CSCE 230 Project Overview

This document will provide an overview of the project. It will outline each part of the project. Recall from the syllabus that the group project carries a substantial portion (18%) of your grade for the course. The project is divided into multiple parts that will be completed sequentially. Each part will have a check off date to demonstrate that you are progressing, and each part will all add to the previous part.

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

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 Specification Overview:
You are going to design and implement a basic processor, with optional extensions to it, on the DE1 board following the scheme(s) described in Chapter 5 of the textbook. The ISA you will implement resembles a subset of the NIOS II architecture and includes some features that are unique to ARM.

- All instructions will be 24 bits wide

- The data will be 16 bits wide (16 bit registers)

- The processor will communicate with the memory hierarchy through a processor-memory interface that will be provided to you

- There are 16 registers in the register file and they are each 16 bits wide. The register file will have one write and two read ports and is designed as the one you created in Lab 8. The registers are numbered R0-R15, with R0 being a constant zero, R15 being the link register, and R14 being used as the stack pointer

- The processor uses memory mapped I/O. At a minimum you will implement polling based I/O with the following devices: push buttons, green leds, switches, and one of the Hex displays

- The program counter (PC) is not part of the register file, but a separate register, as it is in MIPS architecture (not ARM). The additional registers, past R0-R15, include the PC (16-bits wide) and the processor status register PS (at least 4-bits wide). PS has the following:

    - Four bits to store the N,Z,V,C flags that are produced by the ALU.
    - The following bits if you choose to implement interrupts
        * One bit to indicate processor mode (user or interrupt)
        * One bit for interrupts enable (IE)
        * The bits for the current priority level, if you choose to implement priority interrupts.

- By borrowing a characteristic of ARM, most instructions can execute conditionally based upon the condition specified and the values of the four flags in the PS

- You will have to implement a certain set of instructions, specified below. You may also choose to implement more instructions for possible bonus points.

Project Parts:

The project will be divided into five mandatory parts and one bonus part. You will already have completed the first part of the project in lab. Documentation and presentation are an important part of this project, and thus to your grade. In addition to your project files for each part, you must submit a two-page report describing the portion of the project you worked. **These reports should also include proof that your components/processor works**. These reports should help you get ready for the final report and will allow us to give you feedback before the final report based on your technical writing.

Below you will find a summary of each part, along with a due date. We will release an additional document for each part to further explain it. These additional documents will come with explanations, suggestions, and images of the work you are to do.

- Part I

  – Build a 16X16 bit register file with one input and two outputs ports, a 16-bit ALU that can produce the NZVC flags, and a control unit that will be able to execute basic R-Type instructions. **This part is already done in the lab.** Make sure you register file and ALU are thoroughly tested.

- Part II

  – Integrate the register file, ALU, control unit, and other components into a basic datapath for single cycle execution of basic R-Type operations (Add, Sub, And, Or, Xor) and demonstrate correct functionality with tests. For this part, you will not need a memory interface. You will need to create a few additional components, namely: IR, immediate extender, and various multiplexors and registers. You will be provided with the instruction address generator and memory interface in later parts. **Check off by November 11.**

- Part III

  – You will add to the basic processor you created in Part II. This part will require you to implement the remainder of the instructions (R-type, D-type, and B-type). **Check off by November 18.**

- Part IV

  – This part will addon basic I/O and ARM-like conditional execution. **Check off by December 2.**

- Part V

  – This stage will involve adding more I/O devices to your processor and implementing the J-type instructions. **Check off by December 9.**

- Part VI

  – The final mandatory part of the project is a written report, oral presentation, and a demonstration. **The written report will be due December 12 at 11:59PM. The presentation and demonstration will take place during the labs' final exam times.**

- Part VII (Bonus)

  – You are able to earn bonus points for your project. You must let a TA know your team is doing something for bonus by December 5. A few possible bonus items include:
    * Add and demonstrate interrupt I/O implementation
    * Add and demonstrate prioritized I/O interrupt implementation
    * Add and demonstrate additional R-type instructions
    * Add and demonstrate pipelined processor execution
    * Build an assembler for your processor
    * Add and demonstrate any other feature of your choice. You will need to get approval before you start this from a TA (Michael/Andrew)

Grading:

