Computer Engineering department, Faculty of engineering, Cairo University

Computer Architecture Project

Fall 2013

Objective

To design and implement a simple CISC processor, Von Neumann or Harvard architecture. The design should conform to the ISA specification described in the following sections.

Main Components

You are required to design a simple 32-bit processor, general needed components are

- Assembly to machine code translator
- Register Table
- ALU
- Control unit
- two levels of memory
- cache controller

Bonus Addition

- Floating point arithmetic unit (IEEE or IBM format)

General Rules

- The processor required is a 32-bit processor.
- The VHDL code must be Synthesizable and pass the functional simulation correctly.
- Design your system using a Bottom-Up strategy, after all the system should be in the form of modules which interact with each other, using these modules in a hierarchical way to build your processor
- Use test benches to test your design and to avoid forcing signals every time by hand
- Test vectors will be provided on the discussion day.
- For the assembly to machine code translator, you can use any available software, and all the teams can share same software as well. You will not be graded for coding the translator.
- Any missing information will be assumed by you and hence you need to justify why you took these assumptions.

Memory Details

The memory is word-addressable, where the word is 16 bits. The memory address bus is 16-bits wide. Thus the PC register need not be more than 16 bits. All memory addresses used will be relative addresses that start from ZERO.

For those who will use Von Neumann architecture, the memory size for 1st level (cache) is 16 word, while for the 2nd level (Main memory) it will be 2 K word locations.

For Harvard Architecture the total size of D and I (Data and Instruction) is 16 word for the cache and 2K word for the main memory. You have choice of how to split them; it is not required to have D and I of equal sizes.

The processor control unit will deal with the 1st level and it doesn't have access to the 2nd level, in case memory address isn't in the 1st level, a miss is introduced and the controller is responsible to retrieve the required addresses from the 2nd level and here comes your own design for the cache controller to decrease the miss rate and hence obtain faster code.

In our processor -after all- both 1st level and 2nd level are just a ram so to simulate that access for 1st level is faster than 2nd level, WMFC is introduced for access of 2nd level address (assume it is fixed delay with additional one clock cycle) while for 1st level there is no WMFC.

You don't need to use the external buses the same way they are used in PDP-11, you are free to design how the processor interact with the memory (but make sure that the cache is transparent to the processor).

Note that when a cache miss happens, there is one clock cycle spent to check the cache and deduce that the needed memory location is not stored in the cache and that a miss happened.

Stack

You need to assume the presence of a static sized stack of 128 words that is available at the end of the memory map. You are free to decide whether the stack is filled in upwards or downwards direction.

Registers

In the testing program that will be used to test your designs, only 4 general purpose 32-bits registers will be used (R0, R1, R2 and R3). Moreover you must implement special purpose registers such as PC (program counter), SP (stack pointer), and you are free to add any number of internal registers that are hidden from the programmer's side. You are supposed to design the suitable internal bus architecture according to your needs.

Addressing Modes

You are required to support three addressing modes:

- Register Direct: Operands are found directly in registers
- Absolute Address: The effective address for an absolute instruction address is the address parameter itself with no modifications. For example [3FH] to access the contents in address 3FH.
- Register Indirect: The effective address for a Register indirect instruction is the address in the specified register. For example, [R1] to access the content of address in register R1.

More details about the addressing modes could be understood from the ISA table.

Instruction Set

Below is a table that describes all the instructions that must be supported by your processor. The default type of operand is double word (32-bits) unless otherwise indicated. For each instruction, the supported addressing modes are presented. For a given instruction, if one of the addressing modes is not identified, then it is NOT required.

Mnemonic	Description	Addressing Modes			Operation
		Register Direct	Register Indirect*	Absolute**	
Add	Adds two 32-bit operands	add r1, r2	add r1, r2	add r1, [ADD0]	R1 = R1 + R2
Adc	Adds two 32-bit operands + Carry	adc r1, r2	adc r1, [r2]	adc r1, [ADD0]	R1 = R1 + R2 +Carry
Sub	Subtracts two 32-bit operands	sub r1, r2	sub r1, [r2]	sub r1, [ADD0]	R1 = R1 - R2
Inc	Increments operand by 1	inc r1			R1 = R1 + 1
Dec	Decrements operand by 1	dec r1			R1 = R1 – 1
Mulb	Multiplies two least 8-bits of operands and puts result in the least 16-bits of first operand	mulb r1, r2			R1[15..0] = R1[7..0] * R2[7..0]
Mulw	Multiplies two least 16-bits of operands and puts result in the 32-bits of first operand	mulw r1, r2			R1[31..0] = R1[15..0] * R2[15..0]
Muld	Multiplies two 32-bits operands and puts result two operands where the High Siginificant 32-bits are in the first operand, and the least are in the second operand	muld r1, r2			R1[31..0].R2[31..0] = R1[31..0] * R2[31..0]
div	Divides two 32-bits operands and put result in first operand	div r1, r2			R1 = R1 / R2 R2 = R1%R2
And	Logical AND for 32-bits operands	and r1, r2	and r1, [r2]		R1 = R1 and R2
Xor	Logical XOR for 32-bits operands	xor r1, r2	xor r1, [r2]		R1 = R1 xor R2
Or	Logical OR for 32-bits operands	or r1, r2	or r1, [r2]		R1 = R1 | R2
Not	Inverts 32-bit operand	not r1			R1 = not R1
cmp	Compares two 32-bits operands	cmp r1, r2			Flags are modified
swp	Swaps operands	swp r1, r2			R1 swapped with R2
clr	Clears operand	clr r1			R1 = ZERO

