part (b):

SRC Instruction	Addressing Mode
ld r7, 12(r5)	Displacement
ld r2, 32	Direct
la r9, 32	Immediate
ldr r12, -4	PC Relative
lar r3, 0	Register
st r2, 0(r6)	Register Indirect
str r3, -8	PC Relative
st r4, 32	Register Direct

Fig:5

Notes:

* Relative addressing is always PC relative in the SRC

*** Displacement addressing mode is the same as Based or Indexed in the SRC. It is also the same as Register Relative addressing mode

Memory address register circuitry

We have already talked about the functionality of the **MAR.** It provides a temporary storage for the address of memory location to be accessed. We now take a detailed look at how it is interconnected with other components. The MAR is connected directly to the CPU internal bus, from which it is loaded (receives a value). The LMAR signal causes the contents of the internal CPU bus to be loaded into the MAR. It writes onto the CPU external address bus. The MARout signal causes the contents of the MAR to be placed on the address bus. Thus, it provides the addresses for the memory and I/O devices over the CPU's address bus. A set of tri-state buffers

is provided with these connections; the tri-state buffers are controlled by the control signals, which in turn are issued when the corresponding instruction is decoded. The whole circuitry is shown in Fig.6.

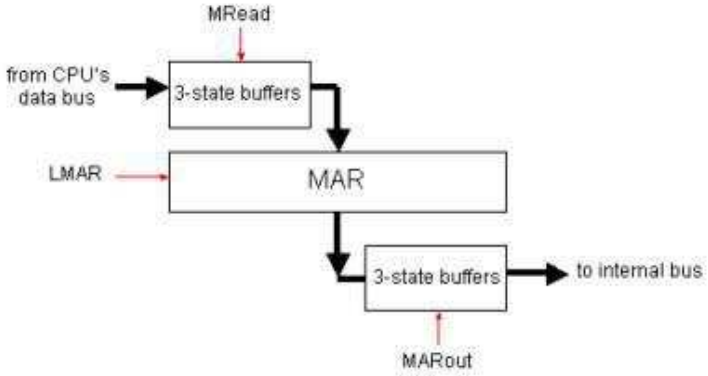


Fig:6

Memory buffer register circuitry

The Memory Buffer Register (MBR) holds the value read from the memory or I/O device. It is possible to load the MBR from the internal CPU bus or from the external CPU data bus. The MBR also drives the internal CPU bus as well as the external CPU data bus. Similar to the MAR register, tri-state buffers are provided at the connection points of the MBR, as illustrated in the Fig.7.

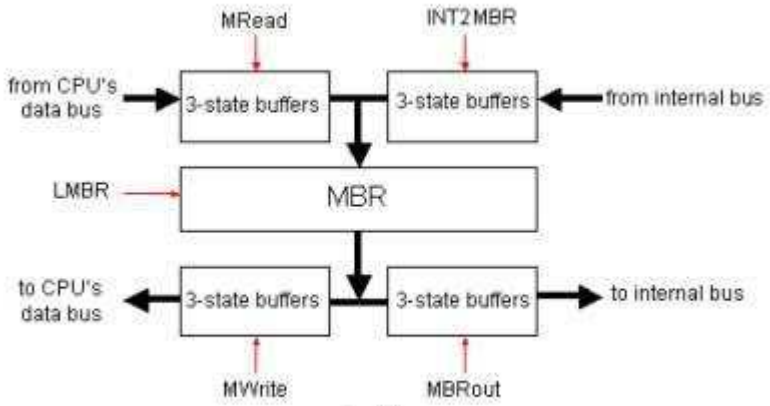


Fig:7

Register connections

The register file containing the General Purpose Registers is programmer visible. Instructions may refer to any of these registers, as source operands in an operation or as the destination registers. Appropriate circuitry is needed to enable the specified register for read/ write. Intuitively, we can tell that we require connections of the register to the CPU internal bus, and we need control signals that will enable specified registers to be read/ write enabled as a corresponding instruction is decoded. Fig.8 illustrates the register connections and the control signals generation in the uni-bus data path of the SRC. We can see from this figure that the ra, rb and rc fields of the Instruction Register specify the destination and source registers. The control signals RAE, RBE and RCE can be applied to select any of the ra, rb or rc field respectively to apply its contents to the input of 5-to-32 decoder. Through the decoder, we get the signal for the specific register to be accessed. The BUS2R control signal is activated if it is desired to write into the register. On the other hand, if the register contents are to be written to the bus, the control signal R2BUS is activated.

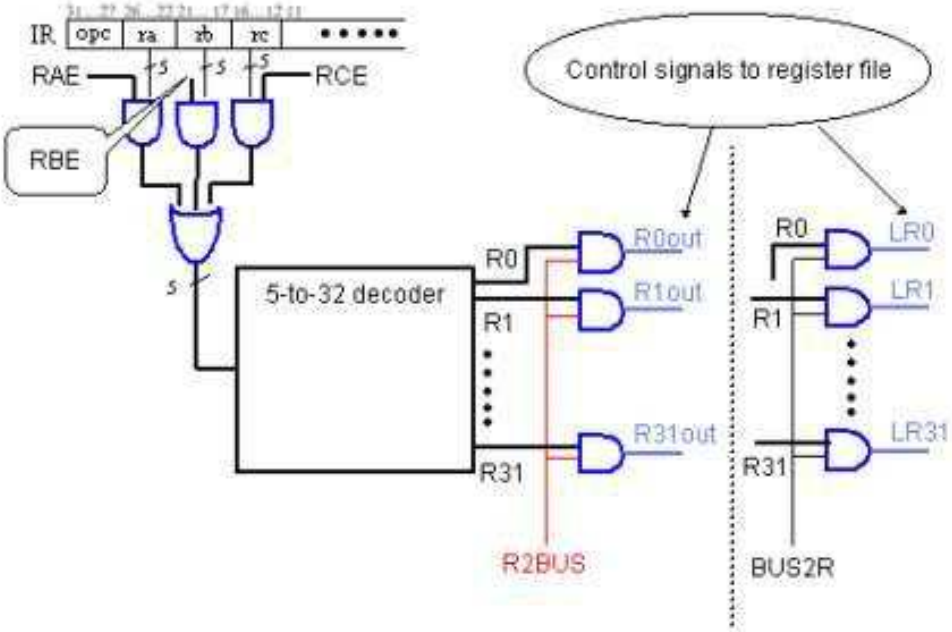


Fig.8

Alternate control circuitry for register selection

Fig.9 illustrates an alternate circuitry that implements the register connections with the internal processor bus, the instruction register fields, and the control signals required to coordinate the appropriate read/write for these registers. Note that this implementation is somewhat similar to our earlier implementation with a few differences. It illustrates the fact that the implementations we have presented are not necessarily the only solutions, and that there may be other possibilities.

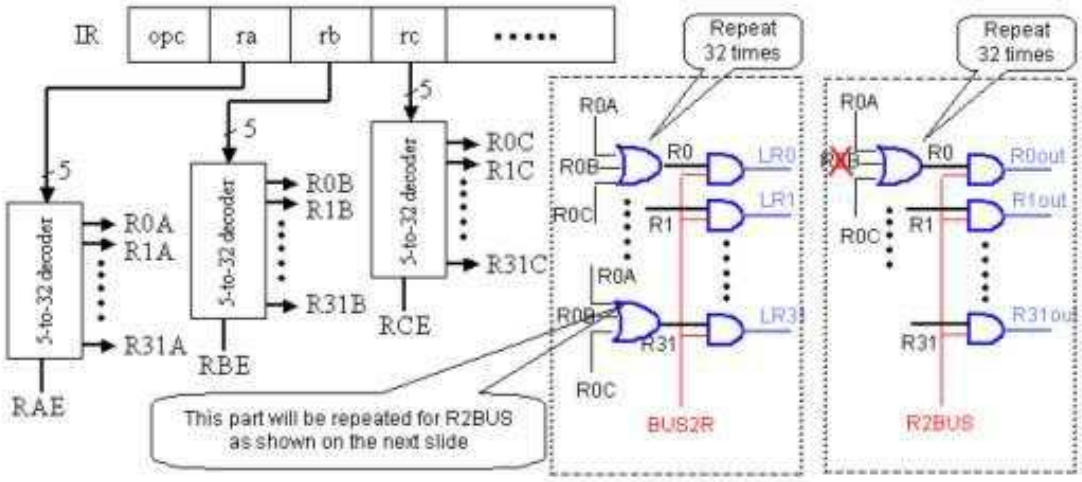


Fig.9

In this alternate circuitry, there is a separate 5-to-32 decoder for each of the register fields of the instruction register. The output of these decoders is allowed to be read out and enables the decoded register, if the control signal (RAE, RBE or RCE) is active.