The project is worth a total of 600 points that determines 18% of your course grade. You can earn up to 150 bonus points. Here is the distribution of points:
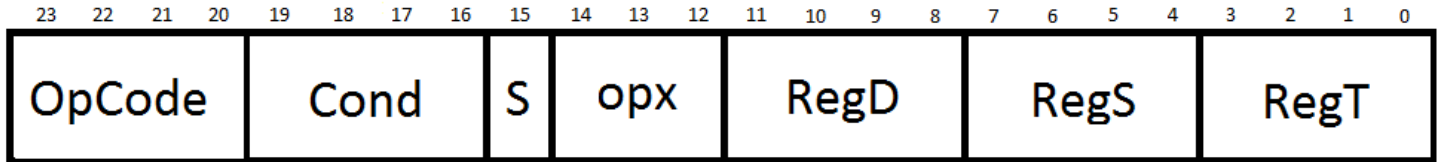
- Control Unit: 25 points. This is for completion of Lab 9 where you created the control unit.

- Part II - V: 75 points each. A breakdown of each parts points will be included in the document handed out with it. It will include points for the report, the check off, and the actual design

- Part VI: 200 points. This will be divided among the final report, the presentation, and the demonstration:

  - **100 points: Report**

    This should be at least 4 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**

    The presentation should summarize the interesting portions of your report and project as well as what you accomplished. A rubric will be handed around the time Part V starts.

  - **50 points: Demonstration**

    This demonstration should showcase an interesting program running on your board. The demonstration will take place at the end of your presentation. Most instructions should be used for this, and any bonus will have to be demonstrated to receive credit for it.

- Individual: 75 points. This grade will be based on our assessment of each members participation in the project and the results from the **peer evaluation form** that your group members will fill out.

- Part VI: Up to 150 bonus points. Reaching 150 bonus points will be extremely difficult, and the average given is 40 points.
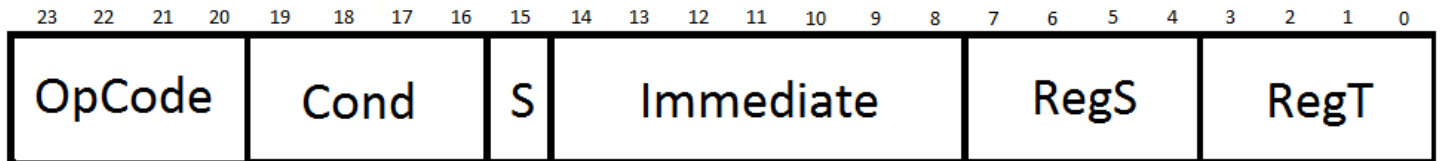
Instruction Formats:

The basic instructions used in the processor you will make are of four types (R,D,B,J). In the following, a subset of the instructions defined for the processor are shown.

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.

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

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

(D) Data transfer (between memory and CPU) types: load, store, and add/sub immediate

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

(B) Branch types: Branch and branch and link instructions. Branches are PC relative but can be conditionally executed as in ARM. The immediate value will not need to be sign extended as it is already 16 bits.

23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Cond	Label
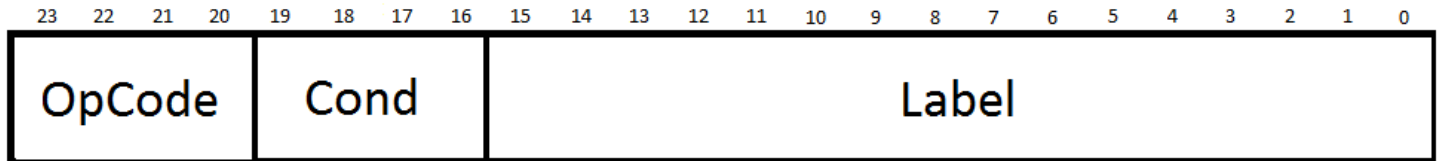
(J) Jump types: Jump, jump and link, and load immediate instructions. Bits 19-16 can specify rd for load immediate.

23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Constant
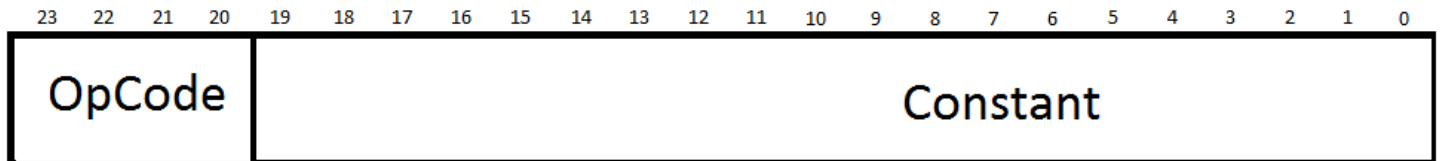
R-Type:

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

Instruction	Add	Sub	AND	OR	XOR	sll	cmp	jr
OpCode(in binary)	0000	0000	0000	0000	0000	0011	0010	0001
Opx(in binary)	100	011	111	110	101	000	000	000

All of these instructions, except jr, can set the NCVZ flags.

