Cairo University Faculty of Engineering CCE Program	CMP N 301 Spring 2020

COMPUTER ARCHITECTURE FINAL ASSESSMENT

Objective

To design and implement a simple 5-stage pipelined processor, **Harvard Architecture**. Two separate memories with their first level cache system (one for data and one for instructions). **The cache system is depending on direct mapping cache system.**

The design should conform to the ISA specification described in the following sections.

Introduction

The processor in this project has a RISC-like instruction set architecture. There are eight 4-byte general purpose registers; R_0, till R7. Another two general purpose registers, one works as program counter (PC). And the other, works as a stack pointer (SP); and; hence, points to the top of the stack. The initial value of SP is (**2^12-1**). **The memory address space is 4 KB of 16-bit width and is word addressable. (N.B. word = 2 bytes). The bus between memory and the processor is (16-bit or 32-bit) widths for instruction memory and 32-bit widths for data memory**

When an interrupt occurs, the processor finishes the currently fetched instructions (instructions that have already entered the pipeline), then the address of the next instruction (in PC) is saved on top of the stack, and PC is loaded from address [2-3] of the memory (the address takes two words). To return from an interrupt, an RTI instruction loads PC from the top of stack, and the flow of the program resumes from the instruction after the interrupted instruction. **Take care of corner cases like Branching,Push,POP.**

ISA Specifications

A) Registers

R[0:7]<31:0> ; Eight 32-bit general purpose registers
PC<31:0> ; 32-bit program counter
SP<31:0>; 32-bit stack pointer
CCR<3:0> ; condition code register
Z<0>:=CCR<0> ; zero flag, change after arithmetic, logical, or shift operations
N<0>:=CCR<1> ; negative flag, change after arithmetic, logical, or shift operations
C<0>:=CCR<2> ; carry flag, change after arithmetic or shift operations.

B) Input-Output

IN.PORT<31:0> ; 32-bit data input port
OUT.PORT<31:0> ; 32-bit data output port
INTR.IN<0> ; a single, non-maskable interrupt
RESET.IN<0> ; reset signal

Rsrc1 ; 1st operand register
Rsrc2 ; 2nd operand register
Rdst : result register
EA ; Effective address (20 bit)
Imm ; Immediate Value (16 bit)

Take Care that Some instructions will Occupy more than one memory location

Mnemonic	Function	Grade
One Operand		
NOP	PC ← PC + 1	**4 Marks**
SETC	C ←1	
CLRC	C ←0	

NOT Rdst	NOT value stored in register Rdst
	R[Rdst] ← 1's Complement(R[Rdst]);
	If (1's Complement(R[Rdst]) = 0): Z ←1; else: Z ←0;
	If (1's Complement(R[Rdst]) < 0): N ←1; else: N ←0	

INC Rdst	Increment value stored in Rdst
	R[Rdst] ←R[Rdst] + 1;
	If ((R[Rdst] + 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rdst] + 1) < 0): N ←1; else: N ←0	

DEC Rdst	Decrement value stored in Rdst
	R[Rdst] ←R[Rdst] – 1;
	If ((R[Rdst] – 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rdst] – 1) < 0): N ←1; else: N ←0	

OUT Rdst	OUT.PORT ← R[Rdst]	
IN Rdst	R[Rdst] ←IN.PORT	
Two Operands		
SWAP Rsrc, Rdst	Store the value of Rsrc 1 in Rdst and the value of Rdst in Rsc1 flag shouldn't change	**4 Marks**
ADD Rsrc1, Rsrc2, Rdst	Add the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0	
IADD Rsrc1,Rdst,Imm	Add the values stored in registers Rsrc1 to Immediate Value and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0	
SUB Rsrc1, Rsrc2, Rdst	Subtract the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0	
AND Rsrc1, Rsrc2, Rdst	AND the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0	
OR Rsrc1, Rsrc2, Rdst	OR the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0	
SHL Rsrc, Imm	Shift left Rsrc by #Imm bits and store result in same register **Don't forget to update carry**	
SHR Rsrc, Imm	Shift right Rsrc by #Imm bits and store result in same register **Don't forget to update carry**	
Memory Operations		
PUSH Rdst	M[SP--] ← R[Rdst];	**4 Marks**
POP Rdst	R[Rdst] ← M[++SP];	
LDM Rdst, Imm	Load immediate value (16 bit) to register Rdst R[Rdst] ← {0,Imm<15:0>}	
LDD Rdst, EA	Load value from memory address EA to register Rdst R[Rdst] ← M[EA];	
STD Rsrc, EA	Store value in register Rsrc to memory location EA	

