CSCI3420 Computer System Architectures
Project Phase 2
Deadline: 23:59 22th Mar, 2013

1. Introduction

In order to have a more in depth understanding of CPU architecture, you have to write assembly programs, a single-cycle simulator and a multi-cycle pipelined simulator of a simplified CPU. The project is divided into 3 related phases. In phase 2, you have to write a single cycle simulator in standard C, so that you can understanding the datapath design and how instructions are executed in a single cycle machine.

2. Architecture

Figures 1 and 2 in this specification show the datapath you need to simulate. The architecture you are simulating is simplified as follows:

- Each machine word is 32 bits long
- Memory size is exactly 64Kb (16 bits are sufficient to specify an address).
- All store and load operations are word-aligned.
- All instructions are 32 bits (= one machine word).
- The machine is big-endian, i.e. the most significant byte comes first.
- The registers available are listed in table 1.
- You are required to implement only a **reduced subset** of common CPU instructions, as shown in table 2.
- The encoding of the instructions is given in table 3. For the representation, the row shows what is contained in a field, and the header row shows the number of bits of each field.

3. Simulator

In phase 2, you are required to implement a single-cycle simulator in C using only the standard libraries of C89. The simulator should be able to simulate an assembled program (in binary) **cycle-by-cycle**. The points to note are:

- The usage of the simulator should be "`./simulator infile`", where `infile` is the path to the input binary file, of which you should assume only read permission.
- The input to the simulator is a binary file, where every 4 bytes represent a word in big-endian format.
- You may assume that the input file size is always a multiple of 4 bytes.
- Your simulator should treat each word in the input as an instruction and try to simulate it.
- Your simulator should halt when the fetched instruction is 0x00000000.

- Your simulator should also halt when the fetched instruction is **invalid** (which means instructions not shown in Table 3). In this case, you need to **store your student ID** in register **$s26**.
- For simplicity, all `rs` and `rt` for `mult` and `div` should only contain non-negative numbers.
- A skeleton program (with components.h, phase2.c, phase2.h, simulator.c) will be provided to you. You are **ONLY** allowed to edit phase2.c.
- Your program would be compiled using:

“`gcc -o simulator simulator.c phase2.c -Wall`”.

4. Task

You are required to complete the following 23 methods in phase2.c:

No.	Method
1	void run_program_counter(unsigned, unsigned *);
2	void run_pc_adder(unsigned, unsigned, unsigned *);
3	void read_machine_code_file(FILE *);
4	void run_instruction_memory(unsigned, unsigned *);
5	void run_instruction_partitioner(unsigned, unsigned *, unsigned *, unsigned *, unsigned *, unsigned *, unsigned *, unsigned *);
6	int run_control(unsigned, char *, char *, char *, char *, char *, char *, char *, char *, char *);
7	void run_register_read(unsigned, unsigned, unsigned, unsigned *, unsigned *);
8	void run_register_write(unsigned, unsigned, unsigned, unsigned, unsigned, char, char, char, char);
9	void run_alu_control(unsigned, char, unsigned *, char *, char *);
10	void run_alu(unsigned, unsigned, unsigned, char *, unsigned *, unsigned *);
11	void run_and_gate(char, char, char *);
12	void run_or_gate(char, char, char *);
13	void run_sign_extend(unsigned, unsigned *);
14	void run_twobit_shifter(unsigned, unsigned *);
15	void run_address_adder(unsigned, unsigned, unsigned *);
16	void run_data_memory(unsigned, unsigned, char, unsigned *);
17	void run_mux(unsigned, unsigned, char, unsigned *);
18	void run_mult_div(unsigned, unsigned, unsigned, unsigned *, unsigned *);
19	void run_register32(unsigned, unsigned *);
20	void run_sub_alu(unsigned, unsigned, unsigned, unsigned *);
21	void run_shifter64(unsigned, unsigned, char, char, char, unsigned *, unsigned *);
22	void run_test_control(unsigned, unsigned, unsigned, char *, char *, char *, unsigned *);
23	void run_store_sid();

The points to note are:

- Method 1 is completed for you as an example.

- In method 2, you should add the two inputs.
- In method 3, you should read the binary file and save the instructions to the Instruction Memory.
- For methods 4 to 17, each of them corresponds to a hardware component in Figure 1, you have to simulate how that corresponding hardware is working. Noted that component Register is simulated twice in one clock cycle. The calling sequence is implemented in the `main` function in simulator.c.
- In method 18, you should design the calling sequence of methods 18 to 21 of the Multiple-Division Unit (Figure 2) for both multiplication and division.
- For methods 19 to 22, each of them corresponds to a hardware component in Figure 2, you have to simulate how that corresponding hardware is working.
- For method 23, you should store your student ID in register $s26.
- It is FORBIDDENED to declare any global variables and new functions in any files.
- For any control signal lines (Orange lines on Figure 1 and Figure 2), they allow **<u>ONLY ONE</u>** bit signal (either 0 or 1) to be passed.

