CSCE230 Project Overview

This is the first handout outlining the project that you will carry out with your assigned. Recall from the syllabus that the group project carries a substantial portion (20%) of your grade for the course. The project is divided into multiple parts that must be accomplished sequentially. Each will have its own check-off date to demonstrate progress and completion, and each part will add to the already completed design.

This handout starts with a statement of objectives for the project, followed by an overview and specification of the processor. It ends with a preview of the different parts of the project (with their check-off dates) and an overview of the grading scheme. Appendix A provides overview and specification of the instructions and instruction formats and Appendix B reproduces the figures from the textbook referenced in this document.

Objectives

1) Understand software/hardware interface better by implementing a substantial subset of a Reduced Instruction Set Computer (RISC) instruction set architecture (ISA).

2) Understand design parameters that determine the performance of hardware design in terms of timing and utilization of hardware resources.

3) Learn to work as a team to carry out a complex design task requiring task partitioning, effective communication, and cooperation.

4) Produce written and oral reports that accurately describe your work.

Design Specifications Overview

You are to design and implement on the DE1 board a basic processor and optional extensions to it, following the scheme(s) described in Chapters 5 and 6 of the textbook. The ISA resembles a subset of the NIOS II architecture and includes some features that are unique to ARM. You can earn **bonus points** by implementing certain enhancements. Here is a top-level view of the processor architecture:

- All instructions are 24-bits wide and data is 16-bits wide.
- The processor communicates with the memory hierarchy through a processor-memory interface that will be provided to you. Further, you may use (or adapt) the datapath design of the processor shown in Figures 5.8 – 5.10 of the textbook.
- The instructions only support 16-bit data operands, i.e., there is no byte or bit-level addressing modes in the instruction set. Nor are there instructions for double-word or higher precision arithmetic.
- There are 16 registers in the register file; each is 16-bits wide. The register file has one write and two read ports and is compatible with the one you designed in Lab 9. We number the registers: $0 to $15. $0 is assumed to be constant 0. $15 is the return address register for subroutine linkage. $14 is the stack pointer.
- The processor uses *memory-mapped* I/O. As a minimum, you should implement polling based I/O and may implement interrupt I/O for extra credit. The basic I/O described in Chapter 3 of the textbook should be sufficient guide for designing the I/O.

- The processor may operate in two different modes:
  - User: Unprivileged mode under which most tasks run
  - Interrupt: Privileged mode entered when an external (I/O) interrupt occurs (extra credit)
- The program counter (PC) is not a part of the register file but a separate register, as it is in the MIPS architecture (but not ARM). The additional registers in the CPU include: PC (24-bit wide) and the processor status register PS (at least 9-bits wide). PS should have the following:
  - Four bits to store the N, Z, C, and V flags produced by the arithmetic logic unit (refer to Lab 10 for their definitions)
  - One bit to indicate the processor mode (user or interrupt) as described above
  - One reserved bit for the interrupt enable bit (IE). This bit will be utilized if your team chooses to implement interrupts
  - Three reserved bits for the current priority level (PRL). This bit will be utilized if your team chooses to implement prioritized interrupts.
- Borrowing a characteristic feature of ARM, every instruction executes conditionally based on the condition specified by the four flags in the PS register.
- Instruction types and instructions: In Appendix A, we specify a small subset that includes representative instructions from each class. Further, we may divide the set of instructions into a "must-implement" set and a set that can be implemented for bonus credit.
- If interrupts are implemented for bonus credit, the reserved interrupt enable bit (IE) in the PS should be utilized and additional processor control registers would need to be implemented as shown in Figure 3.7 of the textbook.

Project Parts and Grading

The project is divided into **seven** parts, of which the first should have been completed already as part of CSCE 230L Lab assignments (and hence carry no additional credit). Part VII allows you to earn extra credit. Documentation and presentation are an important part of this project, and thus your grade. In addition to your project files, every week you must submit a three-page report describing the portion of the project you worked on that week. These mini-reports should also include proof that your components/processor works. These weekly mini-reports should help you with the final report (and allow us to give you feedback as you work).