Control signals Generation in SRC

We take a few example instructions to study the control signals that are required in the instruction execution phase.

Control signals for the add instruction

The add instruction has the following syntax:

add ra, rb, rc

Table: 4 lists the control signals that are applied at each of the time steps. The first three steps are of the instruction fetch phase, and we have already discussed the control signals applied at this phase.

Step	RTL	Control Signals
T0 – T2	Instruction Fetch	As before
T3	A ← R[rb];	RBE, R2BUS, LA
T4	C ← A+R[rc];	RCE, R2BUS, ADD, LC
T5	R[ra] ← C;	Cout, RAE, BUS2R

At time step T3, the control RBE is applied, which will enable the register rb to write its contents onto the internal CPU bus, as it is decoded. The writing from the register onto the bus is enabled by the control signal R2BUS. Control signal LA allows the bus contents to be transferred to the register A (which will supply it to the ALSU). At time step T4, the control signals applied are RCE, R2BUS, ADD, LC, to respectively enable the register rc, enable the register to write onto the internal CPU bus (which will supply the second operand to the ALSU from the bus), select the add function of the ALSU (which will add the values) and enable register C (so the result of the addition operation is stored in the register C). Similarly in T5, signals Cout, RAE and BUS2R are activated.

Sign extension

When we copy constant values to registers that are 32 bits wide, we need to sign extend the values first. These values are in the 2's complement form, and to sign-extend these values, we need to copy the most significant bit to all the additional bits in the register.

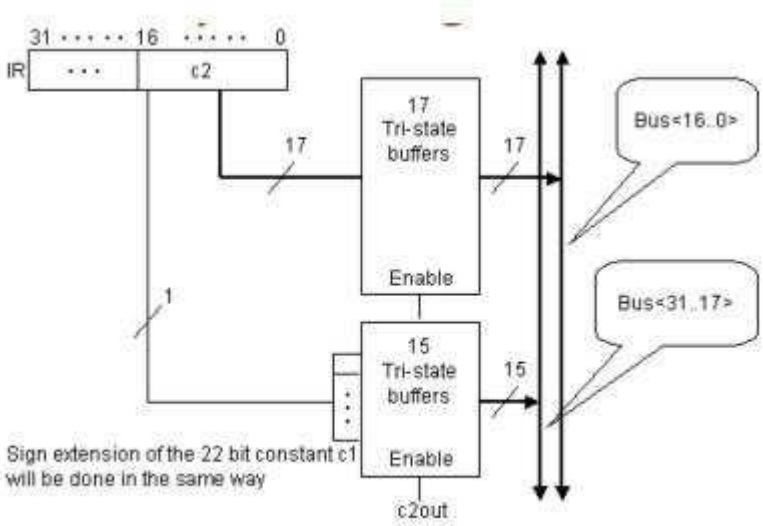


Fig:10

We consider the field c2, which is a 17 bit constant. Sign extension of c2 requires that we copy c2<16> to all the left-most bits of the destination register, in addition to copying the original constant values to the register. This means that bus<31...17> should be the same as c2<16>. A 15 line tri-state buffer can perform this sign extension. So we apply c2<16> to all the inputs of this tri-state buffer as illustrated in the Fig.10.

Structural RTL for the addi instruction

We now return to our study of the control signals required in the instruction execute phase. We have already looked at the add instruction and the corresponding signals. Now we take a look at the **addi** (add immediate) instruction, which has the following syntax:

addi ra, rb, c2

Table: 5 lists the RTL and the control signals for the **addi** instruction:

Step	RTL for addi	Control signals
T0-T2	Instruction fetch	As before
T3	A ← R[rb];	RBE, R2BUS, LA
T4	C ← A + c2(sign extend);	c2out, ADD, LC
T5	R[ra] ← C;	Cout RAE, BUS2R

Table:5

The table shows that the control signals for the addi instruction are the same as the add instruction, except in the time step T4. At this time step, the control signals that are applied are c2out, ADD and LC, to respectively do the following:

Enable the read of the constant c2 (which is sign extended) onto the internal processor bus. Add the values using the ALSU and finally assign the result to register C by enabling write for this register.

To place a 0 on the bus

When the field rb is zero, for instance, in the **load** and **store** instructions, we need to place a zero on the bus. The given circuit in Fig.11 can be used to do this.

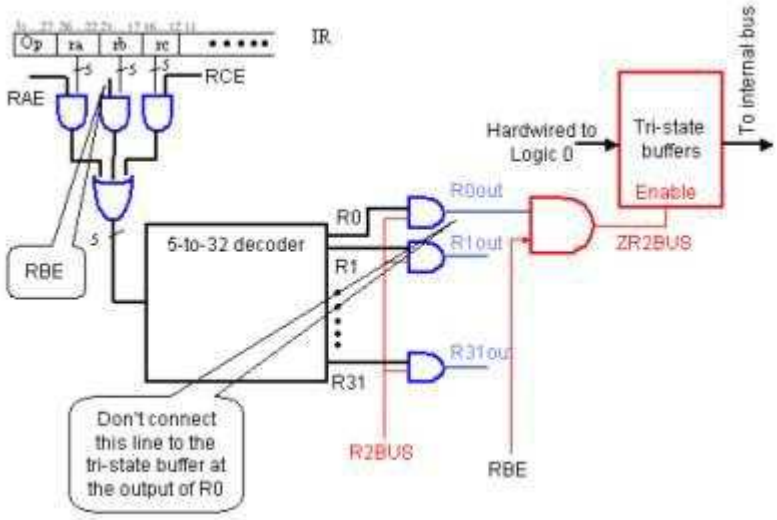


Fig:11

Note that, by default, the value of register R0 is 0 in some cases. So, when the selected register turns out to be 0 (as rb field is 0), the line connecting the output of the register R0 is not enabled, and instead a hardwired 0 is output from the tri-state buffer onto the CPU internal bus. An alternate circuitry for achieving the same is shown in the Fig.12.

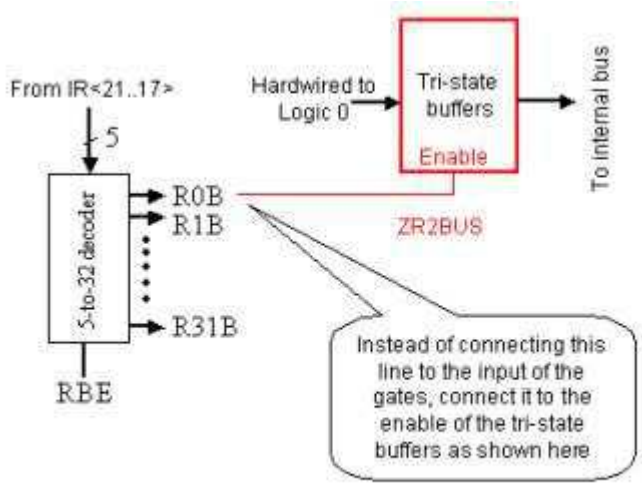


Fig:12

Control signals for the ld instruction

Now we take a look at the control signals for the **load** instruction. The syntax of the instruction is:

ld ra, c2 (rb)

Table: 6 outlines the control signals as well as the RTL for the **load** instruction in the SRC.

The first three steps are of the instruction fetch

Step	RTL for ld	Control Signals
T0-T2	Instruction fetch	As before
T3	A ← ((rb = 0) : 0, (rb ≠ 0): R[rb]);	RBE, R2BUS, LA
T4	C ← A + (16¤IR<16> ©IR<15..0>);	C2out, ADD, LC
T5	MAR ← C;	Cout, LMAR
T6	MBR ← M[MAR];	MARout, MRead, LMBR
T7	R[ra] ← MBR;	MBRout, RAE, BUS2R

Table:6

RBE is issued to allow the register rb value to be read **R2BUS** to allow the bus to read from the selected register

LA to allow write onto the register A. This will allow the CPU bus contents to be written to the register A.

