Cairo University	CMP 301B	
Faculty of Engineering	Spring 2020	
Computer Engineering Department		

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 ($2^{11}-1$). 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 widths for instruction cache and 32-bit widths for data cache.

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,Calling.**

ISA Specifications

1. **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.

2. **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

1

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	

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
		6 Marks

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	
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	**8 Marks**
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];	
POP Rdst	R[Rdst] ← M[++SP];	**5 Marks**
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)	
JMP Rdst	Jump PC ←R[Rdst]	**5 Marks**
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 and all registers get reseted	**2 Mark**
Interrupt	M[Sp]←PC; sp-2;PC ← {M[3],M[2]}; Flags preserved	**4 Mark**

Memory cache system(Direct mapping cache system)

The 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**
- In case of cache HIT, it will take **1 clock cycle**
- In case of cache MISS:
  - It will take **1 cycle** to detect that it is miss (normal stage)
  - In case of write back a dirty block it will take **4 cycles** to write it to the cache (Stall processor)
  - It will take **4 cycles** to read from Main memory to Cache (Stall the processor)
- Emulate memory delay using a counter that counts the number of cycles.

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.

The bus between the cache and main memory is 128 bit.

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 offset (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 offset
- 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 for each cache separately (what is the length of each array?)** and uses the index and tag parts of the requested memory address (but not offset why?) to decide whether there is a hit or a miss. It is also responsible for controlling the instruction cache, data cache and the main memory module as explained in the scenarios below:
1. **Read Hit:** The processor requests a read operation (reading instruction or executing a LDD/POP instruction) and the cache controller decides that it is a hit. In this case, the data is read from the cache module.(**1 cycle**, **no processor stalling**, how did it decide that it is a hit?)
2. **Read Miss:** The processor requests a read operation and the cache controller decides that it is a miss. (**1 cycle** to detect miss, how did it decide that it is a miss?)
   a. **Stall** the processor
   b. Check whether the block to be replaced in the cache is valid and dirty (has modified data) If valid and dirty, do the following:
      i. **Write Back:** write the **whole block** back to the main memory (**4 cycles** to save data to main memory)
      ii. when the memory finishes writing the data, it sends a **ready signal** to the controller.
   c. The data/instruction is read from the main memory module which provides **1 block (16 bytes or 128 bits)** of data to the cache. (**4 cycles** to get data from main memory)

4

        i. When this data is available, the main memory module asserts **a ready signal** to the cache controller

       ii. The cache controller asks the cache to fill the corresponding block with the data coming from the memory and updates the corresponding tag, valid and dirty bits
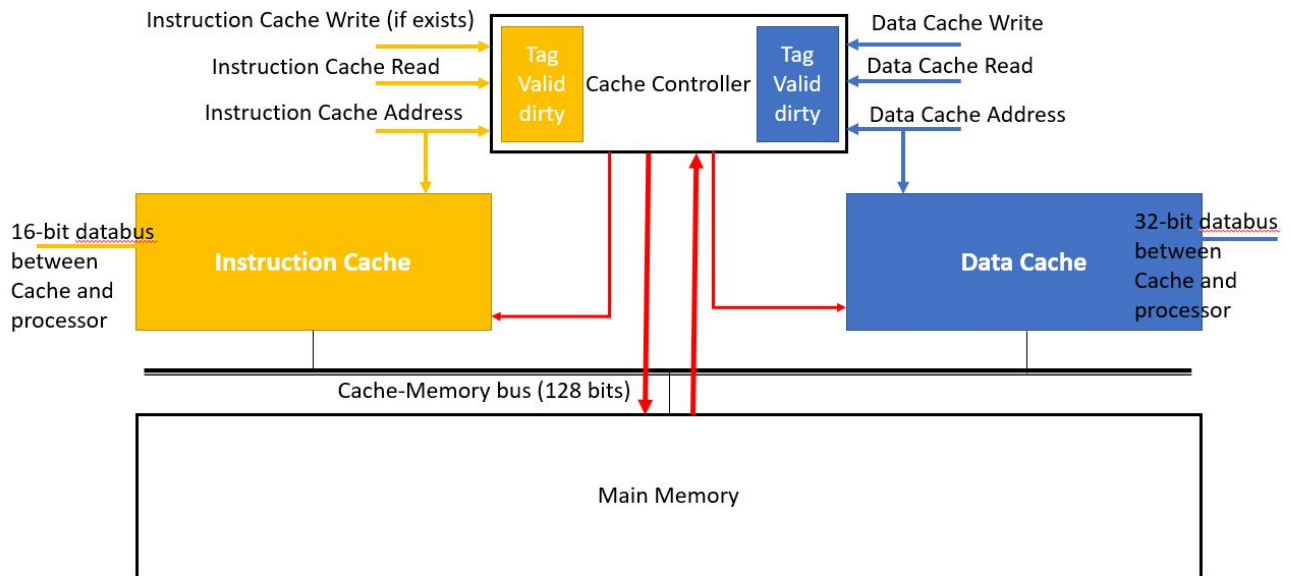
   d. Go to no. 1 (Read Hit)

Figure:1- Memory and Cache, Red-lines are control lines that should be designed by you, some signals like clk are omitted from drawing

3. **<u>Write Hit:</u>** The processor requests a write operation (executing a STD/PUSH instruction) and the cache controller decides that it is a hit. (**1 cycle, no stalling**, how did it decide that it is a hit?)

    o Write the word in the cache <u>only</u>

    o set the dirty bit of this block in cache to dirty (to be written in main memory when this block is <u>replaced by another block this is called write-back policy</u>)

4. **<u>Write Miss:</u>** The processor requests a write operation and the cache controller decides that it is a miss. (**1 cycle**, how did it decide that it is a miss?)

    o Do same steps as Read Miss: 2.a, 2b, 2c

    o Go to no. 3 (Write Hit)

5. **<u>Structural Hazard on miss:</u>** In case of cache miss from both Instruction cache and Data cache at the same time, Data cache should have a priority and served first.

6. you can assume that data (not instruction) always start on an even address (a.k.a no split on several blocks)

5

Final deliverables:
- **Due Date : 2ⁿᵈ June 2020**
- Delivery on E-learning,
- **A zipped folder with named with your team number: i.e. team12.zip containing the following**
    - 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))
    - All HexFiles (memory files) from testcases. **.**
    - Implement and integrate your architecture
        - i. **VHDL** Implementation of each component of the processor
        - ii. **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.
        - i. Setup the simulation wave
        - ii. Load Memory File & Run the test program
- **A Report in PDF format named with your team number that contains the following (IS A MUST):**
    - **Your Team No**
    - **Your Names and Bench No.**
    - **The full design of your processor**
    - **What is not working/not implemented in 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 the right memory location. 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).
- **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

1. 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.
2. <mark>Delivered report showing the running of your processor on different test cases + the analysis of the effectiveness of each module as explained earlier.</mark>
3. Failing to implement a working processor will nullify your project grade. No credits will be given to individual modules or a non-working processor.
4. Unnecessary latching or very poor understanding of underlying hardware will be penalized.
5. **Cheating == Zero**

Table 2: Evaluation Criteria

Marks Distribution	Report (**IS A MUST**)	**20 Points**
	Memory cache	**10 Points**
	Instructions	**30 Points**
	Handling Hazard	**10 Points**
	2-bit dynamic branch prediction with address calculation in fetch stage (take care of hazards and forwarding)	**5 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 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.