mov	Copies second operand into first operand	mov r1, r2	mov [r1], r2	mov r1, [ADD0]	R1 = R2
shl	Logical Shift Left	shl r1			R1[31..1] = R1[30..0], R1[0] = 0
shr	Logical Shift Right	shr r1			R1[30..0] = R1[31..1], R1[31]=0
ror	Logical Rotate Right	ror r1			R1[30..0] = R1[31..1], R1[31]=R1[0]
rorc	Logical Rotate Right with Carry	rorc r1			ROR R1, Carry = R1[0]
rol	Logical Rotate Left	rol r1			R1[31..1] = R1[30..0], R1[0] = R1[31]
rolc	Logical Rotate Left with Carry	rolc r1			ROL R1, Carry = R1[31]
jmp	Jump to operands address		jmp [r1] ++	jmp [074F] ++	PC = 074F
jz	Jump if ZERO		jz [r1] ++	jz [074F] ++	If Z=1, PC = 074F
jnz	Jump if NOT ZERO		jnz [r1] ++	jnz [074F] ++	If Z=0, PC = 074F
push***	Push operand on top of stack	push r1			SP = SP -2, [SP] = R1
pop***	Pops top of stack into operand	pop r1			R1 = [SP], SP = SP + 2
clc	Clear Carry flag				Carry = 0
stc	Sets Carry flag				Carry = 1
clz	Clear ZERO flag				ZERO = 0
hlt	Halt Processor. PC does not increment anymore				Stop Execution
reset	Resets processor. PC is initialized to the beginning of the program				PC = Initial value
nop	No Operation				PC = PC + 2

** Note that in the table above, for all Register Indirect operations, the indirect addressing could be applied to first operand, or second operand, or both. For example, the add instruction could be in one of these forms: add r1, [r2] or add [r1], r2 or add [r1], [r2]. Hence the MOV instruction is used load and store data in memory by calling MOV R1, [R2] and MOV [R1], R2, respectively. Moreover, the MOV instruction could be used to copy memory locations by calling MOV [R1], [R2].*
*** The same rule for Register Indirect applies to the Absolute instructions.*
**** The operation of these instructions are for stacks that are filled in upwards direction.*
++ For jmp, jz, jnz: The absolute mode is like when we are using a label for a loop, in this case the address in in the IR and no memory accesses needed. For the indirect mode, the address to jump to is stored in memory.

- For instructions related to bonus sections (such as floating point instructions), each group is free to design his own set of instructions and prepare several programs to demonstrate it.

Evaluation Criteria

Here are some guidelines to how the evaluation process will be going to give you some insight about what you need to care about when thinking about your design.

- Number of supported Instructions according to the ISA given above (you have to implement all of them of course).
- Well documented reports.
- Average CPI.
- Cache protocol used by your cache controller (hit count and miss count as well will be taken into account so you need to show them as output signals).
- Teamwork (organization, efficient workload distribution).
- Code comments.
- Following a naming convention in your code.
- **(Bonus)** Fast implementation for Multiply and Divide operations without usage of embedded functions.

How will the TA test my processor?

You will be given different memory initialization file that contains different test programs. You are required to load it in the RAM, reset your processor to start and execute from memory location 0000. Each program would test some instructions (you should notify the TA if you haven't implemented or have logical errors concerning some of the instruction set), each file will include a small program will contain some instructions and some variables.

You MUST prepare a wave form with the main signals that show that the processor is working correctly (R0, R1, R2, and R3), (reset, clk, hit count, miss count), (PC, IR, MAR, MDR, UAR …).

The test programs' code will have the following properties:

- 20% of the instructions will be jump instructions (conditional or unconditional).
- 5% of the instructions will be Mul or Div instructions.
- The mean number of instructions in a loop body (without the loop header) is 3 instructions, with a standard deviation ±1 instruction.

Deliverables

Phase #1

1. Design Document with the following elements
    - Internal Bus architecture and accordingly the data path.
    - Control Unit (whether hardwired or microprogrammed control) including every detail (e.g. optimization done, bit Oring and ROM size in case of microprogammed control).
    - Cache policy and replacement algorithm used.
    - **(Bonus)** Implementation design for both Mul and Div, and an estimate of HW and time needed for each of them.

Phase #2

1. Project Document, describing each module in your system and a small description on how It behaves for different inputs in different modes (I/P O/P relationship or in other words each module as a black box)
2. The code for each group and the testbench for the whole processor (and any extra files like RAM and ROM files) will be put in a folder named with the group name, and delivered on a CD.

Due Date and other issues

- Due date for the phase #1 will be Tuesday 3/12/12 10:00 AM (a hardcopy of your design document in Eng Sherif Shehata's mailbox)
- Due date for the phase #2 will be Sunday 22/12/12 10:00 AM (this due date is for both the CD, in addition to a hardcopy of your project document in Eng Sherif Shehata's mailbox). This doesn't mean that you will start working in phase #2 from 3/12 till 22/12, you are free to start earlier than 3/12 on phase #2.
- Demonstration and oral discussion for it will be announced later
- Meetings are to be scheduled with the TA in advance.
- If you have any questions regarding VHDL, architectural design and project requirements, refer to your Architecture TA.
- Any email should contain subject [CMP301 Computer Architecture project][Group_number] and replace Group_number with your group number in the spreadsheet (e.g. Group 5 would send an email with subject like this [CMP301 Computer Architecture project][G05])