At step T4 the control signals are:

c2out to allow the sign extended value of field c2 to be written to the internal CPU bus **ADD** to instruct the ALSU to perform the add function.

LC to let the result of the ALSU function be stored in register C by enabling write of register C.

Control signals issued at step T5:

Cout is to read the register C, this copies the value in C to the internal CPU bus.

LMAR to enable write of the Memory Address Register (which will copy the value present on the bus to MAR). This is the effective address of memory location that is to be accessed to read (load) the memory word.

During the time step T6:

MARout to read onto the external CPU bus (the address bus, to be more specific), the value stored in the MAR. This value is an index to memory location that is to be accessed.

MRead to enable memory read at the specified location, this loads the memory word at the specified location onto the CPU external data bus.

LMBR is the control signal to enable write of the MBR (Memory Buffer Register). It will obtain its value from the CPU external data bus. Finally, the control signals issued at the time step T7 are:

MBRout is the control signal to allow the contents of the MBR to be read out onto the CPU internal bus.

RAE is the control signal for the destination register field ra. It will let the actual index of the ra register be encoded, and

BUS2R will let the appropriate destination register be written to with the value on the CPU internal bus.

Lecture No. 16

Control Unit Design

Reading Material

Vincent P. Heuring & Harry F. Jordan — Chapter 4
Computer Systems Design and Architecture — 4.2.2, 4.6.1

Summary

- Control Signals Generation in SRC (continued…)
- The Control Unit
- 2-Bus Implementation of the SRC Data Path

This section of lecture 16 is a continuation of the previous lecture.

Control signals for the store instruction st ra, c2(rb)

The store time step operations are similar to the load instruction, with the exception of steps T6 and T7. However, one can easily interpret these now. These are outlined in the given table.

Step	RTL for st	Control Signals
T0-T2	Instruction fetch	As before
T3	A ← ((rb = 0): 0, (rb ≠ 0): R[rb]);	RBE, R2BUS, BAout, LA
T4	C ← A + (16@IR<16> ©IR<15..0>);	C2out, ADD, LC
T5	MAR ← C;	Cout, LMAR
T6	MBR ← R [ra];	RAE, R2BUS, INT2MBR, LMBR
T7	M[MAR] ← MBR;	MARout, MWrite

Control signals for the branch and branch link instructions

Branch instructions can be either be simple branches or link-and-then-branch type. The syntax for the branch instructions is

brzr rb, rc

This is the branch and zero instruction we looked at earlier. The control signals for this instruction are:

Step	RTL for br	Control signals
T0-T2	Instruction Fetch	As before
T3	CON ← cond(R[rc]);	LCON, RCE, R2BUS
T4	CON: PC ← R[rb]	RBE, R2BUS, LPC (if CON=1)

As usual, the first three steps are for the instruction fetch phase. Next, the following control signals are issued:

LCON to enable the CON circuitry to operate, and instruct it to check for the appropriate condition (whether it is branch if zero, or branch if not equal to zero, etc.) **RCE** to allow the register rc value to be read.
R2BUS allows the bus to read from the selected register.
At step T4:
RBE to allow the register rb value to be read. rb value is the branch target address.
R2BUS allows the bus to read from the selected register.
LPC (if CON=1): this control signal is issued conditionally, i.e. only if CON is 1, to enable the write for the program counter. CON is set to 1 only if the specified condition is met. In this way, if the condition is met, the program counter is set to the branch address.

Branch and link instructions

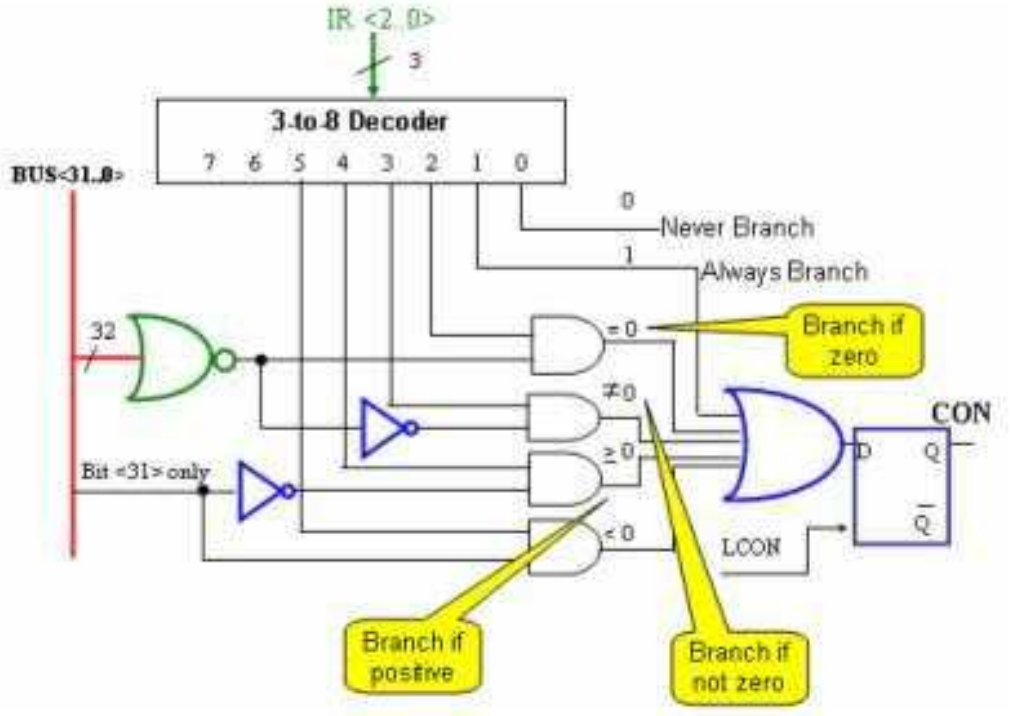
The branch and link instruction is similar to the branch instruction, with an additional step, T4. Step T4 of the simple conditional branch instruction becomes the step T5 in this case.

Step	RTL	Control signals
T0-T2	Instruction Fetch	As before
T3	CON ← cond(R[rc]);	LCON, RCE, R2BUS
T4	CON: R[ra] ← PC;	RAE, BUS2R, PCout (if CON=1)
T5	CON: PC ← R[rb];	RBE, R2BUS, LPC (if CON=1)

The syntax of the instruction 'branch and link if zero' is

brlzr ra, rb, rc

Table that lists the RTL and control signals for the store instruction of the SRC is given: The circuitry that enables the condition checking for the conditional branches in the SRC is illustrated in the following figure:

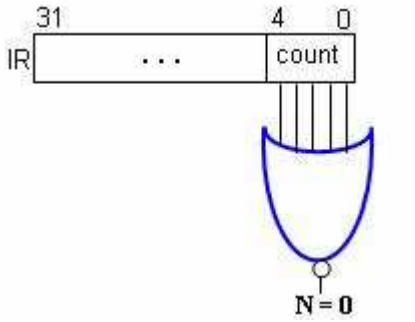


Control signals for the shift right instruction

The given table illustrates the RTL and the control signals for the shift right 'shr' instruction. This is implemented by applying the five bits of n (nb4, nb3, nb2, nb1, nb0) to the select inputs of the barrel shifter and activating the control signal SHR as explained in an earlier lecture.

Step	RTL for shr	Control signals
T0-T2	Instruction Fetch	As before
T3	n<4..0> ← IR<4..0>;	LN
T4	(N = 0) : (n<4..0> ← R[rc]<4..0>);	LN(N=0), RCE, R2BUS
T5	C ← (Nα0) ©R[rb]<31..N>;	LC, SHR(N)
T6	R[ra] ← C;	Cout, RAE, BUS2R

Generating the Test Condition N=0



The Control Unit

The control unit is responsible for generating control signals as well as the timing signals. Hence the control unit is responsible for the synchronization of internal as well as external events. By means of the control signals, the control unit instructs the data path what to do in every clock cycle during the execution of instructions.

Control Unit Design

Since the control unit performs quite complex tasks, its design must be done very carefully. Most errors in processor design are in the Control Unit design phase. There are primarily two approaches to design a control unit.

1. Hardwired approach
2. Micro programming

Hardwired approach is relatively faster, however, the final circuit is quite complex. The micro-programmed implementation is usually slow, but it is much more flexible.

