CSCI3420 Computer Systems Architecture
Project Phase 3 Specification
Deadline: 23:59 26th April, 2013

1 Introduction

This course project is designed to let you gain an in-depth understanding of CPU architecture by implementing a single-cycle simulator and a multi-cycle pipelined simulator of a simplified CPU. In this Phase, you have to write a **multi-cycle pipelined simulator**.

2 Architecture

The (instruction set) architecture you are simulating is almost the same as those in Phase 2. To simplify the design, you are NOT required to implement the **"jal"** and **"jr"** instructions in this Phase. Also, only "$lo" is needed to be handled for "mult" and "div" instruction. Please refer to Appendix for details.

Figure 1 shows the datapath you need to simulate. It should be noted that the datapath in Figure 1 is a conceptual block diagram, not a complete system diagram. You should try to understand the relevant relationships between different blocks before working on the simulator.

3 Simulator

You are required to write a pipelined simulator in standard C (and CANNOT use extra libraries) to simulate the MIPS CPU **cycle-by-cycle**. You are **required** to handle all those instructions defined in Appendix 1. You only need to handle "$lo" for "mult" and "div" instructions. The simulator terminates upon reading 0x00000000. It should be noted that your simulator will be compiled using the following command in a *Linux* environment.

```
gcc –o <simulator filename> <simulator source code> -Wall
```

4 Execution

Your simulator should be able to accept one argument which is the filename of your binary assembly program. The main function is provided for you in the provided resources, and you are prohibited from editing any provided code segments in the skeleton program. **Marks will be deducted if you do so**. It should be noted that your simulator will be executed using the following command in a *Linux* environment.

```
./<simulator filename> <binary assembly program filename>
```

To generate a piece of validate machine code, you are suggested to use the provided program: asm_linux.You can use the program called asm_linux in Linux as follows:

```
./asm_linux assembly_code.asm machine_code.asc
```

It should be noted that assembly_code.asm is the MIPS source file and machine_code.asc is the output file of your MIPS code in binary.

5 Tasks

You are required to complete the methods in the provided Phase3_linux.c (Task I to Task IV). **It is forbidden to declare ANY extra global variables in any file, and it is NOT allowed to modify any method except the following:**

Task 1: Connect Component

In `connect_components` method, you have to connect the components according to the datapath properly by constructing linkage from every input to its corresponding value.

Task 2: Initialization

In `initialize_register_states` method, you have to initialize the output states of some components to 0.

Task 3: Simulation Sequence

In `simulate_cycle` method, you have to put down the correct working sequence of the components in one single cycle by calling the component methods sequentially. It should be noted that the simulation sequence is highly critical and sensitive in this Phase. **Please choose the sequence wisely**.

Task 4: Component Methods

In this task, you have to implement all the `component methods`. For each component method, you have to simulate what it does according to the control signals.

You have to implement the component methods. Please note that usually **EACH** component should have **ONE** component method. However, some units can have more than one method such as the Registers Unit, which has different working tasks (reading, writing… etc.) in one single cycle. Therefore, it can have more than one component method responding for these tasks.

6 Marking Scheme

The distribution of marks of Phase 3 is as follows:

Test cases	Marks
forwarding	20%
hazard	20%
branch	20%
Miscellaneous computations	40%

Similar to Phase 2, the simulation environment of Phase 3 is *Linux*. Please make sure that your submitted program can be compiled on the server *linux1*, using gcc. Make sure your simulator can be compiled and run correctly under this setting. A model simulator has been provided to test the behaviors of your simulator. **Your simulator is deemed as correct, as long as it produces the same output as the model simulator.**

7 Submission Guideline

You should zip the following files as a **SINGLE** zip file and upload it to our submission system on the course webpage. The zip file should be named as your **student ID**:

- For the simulator
  - All `.c` and `.h` files of your simulator
  - A makefile, which when make is executed, should compile your simulator properly
- The submission time will be determined according to our submission system. Late submission will NOT be accepted.
- **To AVOID plagiarism, code similarity will be tested. Manual checking will be performed. Students will be selected to be interviewed. ZERO tolerance for plagiarism is strictly imposed. Suspected cases will be forwarded to CSE Department, or even Engineering Faculty for serious consideration. Course failure is possible.**

8 Remarks

- Signed comparisons and arithmetic should be implemented in the ALU.
- Limited by the 16-bit immediate field, all immediate should be between -2^{15} and $2^{15}-1$.
- The behavior of the sample simulator can be tabulated as follows:

Condition	Number of additional cycles
Jump	1
Branch (if taken)	2
Forwarding	0
Hazard	1
Normal	0

9 Appendix 1:

Instruction	Example	Meaning	Remarks
Add	add $s1,$s2,$s3	$s1 = $s2 + $s3	3 reg
Add Immediate	addi $s1,$s2,100	$s1 = $s2 + 100	2 reg, 1 constant. imm will be sign-extended.
Subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	3 reg
Multiplication	mult $s1,$s2	$s1 * $s2	Signed multiplication, $hi stores the high word, $lo stores the low word of the product.
Division	div $s1,$s2	$s1 / $s2	Signed division, quotient in $lo, remainder in $hi.
And	and $s1,$s2,$s3	$s1 = $s2 & $s3	3 reg
Or	or $s1,$s2,$s3	$s1 = $s2 | $s3	3 reg
Shift Left Logical Variable	sllv $s1,$s2,$s3	$s1 = $s2 << $s3	3 reg
Shift Right Logical Variable	srlv $s1,$s2,$s3	$s1 = $s2 >> $s3	3 reg
Load word	lw $s1,100($s2)	$s1 = mem[$s2 + 100]	2 reg, 1 constant
Store word	sw $s1,100($s2)	mem[$s2 + 100] = $s1	2 reg, 1 constant
Set Less Than	slt $s1,$s2,$s3	$s1 = ($s2 < $s3) ? 1 : 0	3 reg. Signed comparison
Branch on Not Equal	bne $s1,$s2,25	If($s1 != $s2) goto PC + 4 + 100	2 reg, 1 offset counted in words. Offset can be positive or negative
Jump	j label	goto label	Can jump forward and backward

Table 1. Instructions to be implemented

Instruction	Usage	Binary Representation					
		6	5	5	5	5	6
Add	add *rd,rs,rt*	0x0f	*rs*	*rt*	*rd*	0	0x11
Add Immediate	addi *rt,rs,imm*	1	*rs*	*rt*	*imm*		
Subtract	sub *rd,rs,rt*	0x0f	*rs*	*rt*	*rd*	0	0x12
Multiplication	mult *rs,rt*	0x0f	*rs*	*rt*	0		0x18
Division	div *rs,rt*	0x0f	*rs*	*rt*	0		0x1a
And	and *rd,rs,rt*	0x0f	*rs*	*rt*	*rd*	0	0x2a
Or	or *rd,rs,rt*	0x0f	*rs*	*rt*	*rd*	0	0x2b
Shift Left Logical Variable	sllv *rd,rt,rs*	0x0f	*rs*	*rt*	*rd*	0	1
Shift Right Logical Variable	srlv *rd,rt,rs*	0x0f	*rs*	*rt*	*rd*	0	2
Load word	lw *rt,offset* (*rs*)	0x1a	*rs*	*rt*	*offset*		
Store word	sw *rt,offset* (*rs*)	0x1b	*rs*	*rt*	*offset*		
Set Less Than	slt *rd,rs,rt*	0x0f	*rs*	*rt*	*rd*	0	0x25
Branch on Not Equal	bne *rs,rt,label*	2	*rs*	*rt*	*offset*		
Jump	j *target*	3	*target*				

