EE183 LAB 3

12-bit RISC Microprocessor

Purpose
The third project consists of implementing a pipelined processor. This project illustrates many of the datapath and control design issues you have faced in the previous projects. It is complex enough to be interesting, but simple enough to be done in two weeks.

Assignment

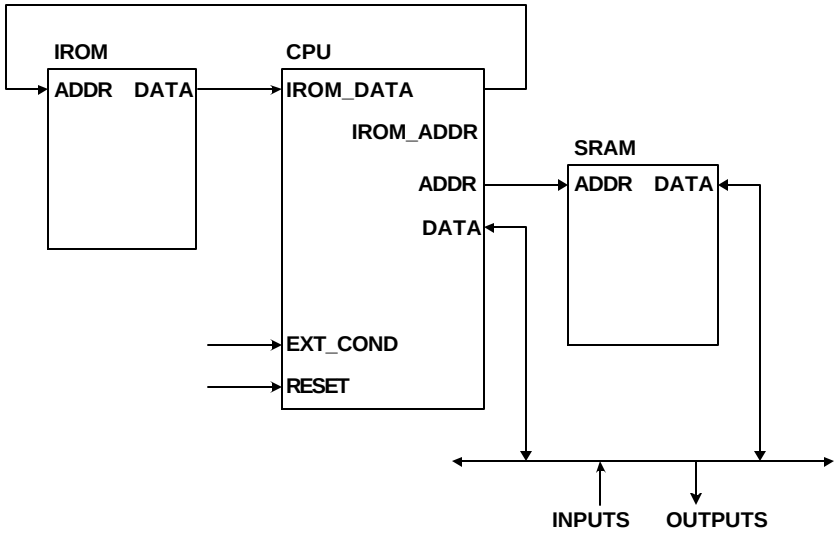
Here are the requirements:

1. Design and implement your processor on one Xilinx FPGA.
2. Demonstrate your working processor by running our test programs.
3. Write several test programs.
4. Demonstrate your working processor by running your test programs.
5. Turn in a write-up.

System Description

There are three main parts to the design: 1) the processor, 2) the instruction ROM, 3) the data RAM. The processor is the only part that you will have to design; the other two parts are provided. Details about each part will follow. Below is a description of the primary inputs and outputs of the system. Figure 6 shows the block diagram of this system.

Figure 1: System Block Diagram



Primary Inputs

CLK	System clock
EXT_COND	Pushbutton which specifies an external condition for conditional branches.
SW_DATA[11:0]	Set of external switches for data input to the processor.

The Processor
The processor is to be designed by you. You must follow the specifications exactly so that programs written for this instruction set will run on your completed hardware. In other words, you cannot modify the instruction set architecture.

The entire processor must fit on a single FPGA. There is plenty of room to fit everything if you are efficient, but a careless design will overflow the part. Parts of the processor design will be given to you. Be sure to look at all the verilog to see what is provided. You must fill in the rest.

The Instruction ROM

The instruction ROM is emulated in the FPGA. We will be using the Xilinx CORE generator to create this element. A sample IROM is included in the starter files, but you will have to generate a separate IROM for each program you wish to run. To do this first open up CORE Generator in Foundation (tools->design entry->CORE Generator). A list of current modules should be displayed including the data ram and irom. Right click on the IROM and select recustomize. This will bring up a window allowing you to redesign the IROM block. If you wish you can change the depth (number of instructions) of the ROM. To load a program into the ROM simply select "Load init file" and load in a .coe file generated by asm183.pl. Note that the ROM generated by the CORE Generator is pipelined and provides the data on the following clock edge. This means that the address provided in the I stage will yield corresponding data at the beginning of the R stage.

The Data RAM

The data RAM is a 12-bit wide memory that you can implement using a CORE Generator module. This module has been provided for you in the starter files. It is worth noting that the output data for a read operation is available on the following clock edge. This means that if you provide the address in the E stage the data is not ready until the W stage.

Important notes

- Make sure that you switch out the ROMs when you want to run a different program. This means you will have to generate a separate bit file for each program. When you are ready to demo it would be nice to have all of these generated and ready to display at once.
- You will need to devise a method for displaying your results to the outside world—but we'll have some suggestions.