"Finite-state machine" concepts are usually used to represent the CU. Every state corresponds to one "clock cycle" i.e., 1 state per clock. In other words each timing step could be considered as just 1 state and therefore from one timing step to other timing step, the state would change. Now, if we consider the control unit as a black box, then there would be four sets of inputs to the control unit. These are as follows:

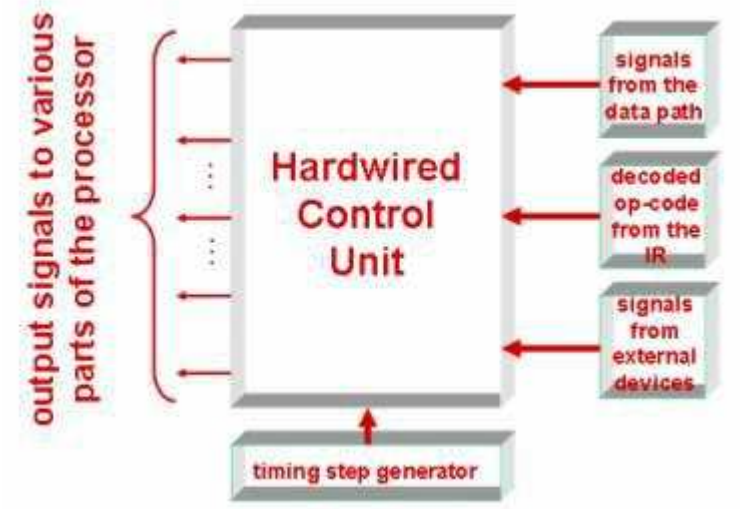
1. The output of timing step generator (There are 8 disjoint timing steps in our example T0-T7).
2. Op-code (op-code is first given to the decoder and the output of the decoder is given to the control unit).
3. Data path generated signals, like the "CON" control signal,
4. Signals from external events, like "Interrupt" generated by the Interrupt generator.

The complexity of the control is a function of the

- Number of states
- Number of inputs to the CU
- Number of the outputs generated by the CU

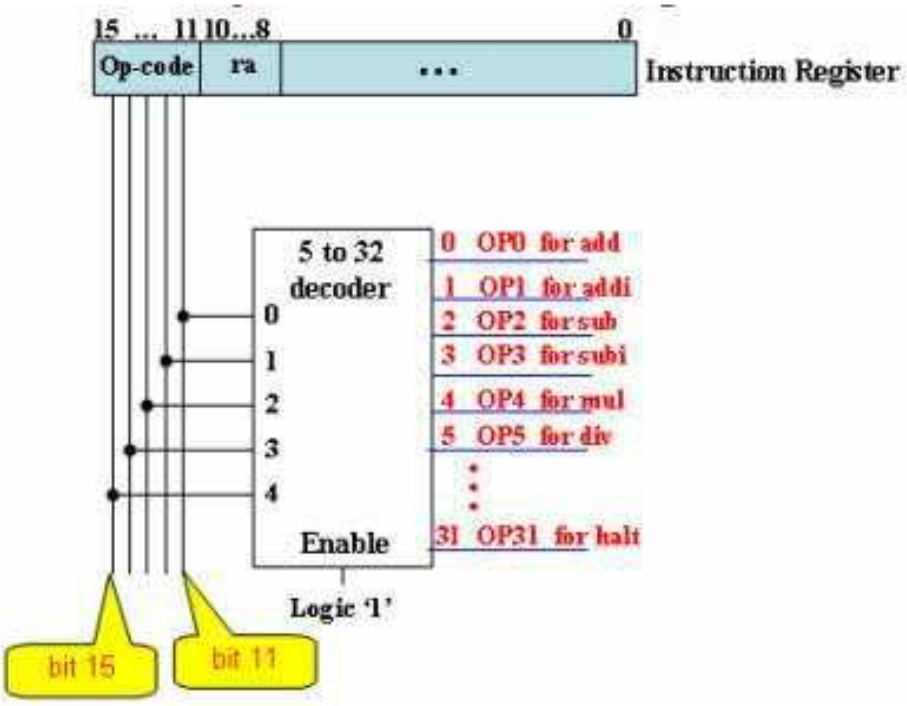
Hardwired Implementation of the Control Unit

The accompanying block diagram shows the inputs to the control unit. The output control signals generated from control unit to the various parts of the processor are also shown in the figure.



Example Control Unit for the FALCON-A

The following figure shows how the operation code (op-code) field of the Instruction Register is decoded to generate a set of signals for the Control unit.



This is an example for the FALCON-A processor where the instruction is 16-bit long. Similar concepts will apply to the SRC, in which case the instruction word is 32 bits and IR <31...27> contains the op-code. Similar concepts will apply to the SRC, in which case the instruction word is 32 bits and IR<31..27> contains the opcode. The most significant 5 bits represent the op-code. These 5-bits from the IR are fed to a 5-to-32 decoder. These 32 outputs are numbered from 0-to-31 and named as op0, op1 up to op31. Only one of these 32 outputs will be active at a given time .The active output will correspond to instruction executing on the processor.

To design a control unit, the next step is to write the Boolean Equations. For this we need to browse through the structural descriptions to see which particular control signals occur in different timing steps. So, for each instruction we have one such table defining structural RTL and the control signals generated at each timing step. After browsing we need to check that which control signal is activated under which condition. Finally we need to write the expression in the form of a logical expression as the logical combination of "AND" and "OR" of different control signals. The given table shows Boolean Equations for some example control signals.

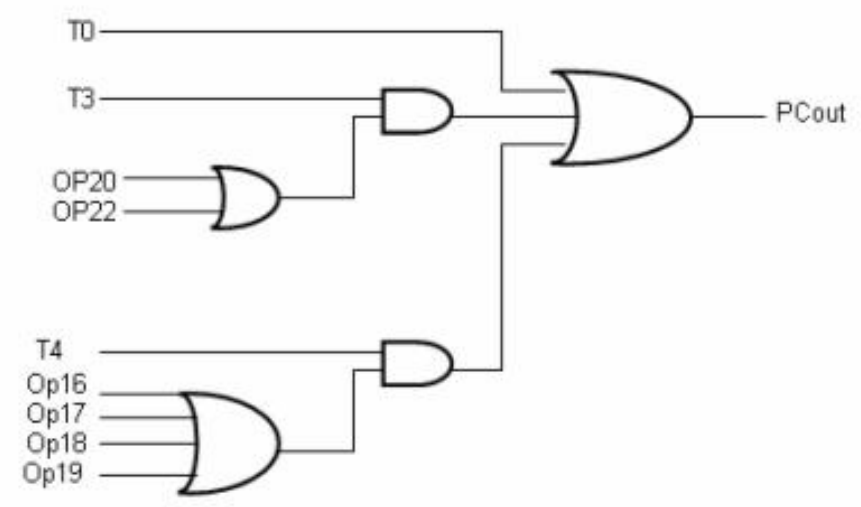
Step	RTL	Control Signals
T0	MAR ← PC;	PCout, LMAR, C=B;
T1	MBR ← M[MAR], PC ← PC + 4;	PCout, INC4, LPC, MRead, MARout, LMBR;
T2	IR ← MBR;	MBRout, C=B, LIR;
T3	Instruction Execution	

For example, PCout would be active in every T0 timing step. Then in timing interval T3 the output of the PC would be activated if the op-code is 20 or 22 which represent jump and sub-routine call. In step T4 if the op-code is 16, 17, 18 or 19, again we need PCout activated and these 4 instructions correspond to the conditional jumps. We can say that in other words in step

T1, PCout is always activated "OR" in T3 it is activated if the instruction is either jump or sub-routine call "OR" in T4 if there is one of the conditional jumps. We can write an equation for it as