Table 2. Instruction Formats

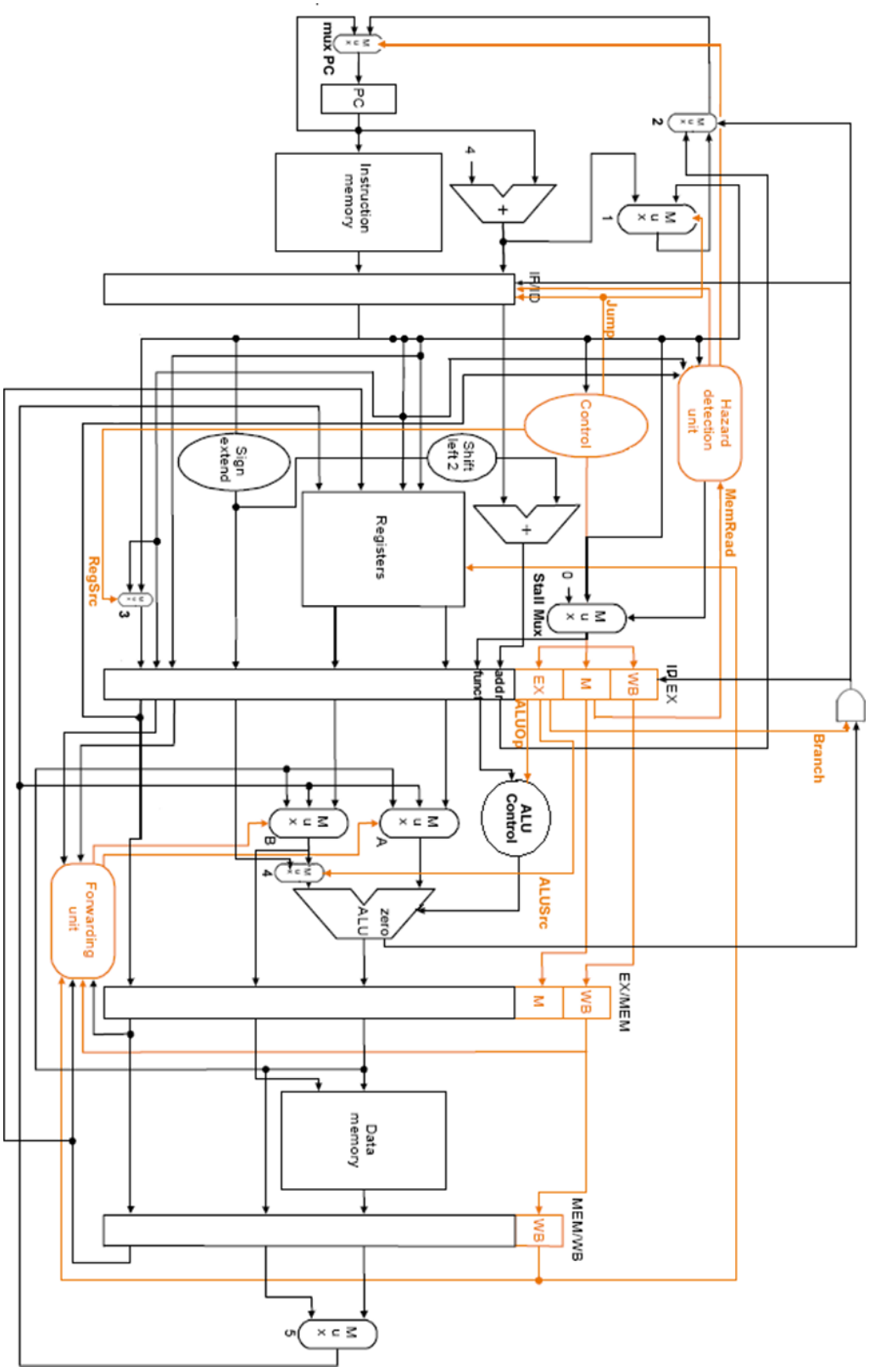

Figure 1. A multi-cycle pipelined datapath