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:

These instructions include load word, store word, add immediate, and set interrupt.

Instruction	lw	sw	addi	si
OpCode(in binary)	0100	0101	0110	0111

All of these instructions can set the NCVZ flags.

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:

These instructions include Branch and Branch and link.

Instruction	b	bal
OpCode(in binary)	1000	1001

These instructions do not set the NCVZ flags.

Branch: *b*

Branch to the instruction address formed by PC+ 1 + 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:

These instructions include jump, jump and link, and load immediate

Instruction	j	jal	li
OpCode(in binary)	1100	1101	1110

These instructions do not set the NCVZ flags.

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 instructions - ARM-like extension:

The R, D, and B type implement a condition code called Cond. When this is specified the instruction will only execute if the corresponding condition flag(s) is/are set. How to set these flags will be specified later.

Conditional execution is an efficient way of generating various instructions from a basic instructions. The most common example is branch. We can specify the COND part of the branch:

23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Cond	Label
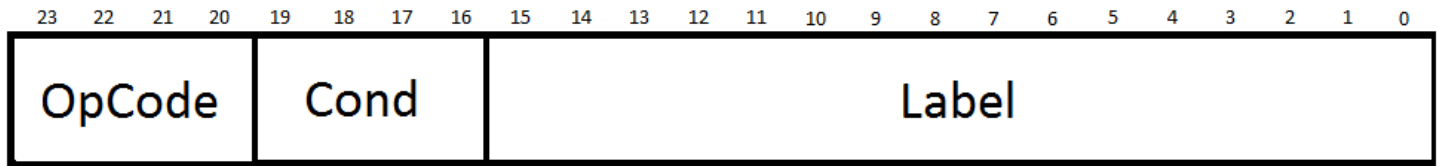
and have beq, bne, bgt, blt, etc...

You can view the condition code as an extension of the opCode that defines distinct instruction sub-types. Due to its nature, conditional execution depends on the previous instructions that sets the condition flags, i.e., Branch_if_[R2-R3] LABEL would be implemented by TWO instructions. The first will compare R2 and R3 and set the NCVZ flags and the second instruction will branch if the Z flag is set.

Each condition code combination is also labeled by a two character string. These can be appended to the instrcution in assembly, e.g., beq means branch if Z is set. The below table shows the possible values of the Cond bits and what they mean:

Cond bits	Char string	Meaning	Flags
0000	AL	Always	
0001	NV	Never	
0010	EQ	Equal	Z
0011	NE	Not Equal	NOT Z
0100	VS	Overflow	V
0101	VC	No Overflow	NOT V
0110	MI	Negative	N
0111	PL	Postive or Zero	NOT N
1000	CS	Unsigned higher or same	C
1001	CC	Unsigned lower	NOT C
1010	HI	Unsigned Higher	C AND (NOT Z)
1011	LS	Unsigned Lower or same	(NOT C) OR Z
1100	GT	Greather than	(NOT Z) AND ((N AND V) OR ((NOT N) AND (NOT V)))
1101	LT	Less than	(N AND (NOT V)) OR ((NOT Z) AND V)
1110	GE	Greater than or equal	(N AND V) OR ((NOT N) AND (NOT V))
1111	LE	Less than or equal	Z OR (((N AND (NOT V)) OR ((NOT Z) AND V)))

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

```
addal R1, R2, R3
```

This will always add R2 + R3 and store that into R1.