Here is a summary list of all the parts and (tentative) check-off dates, more detailed descriptions will follow in their respective weekly handouts:

Part I (Required). Build a 16x16 register file with one input and two output ports and a 16-bit ALU that produces the result plus the NZCV flags. **This part should have been completed already as part of CSCE 230L lab assignments and will not carry any additional grade for the project. Check-off by March 14th 2013.**

Part II. Integrate the register file and ALU with provided components into a basic datapath for single cycle execution of basic R-type operations (add, sub, and, or, xor) and demonstrate correct functionality through tests. Except for the two components you built in Part I, you will be provided with all other necessary components to connect the datapath and test. The mini-report should discuss the operations your processor can currently perform. **Check-off by March 28th**

Part III. Add to the datapath, the remaining R-type instructions sll, comp, jr; D-type instructions: lw, sw, addi; and B-type instructions b, and bal. The mini-report should follow like the previous week's report. **Check-off by April 4rd**

Part IV. Add I/O and ARM-like conditional execution of instructions (see links in Blackboard). The mini-report should discuss how you implemented I/O and conditional execution. **Check-off by April 11th**

Part V. Implement 5-stage pipelining. You will have two weeks for this part. The mini-report should discuss how you implemented the pipelined operation with a detailed timing analysis for different types of instructions. **Check-off by April 25th**

Part VI. Written report, oral presentation, and demonstration. **Written report due by May 2nd 11:59pm, oral presentations and demos will be scheduled during Lab's final exam duration, May 1st 1pm-3pm.**

***Part VII.* (Bonus)** Complete one or more of the following enhancements to the design for bonus credit **(Check-off date will be the same as Part VI)**. The amount of extra credit would vary, depending on the complexity of the enhancement(s) implemented. The difficulty level of each item is indicated within parenthesis.

- Add and demonstrate interrupt I/O implementation (medium to difficult)
- Add and demonstrate prioritized I/O implementation (difficult)
- Add and demonstrate J-type instructions: j, jal, and li (load immediate) (easy to medium)
- Add and demonstrate any other feature of your choice. For this choice you need to get it approved by the instructor first. (easy to difficult)

Grading: The project is worth a total of 500 points that determine 20% of your course grade. The bonus points can earn you an additional 150 points. Here is the distribution of points:

Parts II-IV: 50 points each. These are awarded at the time of check-off. Details on what you need to do for each check-off will appear in the weekly handouts.

Part V: 100 points.

Part VI: 200 points, divided among final written report, oral team presentation, and successful demonstration:

100 points: Report

This should be at least 6 pages long (double spaced) NOT including figures, tables, etc. A majority of the report should describe how your processor was implemented. This portion should include waveform verifications showing how each portion works independently and how the processor works as a whole. Discussions should be more detailed than how each instruction was implemented. A small portion of the report should be devoted to sharing your experiences with the project (novel achievement, pitfalls, frustrations, etc).

Use sections, tables and figures effectively.

Do not narrate your report. (Do not say, "first we did X, then Y", etc, rather you should write, "Part X accomplishes task T by…").

You must discuss each component/portion of the processor to receive credit for it.

50 points: Presentation

This should be a presentation, which summarizes the interesting portions of your report and project as well as what you accomplished.

50 points: Demonstration

This should be a presentation that demonstrates working, interesting program(s). (Just running through each instruction is not interesting; gnome sort is interesting). All (most) instructions should be used.

You may integrate the demonstration with the presentation; more details on timing will be provided later.

Individual: 50 points will be dedicated to individual grading. This grade will be based on our assessment of each member's participation in the project (5 pts/week) and the results from the **peer evaluation form** that will be filled out by each member at the end of the project (25 pts), commenting on each other member's contributions to the project.