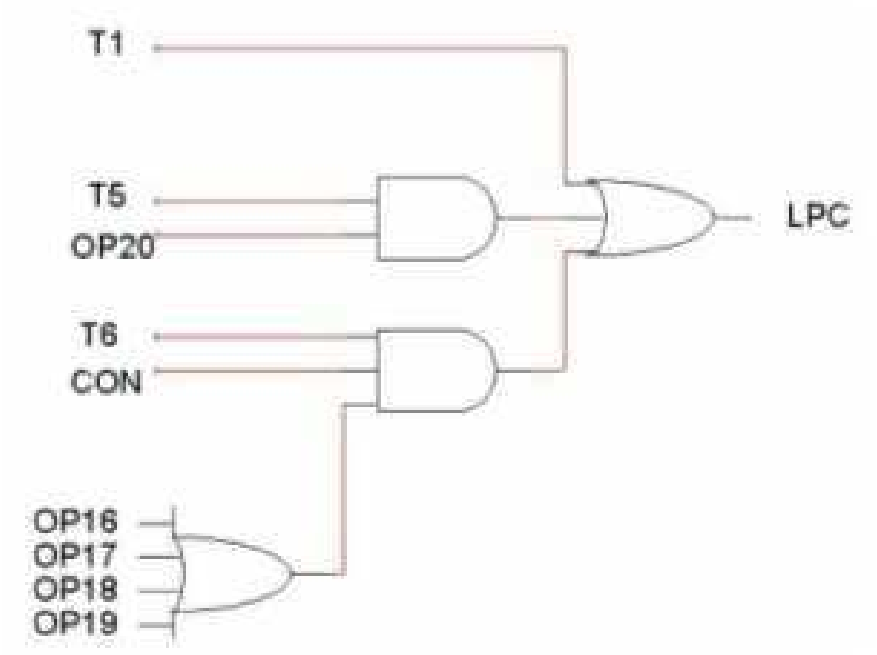
PCout = T0 + T3.(OP20 + OP22) + T4.(OP16 + OP17 + OP18 + OP19)

In the form of logic circuit the implementation is shown in the figure. We can see that we "OR" the op-ode 20 and 22 and "AND" it with T3, then "OR" all the op16 up to op19 and "AND" it with T4, then T0 and the "AND" outputs of T3 and T4 are "OR" together to obtain the PCout.



In the same way the logic circuit for LPC control signal is as shown and the equation would be :

LPC = T1 + T5.OP20 + T6.CON.(OP16 + OP17 + OP18 + OP19)



We can formulate Boolean equations and draw logic circuits for other control signals in the same way.

Effect of using "real" Gates

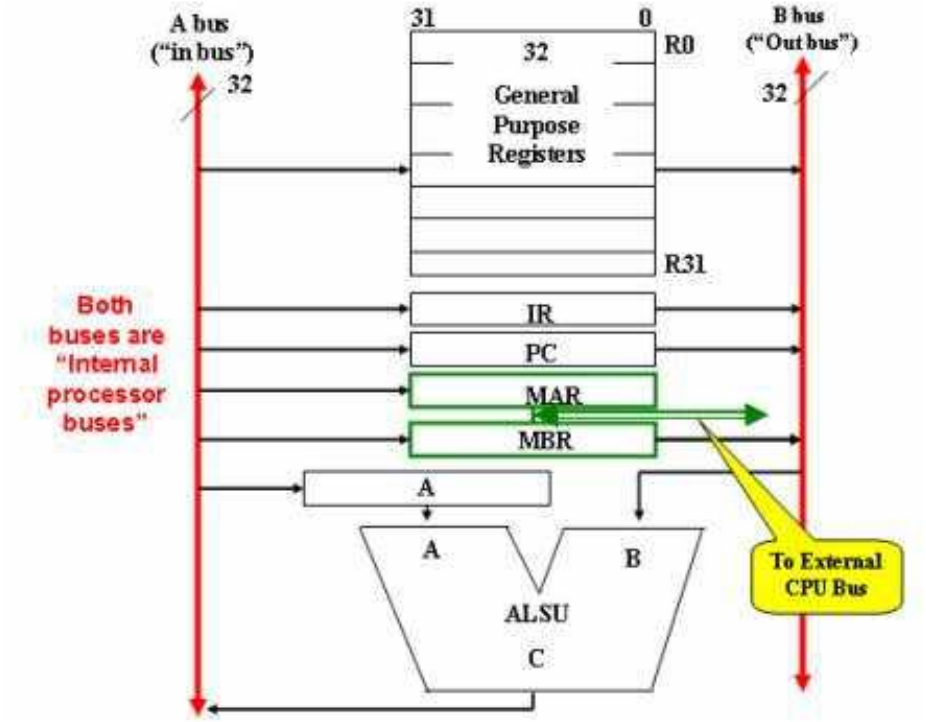
We have assumed so far that the gates are ideal and that there is no propagation delay. In designing the control unit, the propagation delays for the gates cannot be neglected. In particular, if different gates are cascaded, the output of one gate forms the input of other. The propagation delays would add up. This, in turn would place an upper limit on the frequency of the clock which controls the generation of the timing intervals T0, T1… T7. So, we cannot arbitrarily increase the frequency of this clock. As an example consider the transfer of the contents of a register R1 to a register R2. The minimum time required to perform this transfer is given by

$$t_{min} = t_g + t_{bp} + t_{comb} + t_1$$

The details are explained in the text with reference to Fig 4.10. Thus, the maximum clock frequency based on this transfer will be $1/t_{min}$. Students are encouraged to study example 4.1 of the text.

2-Bus Implementation of the SRC Data Path

In the previous sections, we studied the uni-bus implementation of the data path in the SRC. Now we present a 2-bus implementation of the data path in the SRC. We observe from this figure that there is a bus provided for data that is to be written to a component. This bus is named the 'in' bus. Another bus is provided for reading out the values from these components. It is called the 'out' bus.



Structural RTL for the 'sub' instruction using the 2-bus data path implementation

Next, we look at the structural RTL as well as the control signals that are issued in sequence for instruction execution in a 2-bus implementation of the data path. The given table illustrates the Register Transfer Language representation of the operations for carrying out instruction fetch, and execution for the sub instruction.

	Step	RTL
Instruction Fetch	T0	MAR ← PC;
	T1	MBR ← M[MAR], PC ← PC + 4;
	T2	IR ← MBR;
Instruction Execute	T3	A ← R[rb];
	T4	R[ra] ← A - R[rc];

The first three steps belong to the instruction fetch phase; the instruction to be executed is fetched into the Instruction Register and the PC value is incremented to point to the next-in-line instruction. At step T3, the register R[rb] value is written to register A. At the time step T4, the subtracted result from the ALSU is assigned to the destination register R[ra]. Notice that we did not need to store the result in a temporary register due to the availability of two buses in place of one. At the end of this sequence, the timing step generator is initialized to T0.

Control signals for the fetch operation

The control signals for the instruction fetch phase are shown in the table. A brief explanation is given below:

Step	RTL	Control Signals
T0	MAR ← PC;	PCout, LMAR, C=B;
T1	MBR ← M[MAR], PC ← PC + 4;	PCout, INC4, LPC, MRead, MARout, LMBR;
T2	IR ← MBR;	MBRout, C=B, LIR;
T3	Instruction Execution	

At time step **T0**, the following control signals are issued:

- **PCout**: This will enable read of the Program Counter, and so its value will be transferred onto the 'out' bus
- **LMAR**: To enable the load for MAR
- **C=B:** This instruction is used to copy the value on the 'out' bus to the 'in' bus, so it can be loaded into the Memory Address Register. We can observe in the data-path implementation figure given earlier that, at any time, the value on the 'out' bus makes up the operand B for the ALSU. The result C of ALSU is connected to the "in" bus, and therefore, the contents transfer from one bus to the other can take place.

At time step T1:

- **PCout:** Again, this will enable read of the Program Counter, and so its value will be transferred onto the CPU internal 'out' bus
- **INC4:** To instruct the ALSU to perform the increment-by-four operation.
- **LPC:** This control signal will enable write of the Program Counter, thus the new, incremented value can be written into the PC if it is made available on the "in" bus. Note that the ALSU is assumed to include an INC4 function.
- **MRead:** To enable memory word read.
- **MARout:** To supply the address of memory word to be accessed by allowing the contents of the MAR (memory address register) to be written onto the CPU external (address) bus.
- **LMBR:** The memory word is stored in the register MBR (memory buffer register) by applying this control signal to enable the write of the MBR.

At time step T2:

- **MBRout:** The contents of the Memory Buffer Register are read out onto the 'out' bus, by means of applying this signal, as it enables the read for the MBR.
- **C=B:** Once again, this signal is used to copy the value from the 'out' bus to the 'in' bus, so it can be loaded into the Memory Address Register.
- **LIR:** This instruction will enable the write of the Instruction Register. Hence the instruction that is on the 'in' bus is loaded into this register.