Processor Description

Figure 2 shows a block diagram of the processor you are to design. The top level schematic of the processor is very similar to the block diagram.

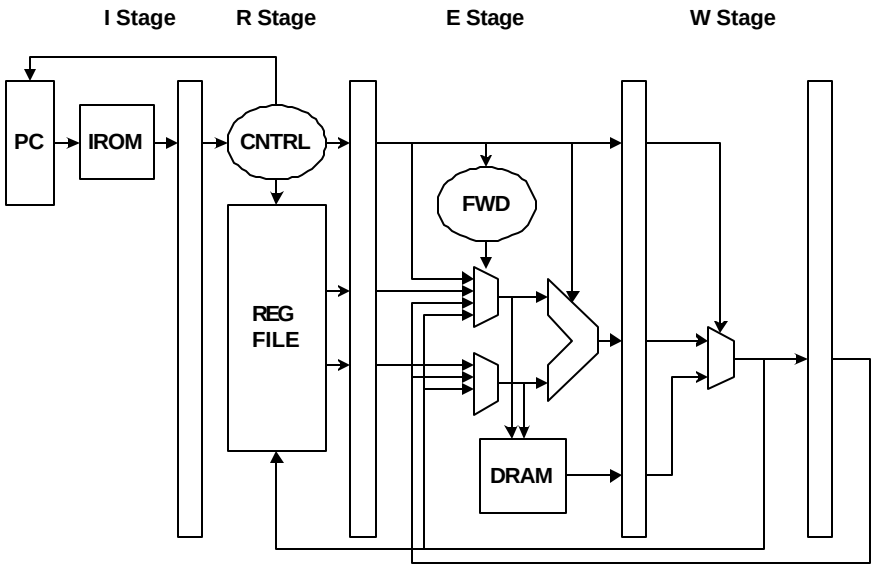


Figure 2: Processor Block Diagram

The processor you are to design is a 12-bit RISC microprocessor. It features 8 general purpose registers, 43 instructions, and a 4 stage pipeline. The external interface signals are listed below. You will be provided with a "template" schematic for the processor design.

CPU_IROM_ADDR_I[11:0]	This output is the address of the instruction to be fetched.
IROM_CPU_DATA_D[15:0]	This input is the instruction word at the address specified by IROM_ADDR[11:0].
CPU_DRAM_ADDR_E[11:0]	This output is the address of the data RAM location being accessed.
CPU_DRAM_DATA_E[11:0]	This is the data to be written on a STORE
DRAM_CPU_DATA_M[11:0]	This is the data that has been read on a LOAD
DRAM_WE	This output is the data RAM write enable.
CLK	This input is the system clock
EXT_COND	This input specifies an external condition for conditional branches.
RESET	This input sets the PC to zero and clears out all the pipeline flops.

Instruction Set

Several fields occur frequently in the instruction set encodings. These fields are the destination register (WC), the A source register (RA), and the B source register (RB). The interpretation of other fields depends on the instruction class.

ALU Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1														
0	1	WC			OP					RA			RB		

ALU Instructions

OP Field	Operation	Mnemonic	
00	C = A + B	ADD	C, A, B
01	C = A + B + 1	ADDINC	C, A, B
02	C = A	PASSA	C, A
03	C = A + 1	INCA	C, A
04	C = A - B - 1	SUBDEC	C, A, B
05	C = A - B	SUB	C, A, B
06	C = A - 1	DECA	C, A
07	C = A	PASSA	C, A
08	C = Logical Shift Left (A)	LSL	C, A
09	C = Arithmetic Shift Right (A)	ASR	C, A
10	C = 0	ZEROES	C
11	C = A · B	AND	C, A, B
12	C = A' · B	ANDNOTA	C, A, B
13	C = B	PASSB	C, B
14	C = A · B'	ANDNOTB	C, A, B
15	C = A	PASSA	C, A
16	C = A xor B	XOR	C, A, B
17	C = A + B	OR	C, A, B
18	C = A' · B'	NOR	C, A, B
19	C = A xor B'	XNOR	C, A, B
1A	C = A'	PASSNOTA	C, A
1B	C = A' + B	ORNOTA	C, A, B
1C	C = B'	PASSNOTB	C, B
1D	C = A + B'	ORNOTB	C, A, B
1E	C = A' + B'	NAND	C, A, B
1F	C = 1	ONES	C

