

CSC3420 Computer System Architectures

Project Phase 1

Deadline: 23:59 14th Feb, 2009

1. Introduction

This course project is intended to let you have a more in depth understanding of CPU architecture, by writing an assembler of a simplified assembly language, a single-cycle simulator and a multi-cycle pipelined simulator of a simplified CPU. The project is divided into 3 related phases. In phase 1, you have to write an assembler in C and write one assembly program.

2. Tasks

In order to encourage real understanding, each of you is required to finish a slightly different task. The variations are:

- 4 sets of opcodes (set 1 to 4)
- 4 kinds of conditional branches – 0: `beq`, 1: `bne`, 2: `blt`, 3: `bge`
- Either 16 registers (hash code 0) or 32 registers (hash code 1)
- 5 assembly programs – 0: Multiplication, 1: Division(Quotient), 2: Division(Remainder), 3: Square Root, 4: Prime Number

Depending on your student ID, you will be assigned to use one set of opcode, implement one of the conditional branches in your assembler and later phases (together with other instructions which are common to all students), have either 16 or 32 registers, write one of the assembly programs (using only the instructions you have implemented). Note that students who are assigned to implement either multiplication or division NEED NOT implement and CANNOT use `mult` and `div`.

The above variations are assigned using the following hashing scheme:

Let your student ID is $S = \{s_1, \dots, s_8\}$, the followings are the 4 hash functions.

For selecting opcode: $hash_value_1 = 1 + \sum_{i=1}^8 s_i \bmod 4$

For selecting jump operation: $hash_value_2 = \left(\sum_{i=1}^8 i \times s_i \right) \bmod 4$

For selecting register number: $hash_value_3 = \left[\left(\sum_{i=1}^8 i \times s_i \right) \bmod (s_8 + 1) + s_4 \right] \bmod 2$

For selecting program segment to code: $hash_value_4 = \left(\sum_{i=1}^8 (9 - i) \times s_i \right) \bmod 5$

Example (if student ID = 09658951):

$$\text{hash_value}_1 = 1 + (0 + 9 + 6 + 5 + 8 + 9 + 5 + 1) \bmod 4 = 4$$

$$\text{hash_value}_2 = (0 \times 1 + 9 \times 2 + 6 \times 3 + 5 \times 4 + 8 \times 5 + 9 \times 6 + 5 \times 7 + 1 \times 8) \bmod 4 = 1 \text{ (bne)}$$

$$\text{hash_value}_3 = [(0 \times 1 + 9 \times 2 + 6 \times 3 + 5 \times 4 + 8 \times 5 + 9 \times 6 + 5 \times 7 + 1 \times 8) \bmod (1 + 1) + 5] \bmod 2 = 0 \text{ (16 registers)}$$

$$\text{hash_value}_4 = (0 \times 8 + 9 \times 7 + 6 \times 6 + 5 \times 5 + 8 \times 4 + 9 \times 3 + 5 \times 2 + 1 \times 1) \bmod 5 = 4 \text{ (Prime Number)}$$

To help you to figure out exactly which variation you are assigned to, a small program will be provided to compute the above hash codes from your student ID.

3. Architecture

The architecture you are simulating is simplified as follows:

- each machine word has 32 bits
- memory size is exactly 64Kb (16 bits are sufficient to specify an address)
- all store and load operations are word-aligned
- every instruction takes one machine word
- big-endian, i.e. the most significant byte come first

The registers of the architecture are:

Register Name (16 registers)	Register Name (32 registers)	Remarks	Encoding
\$zero	\$zero	Always has value 0	0x00
\$ra	\$ra	Return address	0x01
\$fp	\$fp	Frame pointer	0x02
\$sp	\$sp	Stack pointer	0x03
\$s1, \$s2, ..., \$s12	\$s1, \$s2, ..., \$s28	General purpose registers	0x04, 0x05, ...