At time step T3, the execution may begin, and the control signals issued at this stage depend on the actual instruction encountered. The control signals issued for the instruction fetch phase are the same for all the instructions.

Note that, we assume the memory to be fast enough to respond during a given time slot. If that is not true, wait states have to be inserted. Also keep in mind that the control signals during each time slot are activated simultaneously, while those for successive time slots are activated in sequence. If a particular control signal is not shown, its value is zero.

Lecture No. 17
Machine Reset and Machine Exceptions

Reading Material

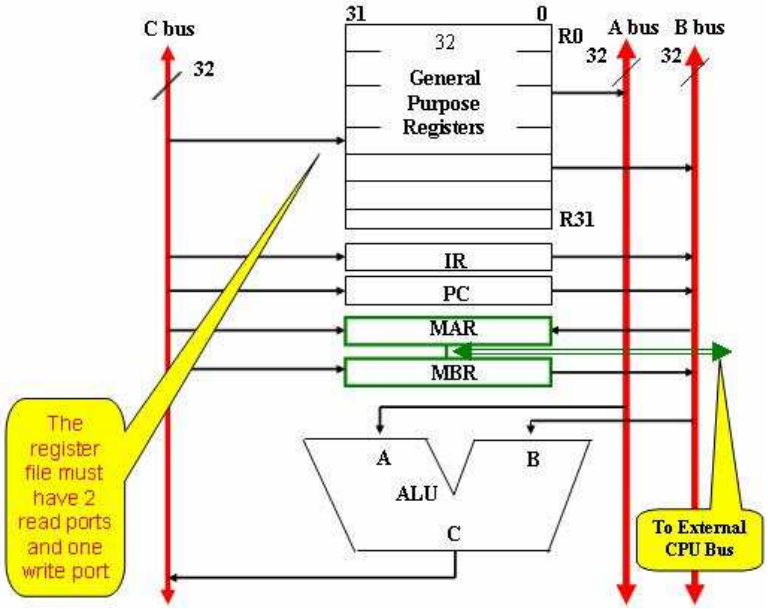
Vincent P. Heuring & Harry F. Jordan — Chapter 4
Computer Systems Design and Architecture — 4.6.2, 4.7, 4.8

Summary

- 3-bus implementation for the SRC
- The Machine Reset
- Machine Exceptions

A 3-bus Implementation for the SRC

Let us now look at a 3-bus implementation of the data-path for the SRC as shown in the figure. Two buses, 'A' and 'B' bus for reading, and a bus 'C' for writing, are part of this implementation. Hence all the special purpose as well as the general purpose registers have two read ports and one write port.



Structural RTL for the Subtract Instruction using the 3-bus Data Path Implementation

We now consider how instructions are fetched and executed in 3-bus architecture. For this purpose, the same 'sub' instruction example is followed.

The syntax of the subtract instructions is

sub ra, rb, rc

The structural RTL for implementing this instruction is given in the table. We observe that in this table, only two time steps are required for the instruction fetch phase.

	Step	RTL
Instruction Fetch	T0	MAR←PC; MBR ← M[MAR], PC ← PC + 4;
	T1	IR ← MBR;
Instruction Execute	T2	R[ra] ← R[rb] - R[rc];

At time step T0, the Memory Address Register receives the value of the Program Counter. This

is done in the initial phase of the time step T0. Then, the Memory Buffer Register receives the memory word indexed by the MAR, and the PC value is incremented. At time step T1, the instruction register is assigned the instruction word that was loaded into the MBR in the previous time step. This concludes the instruction fetch and now the instruction execution can commence.

In the next time step, T2, the instruction is executed by subtracting the values of register rc from rb, and assigning the result to the register ra.
At the end of each sequence, the timing step generator is initialized to T0

Control Signals for the Fetch Operation
The given table lists the control signals in the instruction fetch phase. The control signals for the execute phase can be written in a similar fashion.

Step	RTL	Control Signals
T0	MAR←PC; MBR ← M[MAR], PC ← PC + 4;	PCout, INC4, LPC, LbMAR, MRead,
T1	IR←MBR;	MBRout, C=B, LIR;
T2	Instruction_ Execution	

The Machine Reset

In this section, we will discuss the following

- Reset operation
- Behavioral RTL for SRC reset
- Structural RTL for SRC reset

The reset operation
Reset operation is required to change the processor's state to a known, defined value. The two essential features of a reset instruction are clearing the control step counter and reloading the PC to a predefined value. The control step counter is set to zero so that operation is restarted from the instruction fetch phase of the next instruction. The PC is reloaded with a predefined value usually to execute a specific recovery or initializing program.

In most implementations the reset instruction also clears the interrupt enable flags so as to disable interrupts during the initialization operation. If a condition code register is present, the reset instruction usually clears it, so as to clear any effects of previously executed instructions. The external flags and processor state registers are usually cleared too.

The reset instruction is mainly used for debugging purposes, as most processors halt operations immediately or within a few cycles of receiving the reset instruction. The processors state may then be examined in its halted state.

Some processors have two types of reset operations. Soft reset implies initializing PC and interrupt flags. Hard reset initializes other processor state registers in addition to PC and interrupts enable flags. The software reset instruction asserts the external reset pin of the processor.

Reset operation in SRC

Hard Reset
The SRC should perform a hard reset upon receiving a start (Strt) signal. This initializes the PC and the general registers.

Soft Reset

The SRC should perform a soft reset upon receiving a reset (rst) signal. The soft reset results in initialization of PC only.

The reset signal in SRC is assumed to be external and asynchronous.

PC Initialization

There are basically two approaches to initialize a PC.

1. Direct Approach

The PC is loaded with the address of the startup routine upon resetting.

2. Indirect Approach

The PC is initialized with the address where the address of the startup routine is located. The reset instruction loads the PC with the address of a jump instruction. The jump instruction in turn contains the address of the required routine.

An example of a reset operation is found in the 8086 processor. Upon receiving the reset instruction the 8086 initializes its PC with the address FFFF0H. This memory location contains a jump instruction to the bootstrap loader program. This program provides the system initialization

Behavioral RTL for SRC Reset

The original behavioral RTL for SRC without any reset operation is:

```
Instruction_Fetch :=(! Run&Strt: (Run ← 1; instruction_Fetch,
                    Run : (IR ← M [PC]; PC ← PC+4;instruction_execution)),
                    instruction_execution:= (ld (:=op=1…) ;
```

This recursive definition implies that each instruction at the address supplied by PC is executed.

The modified RTL after adding the reset capability is

```
Instruction_Fetch:=(! Run&Strt :( Run ← 1,
                    PC, R [0...31] ← 0),
                    Run&!Rst :( IR ← M [PC],
                    PC ← PC+4, instruction_execution);
                    Run&Rst:( Rst ← 0, PC ← 0);
                    instruction_Fetch),
```

The modified definition includes testing the value of the "rst" signal after execution of each instruction. The processor may not be halted in the midst of an instruction *in the RTL definition*

To actually implement these changes in the SRC, the following modification are required to add the reset operation to the structural RTL for SRC:

- A check for the reset signal on each clock cycle
- A control signal for clearing the PC
- A control signal to load zero to control step counter

Example: The sub instruction with RESET processing

To actually reset the processor in the midst of an instruction, the "Rst" condition must be tested after each clock cycle.

Let us examine the contents of each phase in the given table. In step T0, if the Rst signal is not asserted, the address of the new instruction is delivered to memory and the value of PC is incremented by 4 and stored in another register. If the "Rst" signal is asserted, the "Rst" signal is immediately cleared, the PC is cleared to zero and T, the step counter is also set to zero. This behavior (in case of 'Rst' assertion) is the same for all steps. In step T1, if the rst signal is not asserted, the value stored at the delivered memory word is stored in the memory data register and the PC is set to its incremented value.

In step T2, the stored memory data is transferred to the instruction register.

In step T3, the register operand values are read.

In step T4, the mathematical operation is executed.
In step T5, the calculated value is written back to register file.

During all these steps if the Rst signal is asserted, the value of PC is set to 0 and the value of the step counter is also set to zero.