	M[EA] ←R[Rsrc];	
Branch and Change of Control Operations		
JZ Rdst	Jump if zero If (Z=1): PC ←R[Rdst]; (Z=0)	**3.5 Marks**
JN Rdst	Jump if negative If (N=1): PC ←R[Rdst]; (N=0)	
JC Rdst	Jump if negative If (C=1): PC ←R[Rdst]; (C=0)	
JMP Rdst	Jump PC ←R[Rdst]	
CALL Rdst	(M[SP] ← PC + 1; sp-2; PC ← R[Rdst])	
RET	sp+2, PC ←M[SP]	
RTI	sp+2; PC ← M[SP]; Flags restored	

Input Signals		Grade
Reset	PC ←{M[1], M[0]} //memory location of zero	**0.5 Mark**
Interrupt	M[Sp]←PC; sp-2;PC ← {M[3],M[2]}; Flags preserved	**1 Mark**

Memory cache system

The Main Memory address space is 4 KB of 16-bit width and is word addressable. (N.B. word = 2 bytes). So the address length for data and instruction memory is 11 bits. **The read and write operations in "Main Memory" takes 4 cycles, you must wait 4 clock cycles before you read from or write to main memory (to read from or write to cache in case of HIT, it will take 1 clock cycle but on MISS it will take more than 4 clock cycles (*Miss Penalty*) to bring the block from main memory to cache memory and in case of MISS you must stall the pipe till the read or write operation complete)**.The cache geometry is (512, 16, 1). This means that the total cache capacity is 512 bytes, that each cache block is 16 bytes (implying that the cache has 32 blocks in total), and that the cache uses direct mapping.

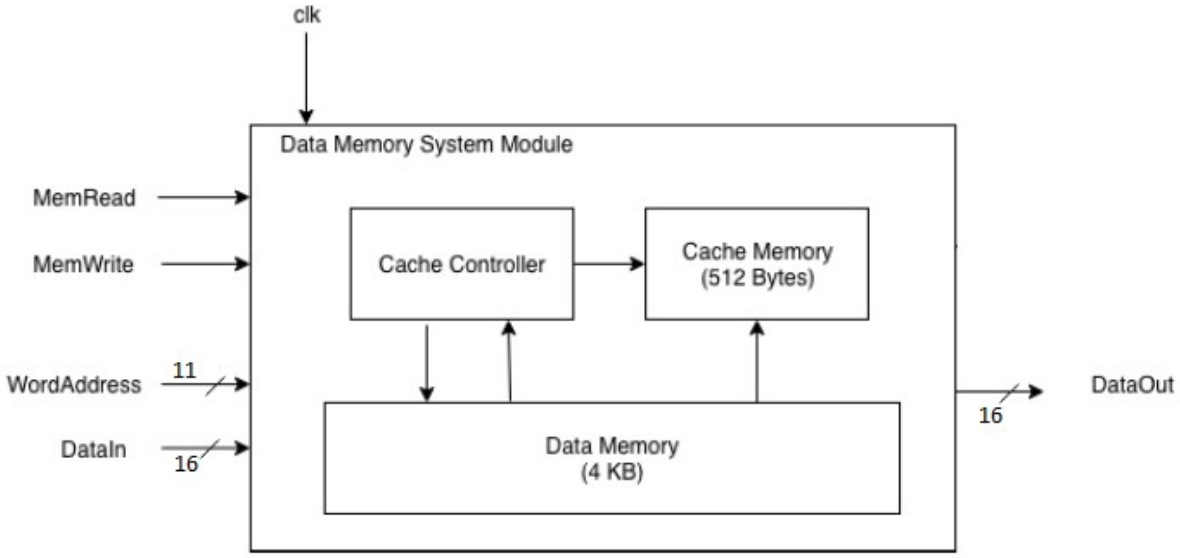
In order to build the caching system, the memory system module is shown in the following Figure. Each block has 16 bytes (8 word) so we need 3 bits for displacement (Word selection within block), and we have 32 blocks so we have 5 bits for index. Therefore, we have 3 bits for tag (11 – 5 – 3). The address is 11 bits is divided as the following:

- Bits from 0 to 2 are displacement
- Bits from 3 to 7 are index
- Bits from 8 to 10 are tag

The cache controller encapsulates the **array of tags, valid bits, and dirty bits** and uses the index and tag parts of the requested memory address to decide whether there is a hit or a miss. It is also responsible for controlling both the cache module and the memory module as explained in the 4 scenarios below:

- The processor requests a read operation (executing a LW instruction) and the cache controller decides that it is a hit. In this case, the data is read from the cache module. (read hit)
- The processor requests a read operation (again executing a LW instruction) and the cache controller decides that it is a miss. In this case, the data is read from the data memory module which provides 1 block (16 bytes or 128 bits) of data to the data cache. When this data is available, the data memory module asserts a ready signal that the cache controller

uses to ask the data cache to fill the corresponding block with the data coming from the memory (read miss)