Part VII: Up to 150 bonus points depending on which extras you implement.

Appendix A: Instruction Formats

The basic instructions used in this processor are of four types (R, D, B, and J) as shown below. In the following, a subset of the instructions defined for the processor is shown. The list may change in minor ways as we gain more experience with the implementation of this ISA.

Note: The following ISA leverages advantages of various existing ISAs. More specifically, most instructions resemble MIPS instructions but properties from ARM instructions are also used. More information about these ISAs will be provided in Blackboard under Course Documents/Project Files.

A general structure of the instructions is as follows:

2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
3	2	1	0	9	8	7	6	5	4	3	2	1	0										
op				Cond				rs				rt				rd				opx			S

(R) Register-Register: Arithmetic, logic, shift, and compare instructions, plus jump-register.

2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
3	2	1	0	9	8	7	6	5	4	3	2	1	0										
1	x	1	x	Cond				rs				rt				rd				opx			S
op																							

(D) Data Transfer (Between memory and CPU): load and store instructions; also add/sub immediate

2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
3	2	1	0	9	8	7	6	5	4	3	2	1	0										
1	x	0	x	Cond				rs				rt				Imm							S
op																							

(B) Branch and Branch-and-Link: Branches are PC relative but conditionally executed as in ARM. The immediate value in the Label field does not need to be sign extended as it is already 16 bits.

2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
3	2	1	0	9	8	7	6	5	4	3	2	1	0										
0	L	0	x	Cond				Label															
op																							

(J) Jump and Jump-and-link: also load immediate to register, in which case bits 23-20 specify rd.

2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
3	2	1	0	9	8	7	6	5	4	3	2	1	0										
0	L	1	x	Const																			
op																							

R-type Instructions

These instructions include: add, sub, and, or, xor, sll, cmp, jr.

Instruction	add	sub	and	or	xor	sll	cmp	Jr
Op (in hex)	A	A	A	A	A	B	E	F
Opx (in hex)	4	0	2	3	1	0	0	0

All of these instructions, **except for jr**, set flags (N, Z, C, V).

Addition: *add*

Perform signed addition on $rs and $rt then store the result in $rd.

$rd = $rs + $rt

Ex:
add $r1 $r2 $r3
$r1 = $r2 + $r3

Subtraction: *sub*

Perform signed subtraction on $rs and $rt then store the result in $rd.

$rd = $rs - $rt

Ex:
sub $r1 $r2 $r3
$r1 = $r2 - $r3

Logical: *and, or, xor*

Perform bitwise logical AND, OR or XOR operation on $rs and $rt then store the result in $rd

$rd = $rs & $rt

Ex:
and $r1 $r2 $r3
$r1 = $r2 & $r3

Logical Shift Left: *sll*

Shift $rs by the value in the field $rt then store the results in $rd

$rd = $rs << $rt

Ex:
sll $r1 $r2 $r3
$r1 = $r2 << $r3

Compare: *cmp*

Compares two register values. The condition flags are updated based on subtracting $rt from $rs, so that subsequent instructions (e.g., branch) can be conditionally executed.

Update N, Z, C, V based on $rs - $rt

The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned overflow) and a signed overflow, respectively.

Ex:
cmp $r2 $r3

Jump Register: *jr*

Jump to the instruction address stored in $rs (the set bit should be ignored)

PC = $rs

Ex:
jr $r1
PC = $r1

D-Type Instructions

These instructions include: lw, sw, addi, and set interrupt

Instruction	lw	sw	addi	si
Op (in hex)	8	9	C	D

All of these instructions set flags (N, Z, C, V).