Literal Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1															
1	L	WC			LITERAL										

Literal Instruction

OP Field	Operation	Mnemonic	
1	C = literal	LOADLIT	C, literal

In the literal instruction, the literal data is not in a contiguous bit field. The most significant bit of the literal is in the L field. Originally this instruction only supported 8-bit literals, but it was extended to 12 bits to make it more useful.

Memory Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1														
0	1	WC			OP					RA			RB		

Memory Instructions

OP Field	Operation	Mnemonic	
0A	C = Mem[A]	LOAD	C, A
0B	Mem[A] = B	STORE	A,B

Control Transfer Instruction Formats

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

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

Control Transfer Instructions

OP Field	Operation	Mnemonic	
0	Jump false	JF.cond	JPC
1	Jump true	JT.cond	JPC
2	Jump	J	JPC

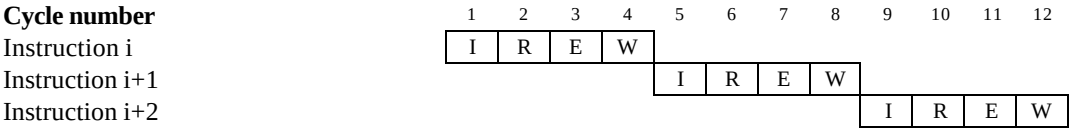
COND Field	Operation	Mnemonic
0	Always true	.TRUE
4	Result < 0	.NEG
5	Result = 0	.ZERO
6	Carry = 1	.CARRY
7	Result <= 0	.NEGZERO
8	Ext Cond = 1	.EXT

The NOP instruction is implemented by using the jump false on condition true instruction (JF.TRUE). The encoding of this instruction results in an instruction word of all zeros.

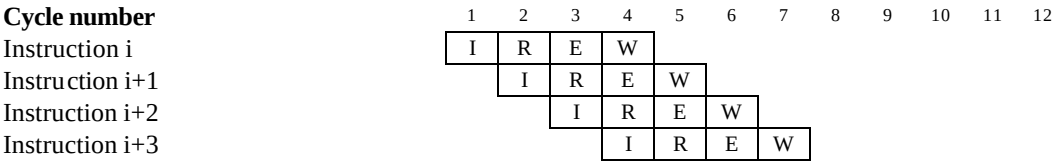
Except for the external and always true conditions, the other conditions are set as a result of ALU operations. In your design, assume that conditional jumps always follow an ALU instruction. In other words, you do not need to save the condition codes from the ALU. Placing a conditional jump after a non-ALU instruction will result in unspecified behavior. Avoid this while writing programs.

Hazards Created By Pipelining

In a non-pipelined processor, the execution of each instruction is completed before the next instruction is begun. Consider the cycle diagram for a non-pipelined, multi-cycle version of the processor:



Each instruction takes four cycles to complete, and the machine throughput is 0.25 instructions per cycle. Now consider the cycle diagram for a four-stage pipelined version of the processor (which you will build):



Each instruction still takes four cycles to complete, but the machine throughput is now (ideally) 1.0 instructions per cycle during sustained execution (see cycles 4 and 5). Pipelining improves performance by increasing instruction throughput. This is important because real programs do not consist of three or four instructions, but billions of instructions.