- **The processor requests a write operation (executing a SW instruction) and the cache controller decides that it is a hit. In this case, the word to be stored has to be written in cache memory but not written in the main memory at the same time but we set the dirty bit of this block in cache to be written in main memory when this block is removed from the cache (when it is replaced by another block this called write-back policy). So the cache controller writing into cache memory the required data and set dirty bit. And when the block is replaced by another block, the cache controller checks the dirty bit to ensure that it need to write this block in main memory or not and if it needs to write it into the main memory then it needs to stall the pipeline till the write operation complete. (write hit)**
- **The processor requests a write operation (again executing a SW instruction) and the cache controller decides that it is a miss. In this case, the cache controller request the memory block from the main memory first and put it in cache then it write the new data in cache memory and set the dirty bit of this block and when this block is replaced by another block the cache controller check the dirty bit and if it is set then the controller will write it back into the main memory (due to the write-allocate policy); however, in this case too, the cache controller asserts the stall signal until the memory finishes the storing. (write miss)**

Note that in order to decide whether the access is a hit or a miss, the cache controller has to be provided with the index and the tag of the address being accessed and it also needs to have access to the array of tags and the array of valid bits corresponding to the cache blocks in the cache module.

Finial deliverables:

- **Due Date : 2nd June 2020**
- Delivery on E-learning,
- A software to compile sample programs and generate the equivalent hex files to be loaded to the memory. (Assembler code that converts assembly program (Text File) into machine code according to your design (Memory File))
- Implement and integrate your architecture
  - **VHDL** Implementation of each component of the processor
  - **VHDL** file that integrates the different components in a single module
- Simulation Test code that reads a program file and executes it on the processor.
  - Setup the simulation wave
  - Load Memory File & Run the test program
- **A Report that contains the following:**
  - **The full design of your processor**
  - **A detailed analysis of the effectiveness of Hazard detection unit, forwarding unit, and implemented branch prediction technique in your processor. You can do this analysis incremental by doing the following steps:**
    i. **For each test case: run your pipelined processor without forward unit, hazard detection and flushing, then report the types of hazards happen in the test case. Write the hazards in the report, then show how you can solve it by stall the pipe using no operation (please support your notes by screenshots from your simulation)**
    ii. **Add the forward unit in your processor and run the test case. Also write the detected hazards and show how you will fix it using no operation (please support your notes by screenshots from your simulation).**
    iii. **Repeat the same procedure on adding the hazard detection with forwarding unit**
    iv. **Repeat the same procedure on adding the flushing with hazard detection and forwarding unit**

Project Testing

- You will be given different test programs. You are required to compile and load it onto the RAM and **reset** your processor to start executing from memory location 0000h. Each program would test some instructions.
- You MUST prepare a waveform using do files with the main signals showing that your processor is working correctly (R0-R7, PC,SP,Flags,CLK,Reset,Interrupt, IN.port,Out.port) and include these forms in your report.

Evaluation Criteria

- Each project will be evaluated according to the number of instructions that are implemented, and Pipelining hazards handled in the design. Table 2 shows the evaluation criteria in detail.
- Delivered report showing the running of your processor on different test cases + the analysis of the effectiveness of each module as explained earlier.
- Failing to implement a working processor will nullify your project grade. No credits will be given to individual modules or a non-working processor.
- Unnecessary latching or very poor understanding of underlying hardware will be penalized.
- **Cheating == Zero**

Table 2: Evaluation Criteria

Marks Distribution	Report	**10 Points**
	Memory cache	**5 Points**
	Instructions	**17 points**
	Handling Hazard	**5 Points**
	Static branch prediction	**3 Points**

Team Members

- Each team shall consist of a maximum of four members .

General Advice

1. Compile your design on regular bases (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
2. Use the engineering sense to back trace the error source.
3. As much as you can, don't ignore warnings.
4. Read the transcript window messages in Modelsim carefully.
5. After each major step, and if you have a working processor, save the design before you modify it (use a versioning tool if you can as git & svn).
6. Always save the ram files to easily export and import them.
7. Start early and give yourself enough time for testing.
8. Integrate your components incrementally (i.e: Integrate the RAM with the Registers, then integrate with them the ALU …).
9. Use coding convention to know each signal functionality easily.
10. Try to simulate your control signals sequence for an instruction (i.e: Add) to know if your timing design is correct.
11. There is no problem in changing the design after phase1, but justify your changes.
12. Always reset all components at the start of the simulation.
13. Don't leave any input signal float "U", set it with 0 or 1.
14. Remember that your VHDL code is a HW system (logic gates, Flipflops and wires).
15. Use Do files instead of re-forcing all inputs each time.