Load Word: *lw*

Load the contents of the memory address formed by $rs + Sign_extend (Imm) into $rt

$rt =Data_Memory[Sign_extend(Immediate) + $rs]

Ex:
lw $r1 (-4)$r2
$r1 = Data_Memory[$r2 + (-4)]

Store Word: *sw*

Store the contents of $rt into the memory address formed by $rs + Sign_extend(Immediate)

Data_Memory[Sign_extend(Immediate) + $rs] = $rt

Ex:
sw $r1 (-4)$r2
Data_Memory[$r2 + (-4)] = $r1

Add Immediate: *addi*

Add $rs to Sign_extend(Imm) then store the result in $rt

$rt = Sign_extend(Imm) + $rs

Ex:
addi $r1 $r2 -4
$r1 = $r2 + (-4)

Set Interrupt: *si* (Optional, required if you implement interrupts)

Sets the interrupt based on the Immediate value:

0 = Disable Interrupts 1 = Enable Interrupts 2 = Toggle Interrupt

Ex:
si 1

B-Type Instructions

These instructions include: b, bal.

Instruction	b	bal
Op (in hex)	0	4

The ‘L’ bit (bit 1) specifies whether the instruction should link or not.

These instructions **do not** set the condition code flags (N, Z, C, and V).

Branch: *b*

Branch to the instruction address formed by PC+2 + Label

PC = PC + 1 + Label

Ex:
b -34
PC = PC + 1 + (-34)

Branch And Link: *bal*

Branch to the instruction address formed by PC+1 + Label and store the next instruction address (PC+1) in $r15

$r15 = PC + 1
PC = PC + 1 + Label

Ex:
bal -34
$r15 = PC + 1
PC = PC + 1 + (-34)

J-Type Instructions

These instructions include: j, jal, li.

Instruction	j	jal	li
Op (in hex)	2	6	3

The ‘L’ bit (bit 1) specifies whether the instruction should link or not.

These instructions **do not** set the condition code flags (N, Z, C, and V).

Jump: *j*

Jump to the instruction address in Const

PC = Const

Ex:
b 256
PC = 256

Jump And Link: *jal*

Jump to the instruction address in Const and store the next instruction address (PC+1) in $r15

$r15 = PC + 1
PC = Const

Ex:
jal 256
$r15 = PC + 1
PC = 256

Load Immediate: *li*

Load the 16-bit constant into the register specified by bits 23-20 ($rd)

$rt = Const (bits 19-4)

Ex:
li $r1 22
$r1 = 22

Conditional Execution of Instruction – ARM-like extension

The R, D, and B type instructions implement a condition code. The instruction will only execute if, based on the given condition code, the corresponding condition flag(s) are set. Please see the 'Set Bit' section for more information about setting the flags.

Conditional execution is an efficient way of generating various instructions from a basic instruction. The most common example is branch. Depending on the condition codes used, b (branch) can be beq, bne, bgt, etc…

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				Cond																			

You can view the condition code as additional bits of the op Code that defines distinct functions. However, in implementation, the directly use the condition flags (N, Z, C, V) generated from ALU.

Due to its nature, conditional execution depends on the previous instruction that sets the condition flags, i.e., Branch_if_[R2=R3] LABEL will be implemented by TWO instructions. One compares R2 and R3, the other branches if they are equal (beq).

Each condition code combination is also labeled by a two character string. These can be appended to the instruction in assembly, e.g., beq means branch if Z is set.

0000	=	EQ	(equal)	-	Z set
0001	=	NE	(not equal)	-	Z clear
0010	=	CS	(unsigned higher or same)	-	C set
0011	=	CC	(unsigned lower)	-	C clear
0100	=	MI	(negative)	-	N set
0101	=	PL	(positive or zero)	-	N clear
0110	=	VS	(overflow)	-	V set
0111	=	VC	(no overflow)	-	V clear
1000	=	HI	(unsigned higher)	-	C set and Z clear
1001	=	LS	(unsigned lower or same)	-	C clear or Z set
1010	=	GE	(greater or equal)	-	N set and V set, or N clear and V clear
1011	=	LT	(less than)	-	N set and V clear, or N clear and V set
1100	=	GT	(greater than)	-	Z clear, and either N set and Vset, or N clear and V clear
1101	=	LE	(less than or equal)	-	Z set, or N set and V clear, or N clear and V set
1101	=	AL		-	always
1111	=	NV		-	never