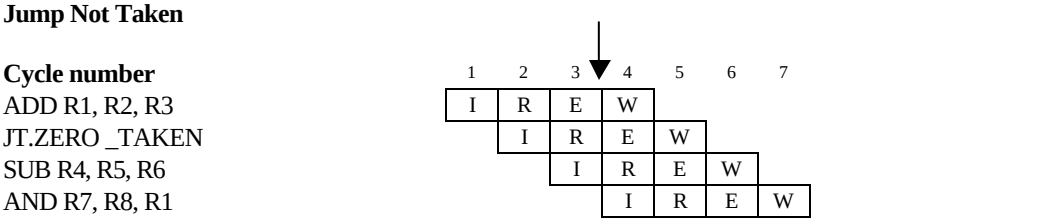
Unfortunately, nothing in this world is free (incidentally, though, pipelining is the closest thing to a free lunch in digital design). By overlapping instruction execution in a pipeline, *hazards* are created. There are two significant types, control hazards and data hazards.

Control Hazards—Consider the following code fragment:

```
                ADD         R1, R2, R3
                JT.ZERO     _TAKEN
                SUB         R4, R5, R6
                AND         R7, R8, R1
                …
                …
_TAKEN          NOR         R7, R8, R1
```

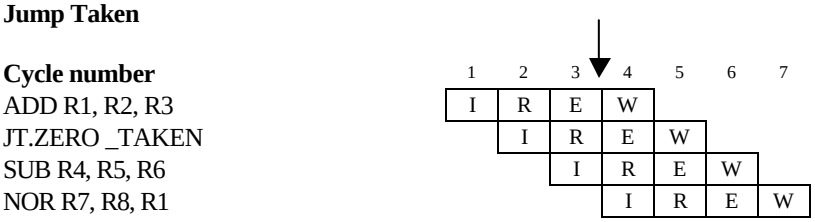
Due to the fact that the condition code (CC) for the conditional jump is not known until the ADD completes the execution stage, the instruction following the conditional jump will enter the pipeline. (Arrow signifies condition code being set)

Jump Not Taken



Since the jump was not taken, the instruction following the conditional jump is meant to be executed and is already in the pipeline. This is not a problem.

Jump Taken



In this case, however, the instruction following the conditional jump is not meant to be executed. There are several ways to deal with this, including flushing the pipeline or inserting bubbles. However, this requires additional control logic. For the purposes of this lab, simply follow all jumps in your program with a NOP instruction. This way, the unwanted execution is harmless.

Data Hazards

Consider the following code fragment:

```
_START    ADD     R1, R2, R3
          SUB     R4, R1, R5
          NOR     R6, R1, R7
```

This code fragment results in the cycle diagram shown below. Note that the source operand of the SUB and NOR instructions (R1) depends on the result of the ADD instruction.

From the above diagram, you can see that the SUB and NOR instructions will attempt to read their operands from the register file in cycles 3 and 4, before the ADD writes back the correct data value into the register file. The data written back by the ADD instruction is not available until the beginning of cycle 5. This will result in incorrect execution of the program.

Conceptually, the write data from the AND must be forwarded to the SUB and NOR instructions to ensure correct execution. Depending on the implementation of the processor, there may be more than one way to achieve this effect. In this implementation of the processor, all data is forwarded to the E stage, as shown in the block diagram of the processor. This requires a copy of the destination register address and data to be saved after the W stage for possible use in the E stage (note the "extra" pipe segment after the W stage in the block diagram).

It is entirely possible to forward from the W stage to the R stage instead of the above arrangement. However, the schematic template you will be given is not arranged this way, so if you implement this method, you will need to alter the template.

Do not forget that the memory instructions also transfer data to and from the register file (in fact, the LOAD and STORE instructions are encoded as ALU operations). You are responsible for implementing complete data forwarding logic so that inserting NOP instructions is not necessary for correct program execution.

Asm183

You will be provided with a perl script called asm183 which will help you with the task of assembling the programs you write. You could do this by hand using the information you have on the instruction set, but this course of action has been reported to be excessively painful. Since asm183 is written in perl, you can run it anywhere you have access to perl. Asm183 takes your program, my_prog.asm, and creates a Simulator command file, my_prog.cmd, so that you may simulate its execution. Invoke asm183 by typing “perl asm183.pl my_prog.asm”. ASM183 provides several output formats. The two that you will find useful

are .coe and .lis. The .coe file can be used by the CORE Generator program to preload your IROM with a program. The .lis file contains the address, machine code, and assembly for each line to make debugging easier to follow.