4. Assembler

You are required to write an assembler in standard C (and CANNOT use extra libraries), for the instruction set to be described below (and your specific variation). The assembler should have the usage:

`asm infile.asm outfile`

4.1 Syntax

For simplicity, the syntax of the assembly language is:

- Comments begin with a sharp (#) and effective till the end of line
- Identifiers (for labels) consists of alphanumeric characters, underscore (_), dots(.), but do NOT start with numbers. The identifiers are *case-insensitive*.
- Instructions (e.g. add, sub) and assembler directives are reserved and therefore CANNOT be used as identifier
- Labels consist of an identifier, followed by a colon (:), and are declared at the beginning of a line
- Numbers appearing in the text are always base 10

4.2 Directives

For simplicity, you are required to implement only one directive, which would be useful for writing your assembly program:

```
.word c1,c2,c3,...,cn
```

which directly puts the n numbers into the n words at this position.

4.3 Instruction Set

You are required to implement only a reduced subset of some usual CPU instructions:

Category	Instruction	Example	Meaning	Remarks
Arithmetic	Add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	3 reg
	Add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	2 reg, 1 constant
	Subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	3 reg
	Multiplication ¹	mult \$s1,\$s2,\$s3	$\$s1 = \$s2 * \$s3$	3 reg
	Division ²	div \$s1,\$s2,\$s3	$\$s1 = \$s2 / \$s3$	3 reg
Logical	And	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	3 reg
	Or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	3 reg
Bit Shifting	Shift Left Variable	sllv \$s1,\$s2,\$s3	$\$s1 = \$s2 \ll \$s3$	3 reg
	Shift Right Variable	srlv \$s1,\$s2,\$s3	$\$s1 = \$s2 \gg \$s3$	3 reg
Data Transfer	Load word	lw \$s1,100(\$s2)	$\$s1 = \text{mem}[\$s2 + 100]$	2 reg, 1 constant
	Store word	sw \$s1,100(\$s2)	$\text{mem}[\$s2 + 100] = \$s1$	2 reg, 1 constant
Conditional Branch ³	Branch on equal	beq \$s1,\$s2,25	If($\$s1 == \$s2$) goto PC + 4 +100	2 reg, 1 offset counted in words
	Branch on not equal	bne \$s1,\$s2,25	If($\$s1 != \$s2$) goto PC + 4 +100	2 reg, 1 offset counted in words
	Branch on less than	blt \$s1,\$s2,25	If($\$s1 < \$s2$) goto PC + 4 +100	2 reg, 1 offset counted in words
	Branch on greater than or equal	bge \$s1,\$s2,25	If($\$s1 \geq \$s2$) goto PC + 4 +100	2 reg, 1 offset counted in words
Unconditional Branch	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>	$\$ra = \text{PC} + 4$ goto <label>	Save the next PC in \$ra, and then jump

¹ Need NOT implement and CANNOT be used for students assigned to write Multiplication or Division.

² Need NOT implement and CANNOT be used for students assigned to write Multiplication or Division.

³ Each student implements only one of the 4, depending on his or her specific variation.

Note that in `addi`, `lw` and `sw`, a label can be used in place of the constant, in which case the resolved address of the label is used as the constant, your assembler SHOULD handle this. In `beq`, `bne`, `blt` and `bge`, a label can be used in place of the offset, in which case the offset should be calculated by the assembler. Also note that for the conditional branches, the offset is positive for jumping forward and negative (in 2's complement form in 16 bits) for jumping backward.

4.4 Instruction Encoding Layout and the Opcodes (and Function codes)

Depending on whether your specific architecture has 16 or 32 registers, the instruction encoding and the opcodes are slightly different:

16 registers:

Mode	8 bits	4 bits	4 bits	4 bits	6 bits	1 bits	5 bits
3 reg	Opcode	Reg 1	Reg 2	Reg 3	Funct	NULL	Shamt
2 reg, 1 constant	Opcode	Reg 1	Reg 2	Constant			
1 constant	Opcode	NULL	NULL	Constant			
1 reg	Opcode	Reg 1	NULL	NULL	NULL		

32 registers:

Mode	6 bits	5 bits	5 bits	5 bits	6 bits	5 bits
3 reg	Opcode	Reg 1	Reg 2	Reg 3	Funct	Shamt
2 reg, 1 constant	Opcode	Reg 1	Reg 2	Constant		
1 constant	Opcode	NULL	NULL	Constant		
1 reg	Opcode	Reg 1	NULL	NULL	NULL	