Step	RTN	Control Sequence
T0	!Rst:(MA ← PC,C ← PC+4), Rst:(Rst ← 0,PC ← 0,T ← 0)	!Rst:(PC_{out}, LMAR, INC4, LC, MRead), Rst:(ClrPC, Goto0);
T1	!Rst:(MD ← M[MA]:PC ← C), Rst:(Rst ← 0:PC ← 0:T ← 0)	!Rst:(C_{out}, LPC, Wait), Rst :(ClrPC, Goto0);
T2	!Rst:(IR ← MD), Rst:(Rst ← 0: PC ← 0:T ← 0)	!Rst:(MBR_{out}, LIR), Rst : (ClrPC, Goto0);
T3	!Rst:(A ← R[rb]), Rst:(Rst ← 0: PC ← 0: T ← 0)	!Rst:(RBE, R2BUS, LA), Rst : (ClrPC, Goto0);
T4	!Rst:(C ← A - R[rc]), Rst:(Rst ← 0: PC ← 0:T ← 0)	!Rst:(RCE, R2BUS, SUB, LC), Rst :(ClrPC, Goto0);
T5	!Rst:(R[ra] ← C), Rst:(Rst ← 0:PC ← 0: T ← 0)	!Rst:(LC, RAE, BUS2R, End), Rst : (ClrPC, Goto0);

Machine Exceptions

- Anything that interrupts the normal flow of execution of instructions in the processor is called an exception.
- Exceptions may be generated by an external or internal event such as a mouse click or an attempt to divide by zero etc.
- External exceptions or interrupts are generally asynchronous (do not depend on the system clock) while internal exceptions are synchronous (paced by internal clock)
- The exception process allows instruction flow to be modified, in response to internal or external events or anomalies. The normal sequence of execution is interrupted when an exception is thrown.

Exception Processing

A generalized exception handler should include the following mechanisms:

1. **Logic to resolve priority conflicts**. In case of nested exceptions or an exception occurring while another is being handled the processor must be able to decide which exception bears the higher priority so as to handle it first. For example, an exception raised by a timer interrupt might have a higher priority than keyboard input.
2. **Identification of interrupting device**. The processor must be able to identify the interrupting device that it can to load the appropriate exception handler routine. There are two basic approaches for managing this identification: exception vectors and

"information" register. The exception vector contains the address of the exception handling routine. The interrupting process fills the exception vector as soon as the interruption is acknowledged. The disadvantage of this approach is that a lot of space may be taken up by vectors and exception handler codes.

3. In the information register, only one general purpose exception handler is used. The PC is saved and the address of the general purpose register is loaded into the PC. The interrupting process must fill the information register with information to allow identification of the cause and type of exception.
4. **Saving the processor state**. As stated earlier the processor state must be saved before jumping to the exception handler routine. The state includes the current value of the PC, general purpose registers, condition vector and external flags.
5. **Exception disabling during critical operation**. The processor must disable interrupts while it is switching context from the interrupted process to the interrupting process, so that another exception might not disrupt the transition.

Examples of Exceptions

- Reset Exception
  Reset operation is treated as an exception by some machines e.g. SPARC and MC68000.
- Machine Check
  This is an external exception caused by memory failure
- Data Access Exception
  This exception is generated by memory management unit to protect against illegal accesses.
- Instruction Access Exception
  Similar to data access exception
- Alignment Exception
  Generated to block misaligned data access

Types of Exception

- **Program Exceptions**
  These are exceptions raised during the process of decoding and executing the instruction. Examples are illegal instruction, raised in response to executing an instruction which does not belong to the instruction set. Another example would be the privileged instruction exception.
- **Hardware Exceptions**
  There are various kinds of hardware exceptions. An example would be of a timer which raises an exception when it has counted down to zero.
- **Trace and debugging Exceptions**
  Variable trace and debugging is a tricky task. An easy approach to make it possible is through the use of traps. The exception handler which would be called after each instruction execution allows examination of the program variables.
- **Non-Maskable Exceptions**
  These are high priority exceptions reserved for events with catastrophic consequences such as power loss. These exceptions cannot be suppressed by the processor under any condition. In case of a power loss the processor might try to save the system state to the hard drive, or alert an alternate power supply.
- **Interrupts (External Exceptions)**
  Exception handlers may be written for external interrupts, thus allowing programs to respond to external events such as keyboard or mouse events.

Lecture No. 18

Pipelining

Reading Material

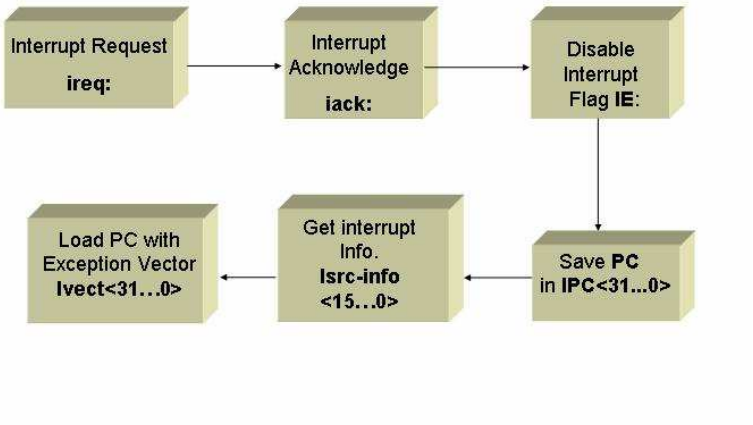
Vincent P. Heuring & Harry F. Jordan — Chapter 4
Computer Systems Design and Architecture — 4.8

Summary

- SRC Exception Processing Mechanism
- Introduction to Pipelining
- Complications Related to Pipelining
- Pipeline Design Requirements

Correction: Please note that the phrase "instruction fetch" should be used where the speaker has used "instruction interpretation".

SRC Exception Processing Mechanism



The following tables on the next few pages summarize the changes needed in the SRC description for including exceptions:

Behavioral RTL for Exception Processing

Instruction_Fetch:=

(!Run&Strt: Run ← 1,	Start
Run & !(ireq&IE):(IR ←M[PC],	Normal Fetch
PC ← PC + 4;	
Instruction_Execution),	
Run&(ireq&IE): (IPC ← PC<31..0>,	Interrupt, PC copied
II<15..0> ← Isrc_info<15..0>,	II is loaded with the info. PC loaded with
IE ← 0: PC ← Ivect<31..0>,	new address
iack ← 1; iack ← 0),	
Instruction_Fetch);	

Additional Instructions to Support Interrupts

Mnemonic	**Behavioral RTL**	**Meaning**
svi (op=16)	R[ra]<15..0> ← II<15..0>, R[rb] ← IPC<31..0>;	Save II and IPC
ri (op=17)	II<15..0> ← R[ra]<15..0>, IPC<31..0> ← R[rb];	Restore II and IPC
een (op=10)	IE ← 1;	Exception enable
edi (op=11)	IE ← 0;	Exception disable
rfi (op=30)	PC ← IPC, IE ← 1;	Return from interrupt

Structural RTL for the Fetch Phase including Exception Processing

Step	Structural RTL for the 1-bus SRC
T0	!(ireq&IE): (MA ← PC, C ← PC + 4); (ireq&IE): (IPC ← PC,II← Isrc_info, IE ← 0,PC ← (22α 0)©(Isrc_vect<7..0>)© 00, iack ← 1; iack ← 0, End) ;
T1	MD ← M[MA], PC ← C;
T2	IR ← MD;
T3	Instruction_Execution;

Combining the RTL for Reset and Exception Instruction_Fetch:=

RTL	Event
(Run&!Rst&!(ireq&IE):(IR ← M[PC], PC ← PC+4; Instruction_Execution),	Normal
Run&Rst: (Rst ←0 , IE ← 0, PC ← 0; Instruction_Fetch),	Fetch
!Run&Strt: (Run ←1, PC ← 0, R[0..31] ← 0; Instruction_Fetch),	Soft Reset
Run&!Rst&(ireq&IE): (IPC ← PC<31..0>,	Hard Reset
II<15..0> ←Isrc_info<15..0>, IE ← 0, PC ← Ivect<31..0>, iack ← 1; iack ← 0; Instruction_Fetch));	Interrupt

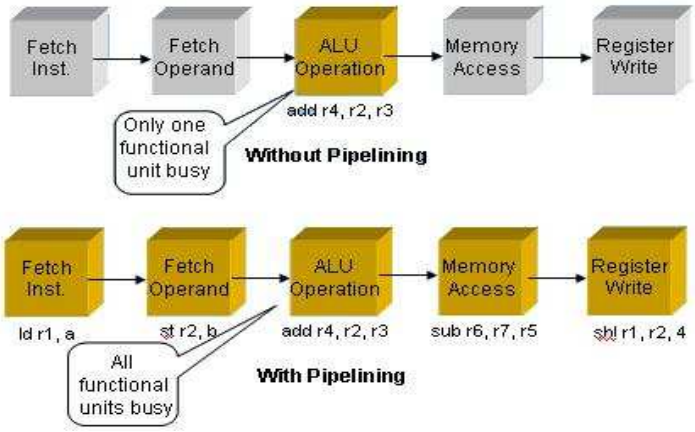
Introduction to Pipelining

Pipelining is a technique of overlapping multiple instructions in time. A pipelined processor issues a new instruction before the previous instruction completes. This results in a larger number of operations performed per unit of time. This approach also results in a more efficient usage of all the functional units present in the processor, hence leading to a higher overall throughput. As an example, many shorter integer instructions may be executed along with a

longer floating point multiply instruction, thus employing the floating point unit simultaneously with the integer unit.