5. Program testing

To test your program before submission, we provide a reference simulator compiled in linux1 of our department called: **simulator_linux1**. The usage of this program:

```
"./simulator_linux machine_code.asc"
```

where **machine_code.asc** is the assembled program (in binary).

To generate **machine_code.asc**, you can use the provided program: **asm_linux** using linux machine. The usage of this program:

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

where **assembly_code.asm** is the MIPS source file and **machine_code.asc** is the output file of your MIPS code in binary.

Please remind that **simulator_linux** and **asm_linux** can be run in Linux only.

To create a machine code which contains invalid instructions, you can use a binary editor to edit the content in the machine code. For example, you can use the provided program **BZ** to open a machine code, and just add a new invalid instruction or change an instruction to an invalid one.

Please note that students should **not** rely on the two test cases provided only, and you should create your own test cases.

6. Assessment of Correctness

We will test your simulator with various test cases in Linux environment. A sample simulator which is compiled in linux1 in our department is provided for you as a reference of the expected behaviors.

For each test case, a short program segment will be simulated on your simulator. Register contents, Memory contents and critical Control Signals will be checked.

7. Assignment Submission

- Submit only “phase2.c”.
- You **MUST** submit the C file to the submission system on our course homepage (within CUHK network), otherwise, we will **NOT** mark your assignment. The details of how to use the system will be announced later.
- Please limit the file size to be less than 1MB.
- **Plagiarism** will be **seriously punished**.
- Late submission will **NOT** be accepted, and the submission time will be determined according to our submission system.

8. Marking Scheme

The distribution of marks is as follows:

Part	Marks
Multiplication Algorithm	20%
Division Algorithm	20%
Test cases	55%
Invalid Instructions	5%

Register Name	Remarks	Remarks
$zero	Always has value 0	0x00
$ra	Return address	0x01
$fp	Frame pointer	0x02
$sp	Stack pointer	0x03
$hi	High-word	0x04
$lo	Low-word	0x05
$s1, $s2,...,$s26	General purpose registers	0x06,0x07,...

Table 1: Name and Encoding of the Registers

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
Jump Register	jr $s1	goto $s1	Jump to address in the register
Jump And Link	jal label	$ra = PC + 4, goto label	Save the next PC in $ra, and then jump

Table 2: 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*				
Jump Register	jr *rs*	0x0f	*rs*	0			3
Jump And Link	jal *target*	4	*target*				