4 sets of Opcode + Funct (32 registers)

Category	Instruction	Set 1	Set 2	Set 3	Set 4
Arithmetic	<code>add</code>	0x00 / 0x01	0x00 / 0x30	0x00 / 0x04	0x00 / 0x01
	<code>addi</code>	0x02	0x20	0x08	0x03
	<code>sub</code>	0x00 / 0x03	0x00 / 0x10	0x00 / 0x0C	0x00 / 0x07
	<code>mult</code>	0x00 / 0x04	0x00 / 0x08	0x00 / 0x22	0x00 / 0x0F
	<code>div</code>	0x00 / 0x08	0x00 / 0x04	0x00 / 0x32	0x00 / 0x1F
Logical	<code>and</code>	0x00 / 0x10	0x00 / 0x02	0x00 / 0x12	0x00 / 0x3F
	<code>or</code>	0x00 / 0x2F	0x00 / 0x01	0x00 / 0x02	0x00 / 0x02
Bit Shifting	<code>sllv</code>	0x00 / 0x05	0x00 / 0x06	0x00 / 0x0E	0x00 / 0x28
	<code>srlv</code>	0x00 / 0x06	0x00 / 0x05	0x00 / 0x0D	0x00 / 0x30
Data Transfer	<code>lw</code>	0x0A	0x0C	0x06	0x06
	<code>sw</code>	0x08	0x08	0x04	0x05
Conditional Branch	<code>beq</code>	0x21	0x12	0x38	0x39
	<code>bne</code>	0x22	0x11	0x30	0x3A

	blt	0x23	0x14	0x28	0x3B
	bge	0x24	0x18	0x20	0x3C
Unconditional Branch	j	0x10	0x21	0x11	0x17
	jr	0x11	0x22	0x09	0x27
	jal	0x12	0x24	0x19	0x37

4 sets of Opcode + Funct (16 registers)

Category	Instruction	Set 1	Set 2	Set 3	Set 4
Arithmetic	add	0x00 / 0x01	0x00 / 0x30	0x00 / 0x04	0x00 / 0x01
	addi	0x02	0x20	0x08	0x03
	sub	0x00 / 0x03	0x00 / 0x10	0x00 / 0x0C	0x00 / 0x07
	mult	0x00 / 0x04	0x00 / 0x08	0x00 / 0x22	0x00 / 0x0F
	div	0x00 / 0x08	0x00 / 0x04	0x00 / 0x32	0x00 / 0x1F
Logical	and	0x00 / 0x10	0x00 / 0x02	0x00 / 0x12	0x00 / 0x3F
	or	0x00 / 0x2F	0x00 / 0x01	0x00 / 0x02	0x00 / 0x02
Bit Shifting	sllv	0x00 / 0x34	0x00 / 0x14	0x00 / 0x2E	0x00 / 0x20
	srlv	0x00 / 0x35	0x00 / 0x16	0x00 / 0x2D	0x00 / 0x30
Data Transfer	lw	0x8A	0x4C	0x66	0x18
	sw	0x88	0x48	0x64	0x14
Conditional Branch	beq	0xA1	0x52	0x98	0xE4
	bne	0xA2	0x51	0x90	0xE8
	blt	0xA3	0x54	0x88	0xEC
	bge	0xA4	0x58	0x80	0xF0
Unconditional Branch	j	0x90	0x61	0x71	0x5C
	jr	0x91	0x62	0x69	0x9C
	jal	0x92	0x64	0x79	0xDC

For example, for set 2 opcode, 16 registers, the following instruction

```
sub $s1, $s2, $s3
```

should have the encoding

Mode	8 bits	4 bits	4 bits	4 bits	6 bits	1 bits	5 bits
3 reg	Opcode	Reg 1	Reg 2	Reg 3	Funct	NULL	Shamt
	0x00	0x04	0x05	0x06	0x10	0x0	0x00

which, if written out, will be 0x00456400

4.5 Output

After assembling the source assembly program, the result would be a bunch of bytes. Your assembler should output these bytes in **binary** and in **big-endian** format (i.e. the most significant byte comes first).