To use a condition code, append the corresponding two character code to the assembling instruction. For example, to execute a register-register addition instruction always, the assembly instruction would be:

addal $r1 $r2 $r3

This always adds $r2 to $r3 and stores the result in $r1

To branch if not equal, the assembly instruction would be:

bne 256

Thus, after execution of this instruction, if the Z flag was cleared, the program counter will be updated with an offset of 256.

To use condition codes, an instruction needs to be executed that sets/clears the desired condition flag(s) (see the 'Set Bit' Section for more information), and then another instruction tests the flag(s) and performs an operation.

See the table on the next page.

Set Bit

The R and D type instructions have an 'S' bit (bit 0) that indicates whether the given instruction should test the condition flags or update them.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																							S

If the 'S' bit is set (bit 0 is 1), then the given instruction will update the condition flag(s) based on the result of the instruction operation. To set the 'S' bit, append an 's' delimited with a space to the end of the assembly instruction.

If the 'S' bit is not set, the previous values continue to exists in the flag register.

For example, to always execute an addition instruction that updates the 'Z' flag, an assembly instruction would be:

addal $r0 $r0 $r0 s

This instruction has an 'al' condition code meaning it always executes. Since the result of this operation is always zero, the 'Z' flag will be set (Note that $0 is never written to).