To branch if R2 is not equal to R3, this code would look like:
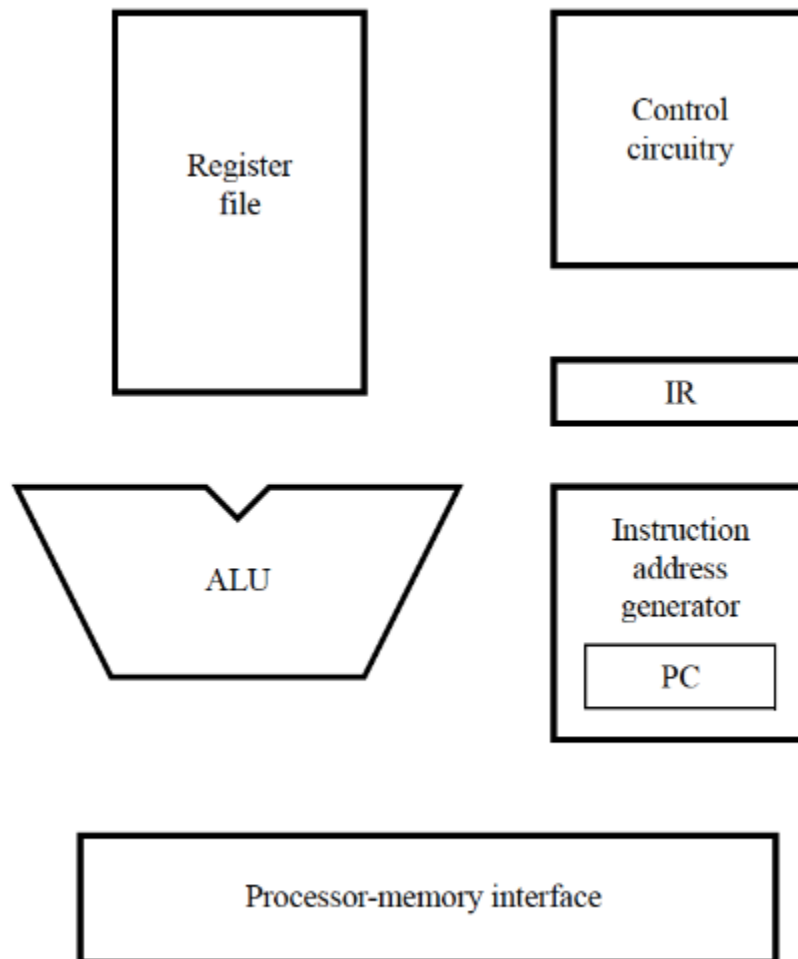
```
cmp R2, R3
bne LABEL
```

The cmp instruction will compare R2 and R3 and set the flags based on R2 - R3. If the Z flag is zero, then the branch will modify the PC so that it is now at LABEL. The cmp instruction uses the S bit of the instruction to set the flags.
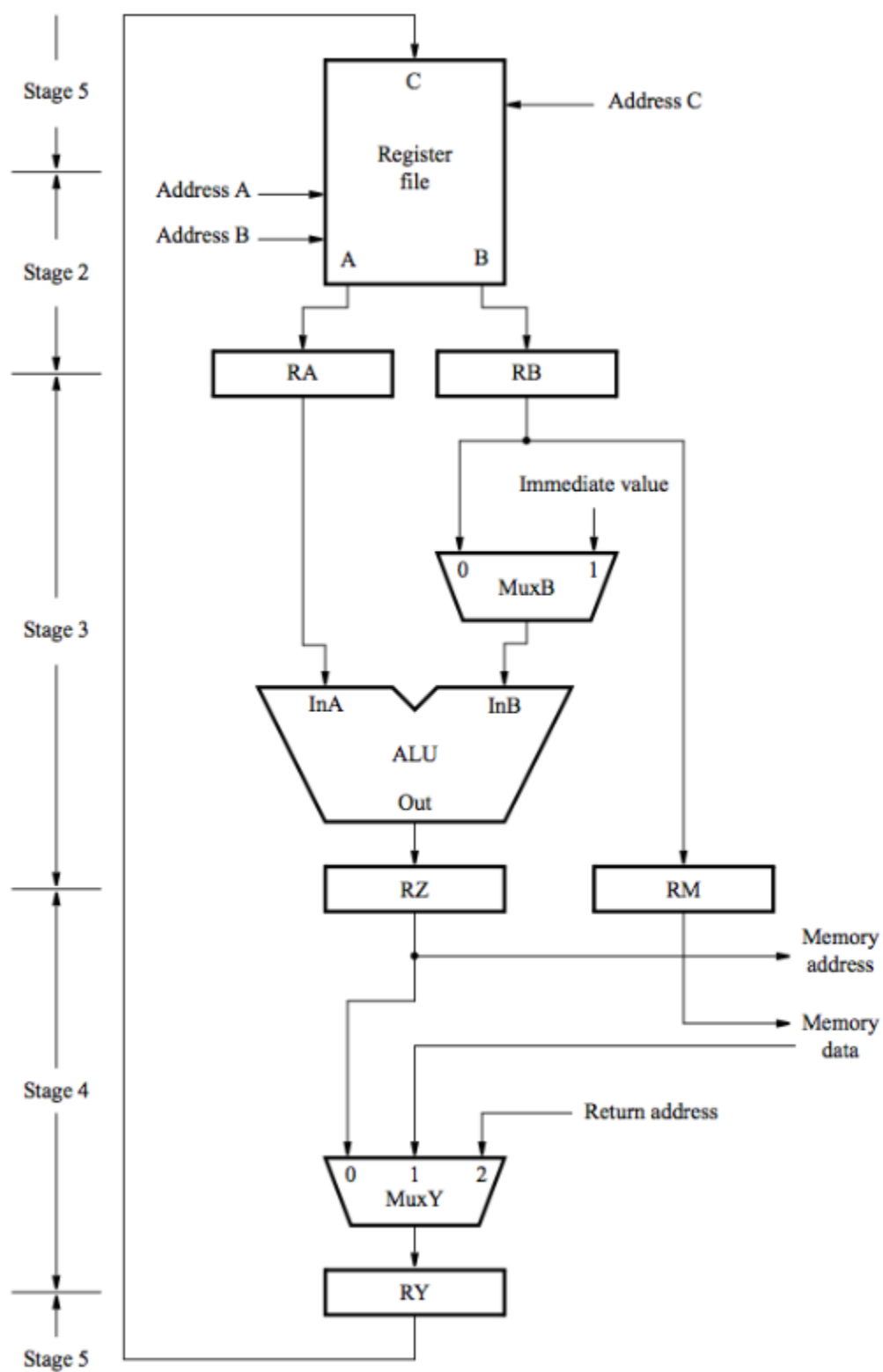
Set Bit:

The R and D type instructions have an 'S' bit (bit 15) that indicates whether the PS register should update its value of the flags. If the S bit is set then the given instruction will update the condition flags, NCVZ, based on the result of the instruction operation. Append an S to an instruction in order for it to set the flags (cmp should always set the flags but other instructions can as well).
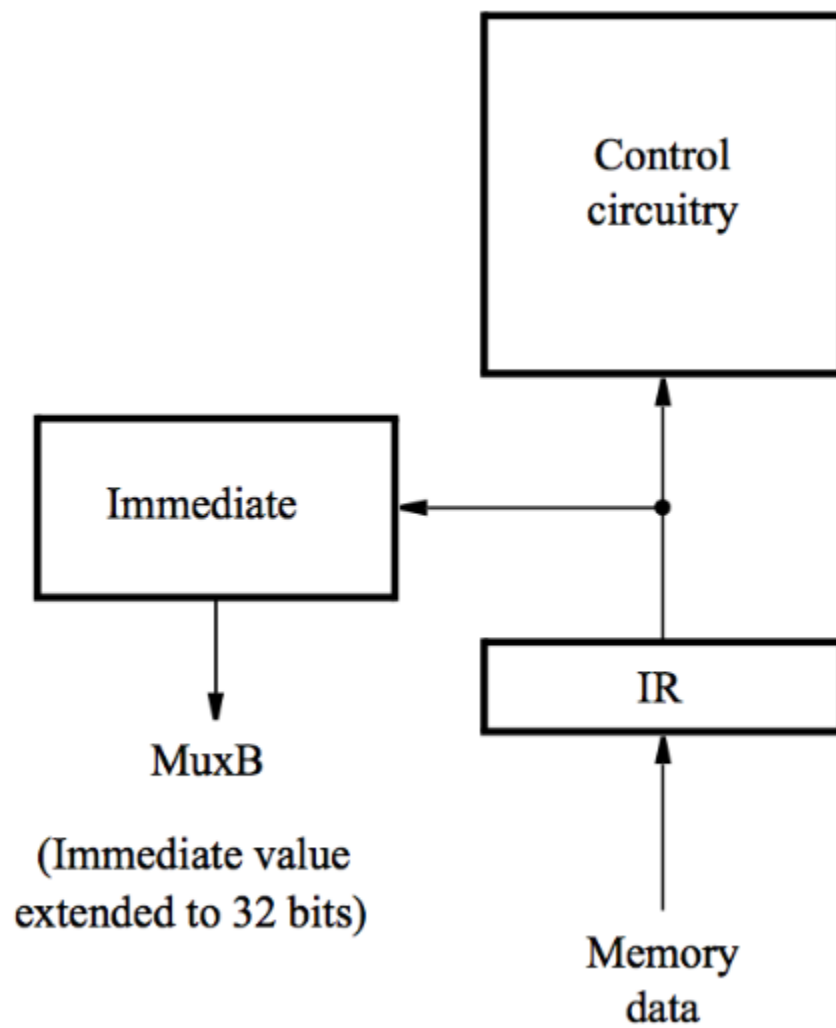
Figures of the processor from the book:

The main hardware

Register
file

Control
circuitry

IR

ALU

Instruction
address
generator

PC

Processor-memory interface

The datapath

Stage 5

Address C

Register file

Address A

Address B

A   B

Stage 2

RA   RB

Immediate value

0   MuxB   1

Stage 3

InA   InB

ALU

Out

RZ   RM

Memory address

Memory data

Stage 4

Return address

0   1   2

MuxY

RY

Stage 5

Control
circuitry

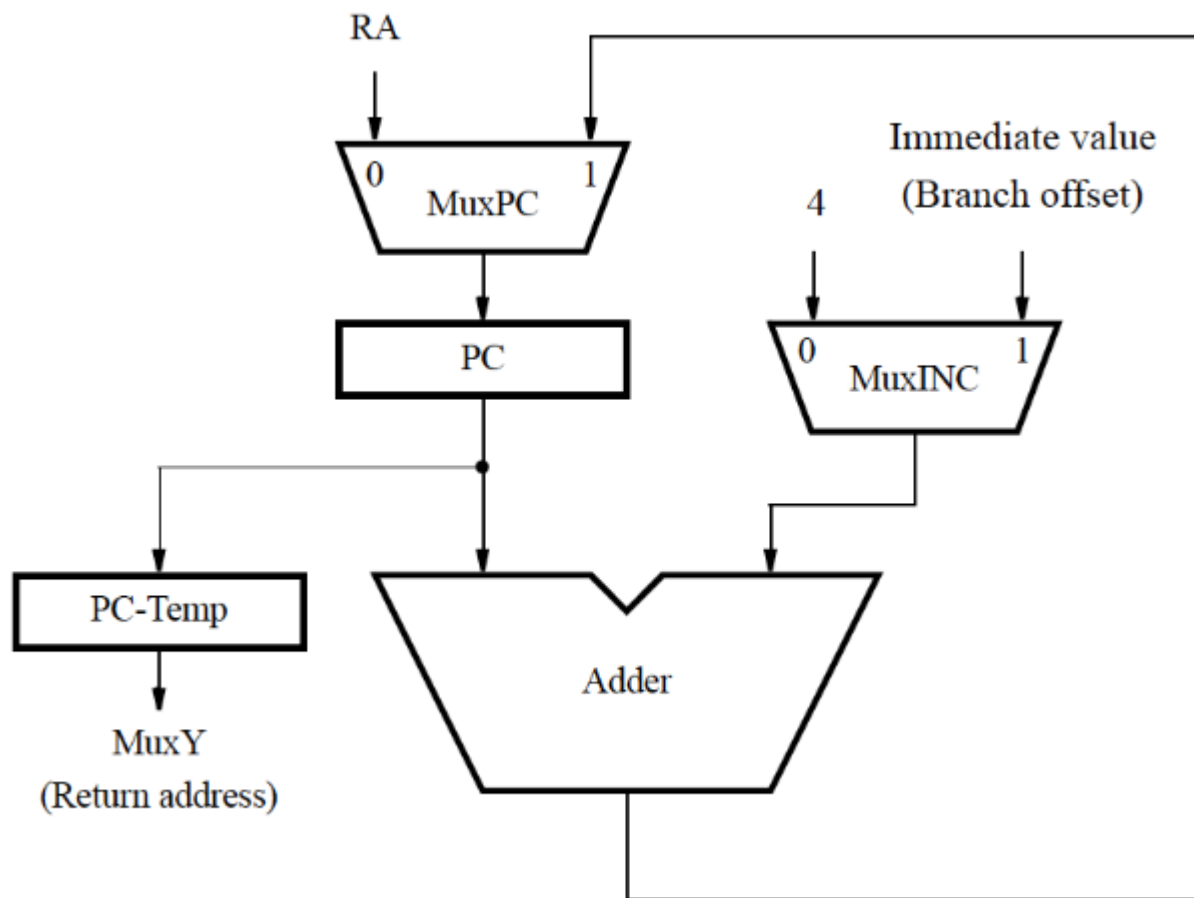Immediate

MuxB

(Immediate value
extended to 32 bits)

IR

Memory
data

The Instruction Address Generator

The Memory Address Generator

Register file

(via RA)   (via RY)

Instruction
address
generator

PC

Register RZ

0   1
     MuxMA

Memory
address