Executing machine instructions with and without pipelining

We start by assuming that a given processor can be split in to five different stages as shown in the diagram below, and as explained later in this section. Each stage receives its input from the previous stage and provides its result to the next stage. It can be easily seen from the diagram that in case of a non-pipelined machine there is a single instruction **add r4, r2, r3** being processed at a given time, while in a pipelined machine, five different instructions are being processed simultaneously. An implied assumption in this case is that at the end of each stage, we have some sort of a storage place (like temporary registers) to hold the results of the present stage till they are used by the next stage.



Description of the Pipeline Stages

In the following paragraphs, we discuss the pipeline stages mentioned in the previous example.

1. Instruction fetch

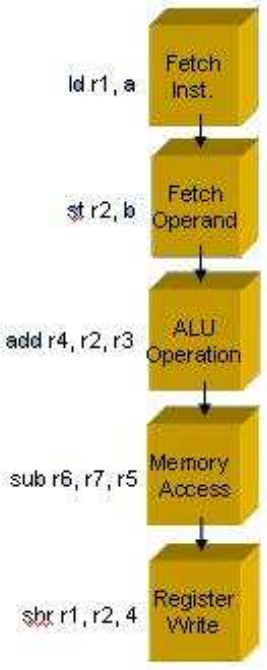
As the name implies, the instruction is fetched from the instruction memory in this stage. The fetched instruction bits are loaded into a temporary pipeline register.

2. Instruction decode/operand fetch

In this stage the operands for the instruction are fetched from the register file. If the instruction is **add r1, r2, r3** the registers r2 and r3 will be read into the temporary pipeline registers.

3. ALU[5] operation

In this stage, the fetched operand values are fed into the ALU along with the function which is required such as addition, subtraction, etc. The result is stored into temporary pipeline registers. In case of a memory access such as a load or a store instruction, the ALU calculates the effective memory address in this stage.



4. Memory access

For a load instruction, a memory read operation takes place. For a store instruction, a memory write operation is performed. If there is no memory access involved in the instruction, this stage is simply bypassed.

[5] The ALU is also called the ALSU in some cases, in particular, where its "shifting" capabilities need to be highlighted. ALSU stands for Arithmetic Logic Shift Unit.

5. Register write
The result is stored in the destination register in this stage.

Latency & throughput
Latency is defined as the time required to process a single instruction, while throughput is defined as the number of instructions processed per second. Pipelining cannot lower the latency of a single instruction; however, it does increase the throughput. With respect to the example discussed earlier, in a non-pipelined machine there would be one instruction processed after an average of 5 cycles, while in a pipelined machine, instructions are completed after each and every cycle (in the steady-state, of course!!!). Hence, the overall time required to execute the program is reduced.

Remember that the performance gain in a pipeline is limited by the slowest stage in the pipeline.

Complications Related to Pipelining
Certain complications may arise from pipelining a processor. They are explained below:

Data dependence
This refers to the situation when an instruction in one stage of the pipeline uses the results of an instruction in the previous stage. As an example let us consider the following two instructions

```
…
S1: add r3, r2, r1
S2: sub r4, r5, r3
…
```

There is a data-dependence among the above two instructions. The register R3 is being written to in the instruction S1, while it is being read from in the instruction S2. If the instruction S2 is executed before instruction S1 is completed, it would result in an incorrect value of R3 being used.

Resolving the dependency
There are two methods to remedy this situation:

1. Pipeline stalls
These are inserted into the pipeline to block instructions from entering the pipeline until some instructions in the later part of the pipeline have completed execution. Hence our modified code would become

```
…
S1: add r3, r2, r1
stall[6]
stall
stall
S2: sub r4, r5, r3
…
```

[6] A pipeline stall can be achieved by using the **nop** instruction.

2. Data forwarding

When using data forwarding, special hardware is added to the processor, which allows the results of a particular pipeline stage to be transferred directly to another stage in the pipeline where they are required. Data may be forwarded directly from the execute stage of one instruction to the decode stage of the next instruction. Considering the above example, S1 will be in the execute stage when S2 will be decoded. Using a comparator we can determine that the destination operand of S1 and source operand of S2 are the same. So, the result of S1 may be directly forwarded to the decode stage.

Other complications include the "branch delay" and the "load delay". These are explained below:

Branch delay

Branches can cause problems for pipelined processors. It is difficult to predict whether a branch will be taken or not before the branch condition is tested. Hence if we treat a branch instruction like any normal instruction, the instructions following the branch will be loaded in the stages following the stage which carries the branch instruction. If the branch is taken, then those instructions would need to be removed from the pipeline and their effects if any, will have to be undone. An alternate method is to introduce stalls, or **nop** instructions, after the branch instruction.

Load delay

Another problem surfaces when a value is loaded into a register and then immediately used in the next operation. Consider the following example:

…
S1: load r2, 34(r1)
S2: add r5, r2, r3
…

In the above code, the "correct" value of R2 will be available after the memory access stage in the instruction S1. Hence even with data forwarding a stall will need to be placed between S1 and S2, so that S2 fetches its operands only after the memory access for S1 has been made.

Pipeline Design Requirements

For a pipelined design, it is important that the overall meaning of the program remains unchanged, i.e., the program should produce the same results as it would produce on a non-pipelined machine. It is also preferred that the data and instruction memories are separate so that instructions may be fetched while the register values are being stored and/or loaded from data memory. There should be a single data path so as not to complicate the flow of instructions and maintain the order of program execution. There should be a three port register file so that if the register write and register read stages overlap, they can be performed in parallel, i.e., the two register operands may be read while the destination register may be written. The data should be latched in between each pipeline stage using temporary pipeline registers. Since the clock cycle depends on the slowest pipeline stage, the ALU operations must be able to complete quickly so that the cycle time is not increased for the rest of the pipeline.

Designing a pipelined implementation

In this section we will discuss the various steps involved in designing a pipeline. Broadly speaking they may be categorized into three parts:

1. Adapting the instructions to pipelined execution

The instruction set of a non-pipelined processor is generally different from that of a pipelined processor. The instructions in a pipelined processor should have clear and definite phases, e.g., **add r1, r2, r3**. To execute this instruction, the processor must first fetch it from memory, after which it would need to read the registers, after which the actual addition takes place followed by writing the results back to the destination register. Usually register-register architecture is adopted in the case of pipelined processors so that there are no complex instructions involving operands from both memory and registers. An instruction like **add r1, r2, a** would need to execute the memory access stage before the operands may be fed to the ALU. Such flexibility is not available in a pipelined architecture.

2. Designing the pipelined data path

Once a particular instruction set has been chosen, an appropriate data path needs to be designed for the processor. The data path is a specification of the steps that need to be followed to execute an instruction. Consider our two examples above

For the instruction **add r1, r2, r3**: *Instruction Fetch – Register Read – Execute – Register Write,*

Whereas for the instruction **add r1, r2, a** (remember a represents a memory address), we have
Instruction Fetch – Register Read –Memory Access– Execute – Register Write,

The data path is defined in terms of registers placed in between these stages. It specifies how the data will flow through these registers during the execution of an instruction. The data path becomes more complex if forwarding or bypassing mechanism is added to the processor.

3. Generating control signals

Control signals are required to regulate and direct the flow of data and instruction bits through the data path. Digital logic is required to generate these control signals.