Appendix B: Figures of the Datapath from the Textbook

Fig. 5.1 Main Hardware Components

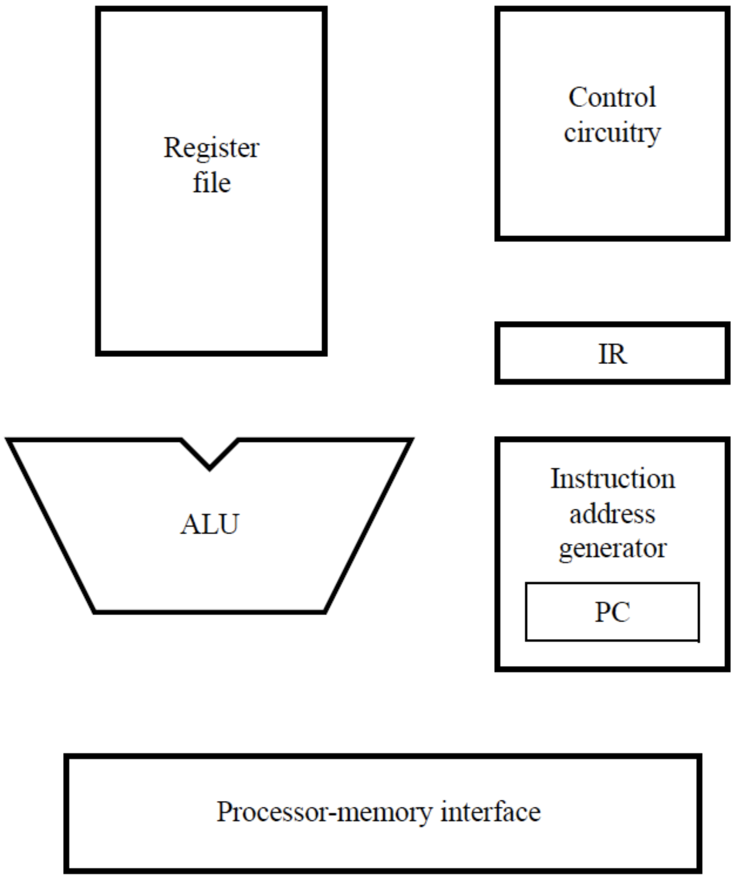


Fig. 5.8 Datapath

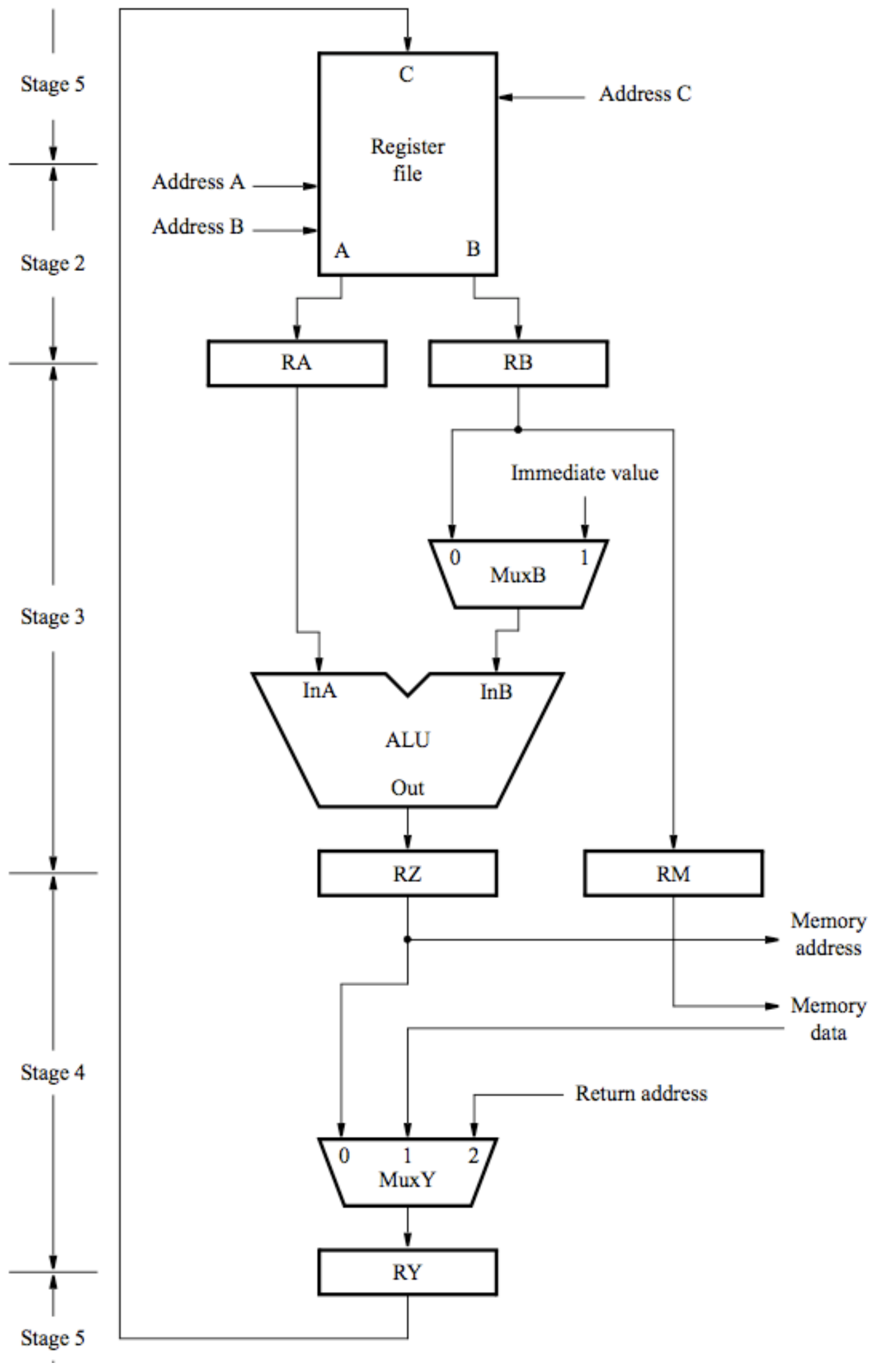


Fig. 5.9 Memory Address Generator

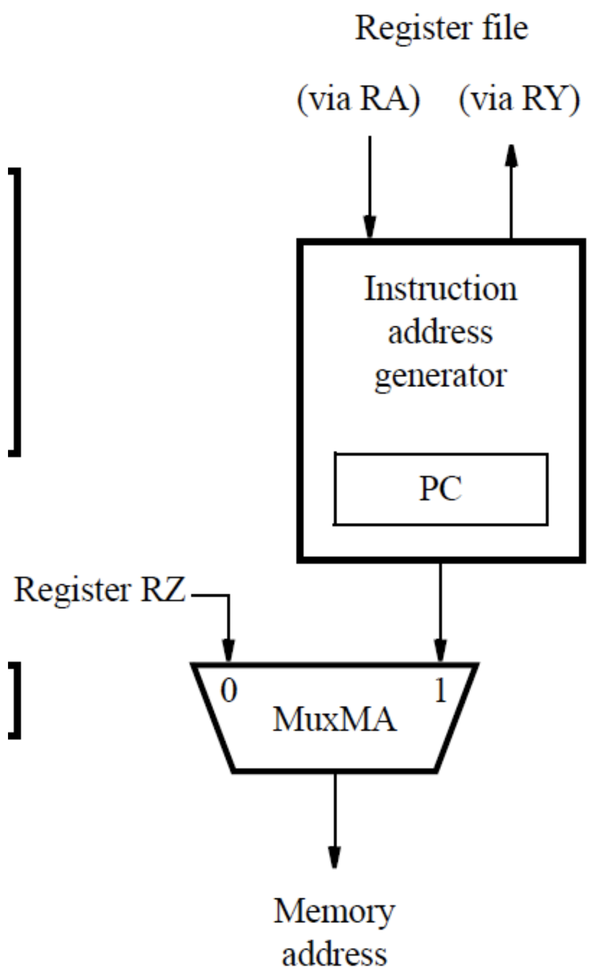


Fig. 5.9 Instruction Fetch and Decode

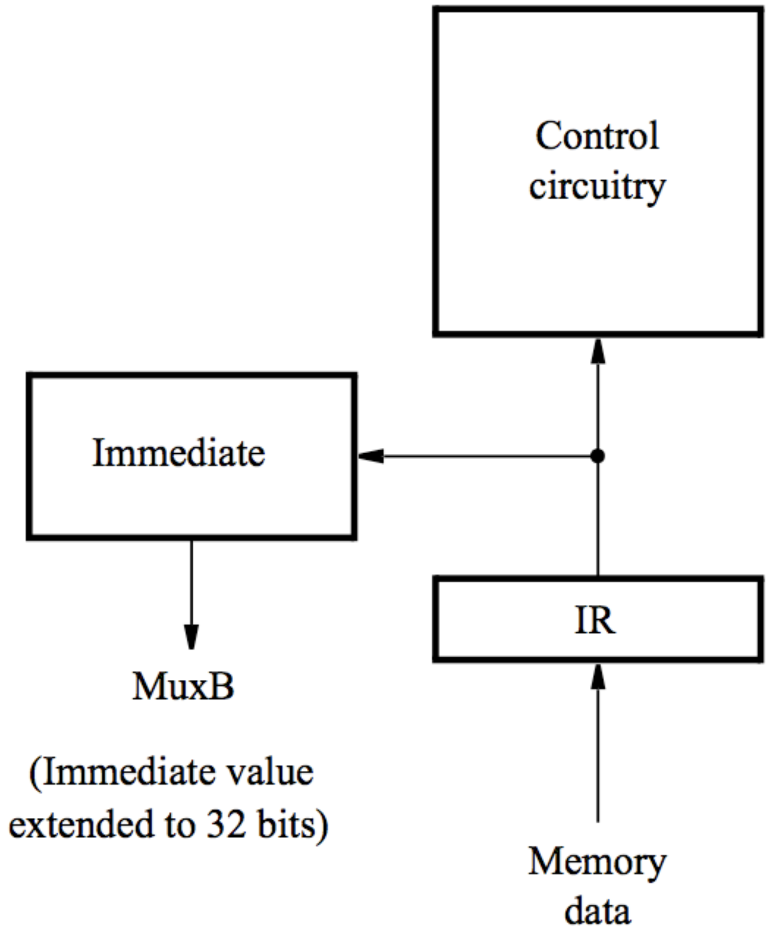
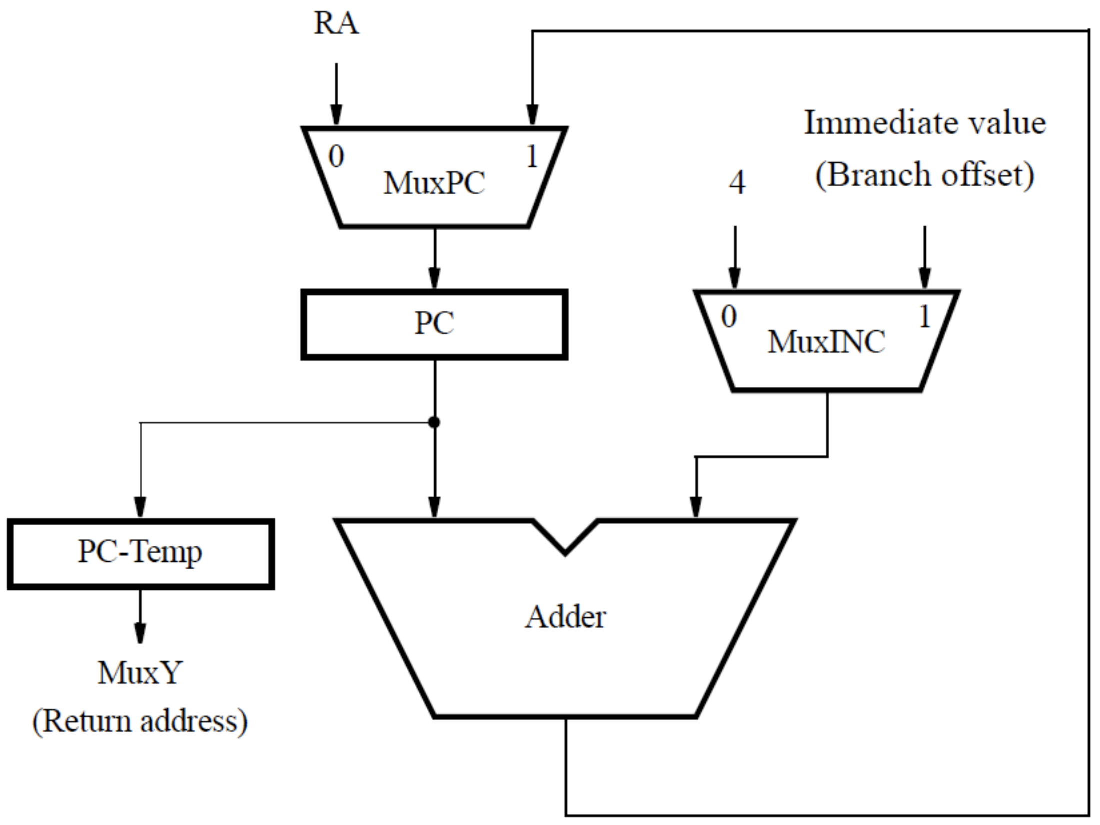


Fig. 5.10 Instruction Address Generator


RA
Immediate value
(Branch offset)
0
1
MuxPC
4
PC
0
1
MuxINC
PC-Temp
Adder
MuxY
(Return address)