Table 3: Instruction Binary Representation

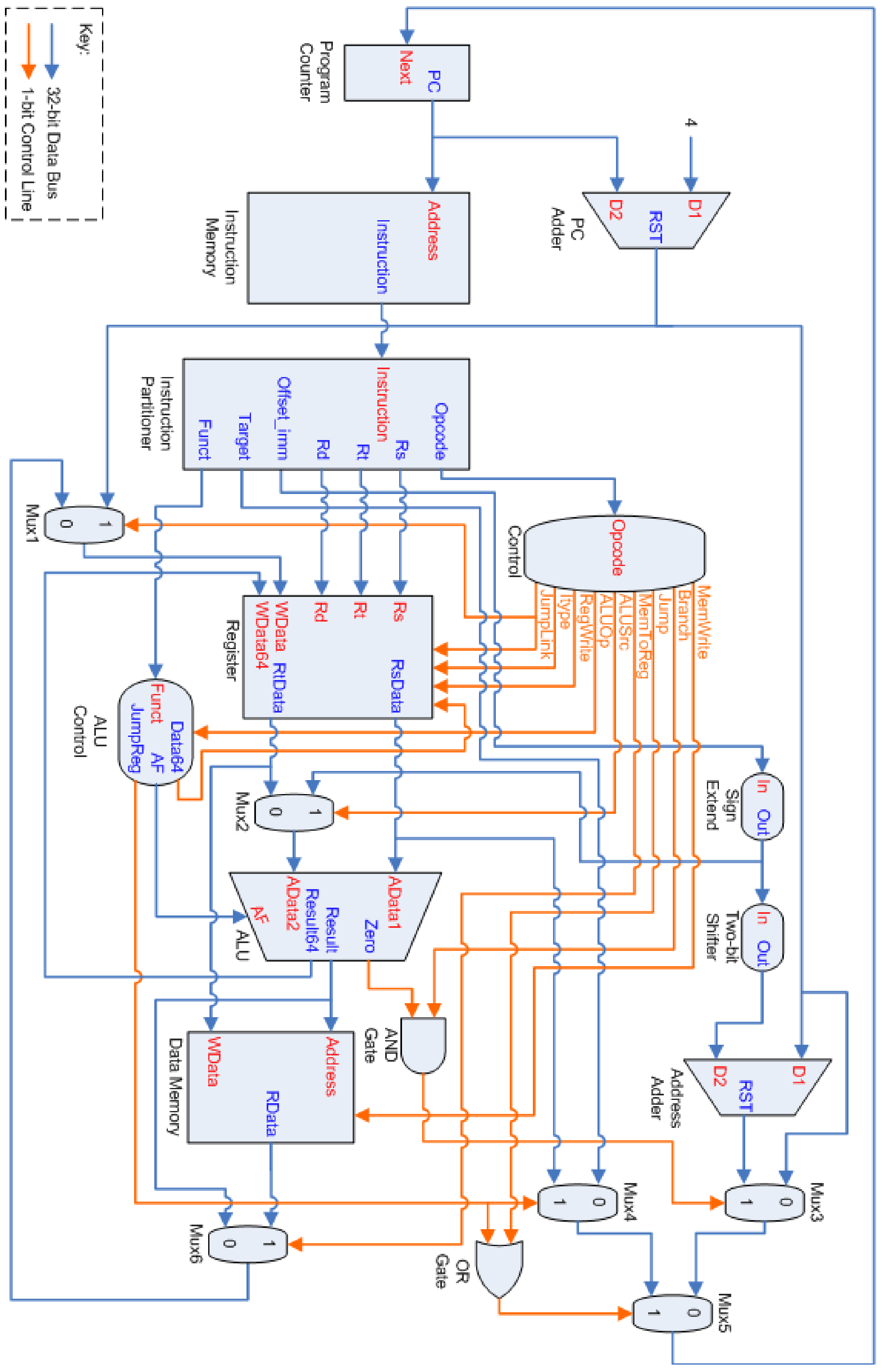

Figure 1. The single-cycle datapath to be implemented

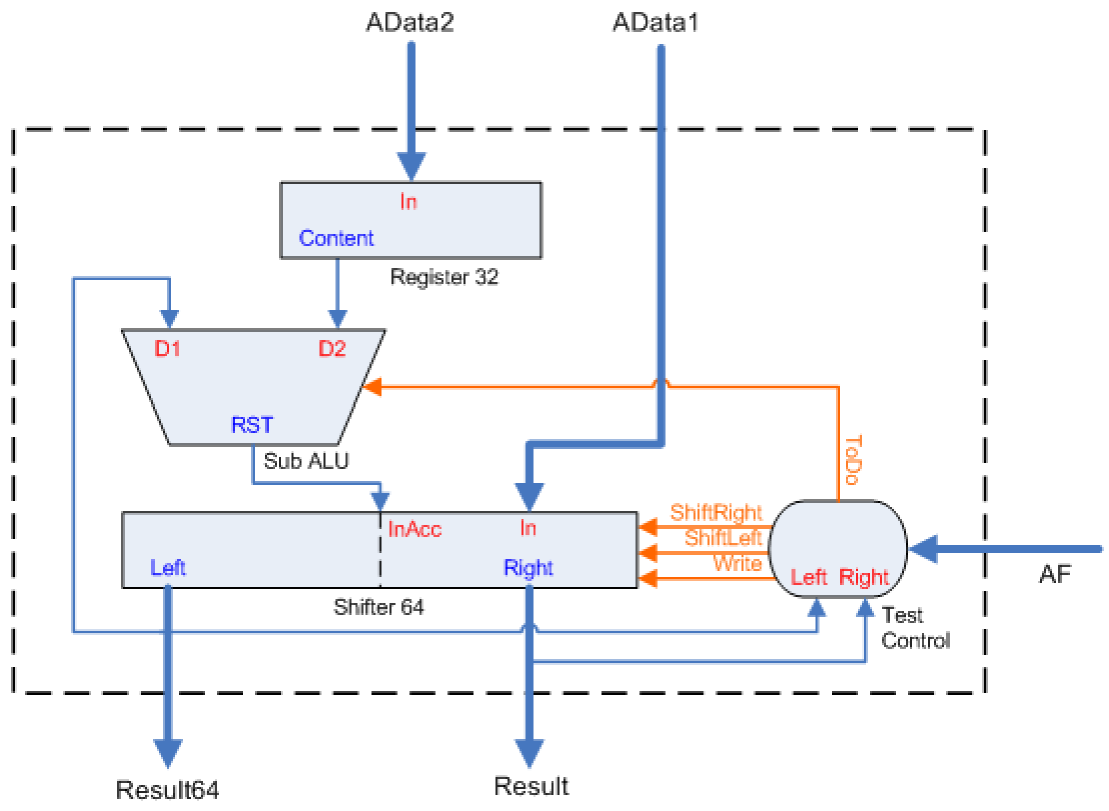

Figure 2. The design of Multiple-Division Unit to be implemented