CSCE 230 Honors Project Specification

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 32 bits wide
- The data will be 32 bits wide (32 bit registers)
- The processor will communicate with the memory hierarchy through a processor-memory interface that will be provided to you
- There are 32 registers in the register file and they are each 32 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-R31, with R0 being a constant zero, R30 being the link register, and R31 being used as the stack pointer
- The processor uses memory mapped I/O. At a minimum you will implement polling based I/O with all of the basic IO pins: Red/Green leds, 4 hex displays, push buttons, and switches
- 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-R31, include the PC (32-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.

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, compare, jump register, and more

RegS					RegT					RegD				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

OPX							S	COND				OP				
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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

RegS					RegD					Immediate				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

Immediate							S	COND				OP				
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(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.

Immediate														
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

Immediate								COND				OP				
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(J) Jump types: Jump, jump and link, and load immediate instructions.

Immediate														
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

Immediate												OP				
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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	0100	0101	0110
Opx(in binary)		0000001	0000011	0000010	0000001			

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

add

Instruction	add
Operation	$RegD \leftarrow RegS + RegT$
Assembler Syntax	add RegD, RegS, RegT
Description	Calculates the sum of RegS and RegT then stores the result in RegD.

sub

Instruction	sub
Operation	$RegD \leftarrow RegS - RegT$
Assembler Syntax	sub RegD, RegS, RegT
Description	Calculates the difference between RegS and RegT then stores the result in RegD.

AND

Instruction	AND
Operation	$RegD \leftarrow RegS * RegT$
Assembler Syntax	and RegD, RegS, RegT
Description	Calculates the value of (RegS and RegT) then stores the result in RegD.

OR

Instruction	OR
Operation	$RegD \leftarrow RegS + RegT$
Assembler Syntax	or RegD, RegS, RegT
Description	Calculates (RegS or RegT) then stores the result in RegD.

XOR

Instruction	XOR
Operation	$RegD \leftarrow RegS \bigoplus RegT$
Assembler Syntax	xor RegD, RegS, RegT
Description	Calculates RegS xor RegT then stores the result in RegD.

sll

Instruction	sll
Operation	$RegD \leftarrow RegS << (RegT)$
Assembler Syntax	sll RegD, RegS, RegT
Description	Calculates the slide left of RegS by RegT positions then stores the result in RegD.

cmp

Instruction	cmpXX
Operation	$RegD \leftarrow RegSXXRegT$
Assembler Syntax	cmpXX RegD, RegS, RegT
Description	Calculates the indicated comparison (XX) of RegS with RegT then stores the result in RegD.

jr

Instruction	jr
Operation	$PC \leftarrow RegS$
Assembler Syntax	jr RegS
Description	Places the value of RegS into the PC register.

D-Type:

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

Instruction	lw	sw	addi	si
OpCode(in binary)	0111	1001	1000	1010

All of these instructions can set the NCVZ flags.

lw

Instruction	lw
Operation	$RegD \leftarrow \#IMM(RegS)$
Assembler Syntax	lw RegD, #IMM(RegS)
Description	Loads memory address stored in RegS and incremented by an immediate value into RegD.

sw

Instruction	sw
Operation	$\#IMM(RegD) \leftarrow RegS$
Assembler Syntax	sw RegS, #IMM(RegD)
Description	Stores the value of RegS into the memory location RegD incremented by an immediate value.

addi

Instruction	addi
Operation	$RegD \leftarrow RegS + \#IMM$
Assembler Syntax	addi RegD, RegS, #IMM
Description	Add RegS and #IMM and store into RegD.

si

Instruction	si
Operation	$IPS \leftarrow RegD$
Assembler Syntax	si RegS
Description	Places RegS into the interrupt status register.

B-Type:

These instructions include Branch and Branch and link.

Instruction	b	bal
OpCode(in binary)	1011	1100

These instructions do not set the NCVZ flags.

b

Instruction	b
Operation	$PC \leftarrow PC + 4 + \#IMM$
Assembler Syntax	bXX #conditional label
Description	Checks condition (XX) against conditional value and branches to label.

bal

Instruction	bal
Operation	$LR \leftarrow PC + 4 + \#IMM$ $PC \leftarrow PC + 4 + \#IMM$
Assembler Syntax	bal #label
Description	Branches to the label and stores current location into link register.

J-Type:

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

Instruction	j	jal	li
OpCode(in binary)	1101	1110	1111

j

Instruction	j
Operation	$PC \leftarrow$location of label
Assembler Syntax	j label
Description	Puts the location of label into the PC register.

jal

Instruction	jal
Operation	$LR \leftarrow PC + 4$ $PC \leftarrow$location of label
Assembler Syntax	jal label
Description	Puts the location of label into the PC register and stores the next command into the link register.

li

Instruction	li
Operation	$PC \leftarrow \#IMM$
Assembler Syntax	li #IMM
Description	Puts the immediate value into the PC register.

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:
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).