5. Assembly Program

In addition to the assembler, to help you understand assembly language more, you are required to write a small program in the specific variation of assembly language implemented in your assembler. Each of you has to write only one of the following, depending on your hashing code.

5.1 Multiplication

In this program, two non-negative integers are given in `$s1` and `$s2` before your program starts, when your program reaches the end, `$s1` should contain the *product* of the two integers. For simplicity, you **NEED NOT** consider the problem of overflow, only the lower 32 bits of the result is required. For this program, you **CANNOT** use `mult` and `div`, but can use any other means you feel appropriate to implement the required operation.

5.2 Division – Quotient

In this program, one non-negative integer x is given in `$s1` and one positive integer y is given in `$s2` before your program starts, when your program reaches the end, `$s1` should contain the *quotient* of x/y (i.e. the result of integer division in C). For this program, you **CANNOT** use `mult` and `div`, but can use any other means you feel appropriate to implement the required operation.

5.3 Division – Remainder

In this program, one non-negative integer x is given in `$s1` and one positive integer y is given in `$s2` before your program starts, when your program reaches the end, `$s1` should contain the *remainder* of x/y (i.e. the result of $x \% y$ in C). For this program, you **CANNOT** use `mult` and `div`, but can use any other means you feel appropriate to implement the required operation.

5.4 Square Root

In this program, one non-negative integer x is given in `$s1` before your program starts, when your program reaches the end, `$s1` should contain the floor of the square root of x (i.e. the smallest non-negative integer a such that $a^2 \leq x$. E.g. when x is 8, the answer should be 2, since $2*2 = 4$, but $3*3 = 9$).

5.5 Prime Number

In this program, one positive integer $x \geq 2$ is given in `$s1` before your program starts, when your program reaches the end, `$s1` should be 1 if x is a prime number, 0 otherwise.

6. Marking Scheme

The distribution of marks of phase 1 is as follows:

Part	Marks
Assembler	
Correctly assemble the instructions	40%
Support the full syntax	10%
Can use labels	10%
Correctly handle the directive	10%
Assembly Program	30%

Your submitted program would be compiled on the server *sparc13*, using *gcc*, so please make sure your assembler can be compiled and runs correctly under this setting. A sample assembler and sample simulator will also be provided for you to test the correctness of your assembler and your assembly program.

7. Submission Guidelines

You should zip the following files as `s01234567.zip` and **send the file to ????????**.

- For the assembler
 - All `.c` and `.h` files of your assembler
 - A makefile, which when *make* is executed, should compile your assembler properly
- For the assembly program
 - A `s01234567.asm` file containing your assembly program

Note the following:

- there will be **ZERO tolerance for plagiarism**
- the deadline will **NOT** be postponed, so you are advised to start doing phase 1 early.

8. Hints

Below gives some hints for writing the assembler, you may consult the MIPS reference (Appendix A of the textbook) for an excellent discussion of some of the issues involved in writing an assembler.

8.1 The flow

For a compiler, the usual flow is:

Source program → Tokens → Parse Tree → Object Code → Machine Code

For the assembler you are writing, there is no distinction between object code and machine code since there is no stage of linking. And the parse tree is particularly simple, since an assembly program is line-oriented, which means the parse tree has a linear structure. One of the major works of the assembler is therefore to know for each line:

- Whether there is a label declared, if so, its identifier
- Whether there is an instruction to be assembled, if so, the instruction, the arguments, e.g. reg1, reg2, reg3, constant,...
- Whether it is a assembly directive only, if so, its arguments (if any)

After knowing all these for each line, the lines containing the instructions can then be assembled accordingly. Note that if we know the arguments, then assembling an instruction is a matter of putting suitable parts (the encoding of opcode, function code, registers, constant, offset) into a word, which requires only some bit manipulations. Also note that handling directive lines such as `.word` directives requires only putting the suitable words into the output. When handling lines containing labels, we have to record down the address (you can keep track of the address by counting how many words are in the output, this is particularly simple when every instruction has the same length) of the label, so that when it is referenced later, we know its address for computing the constant or offset.

The flow may therefore look like this:

```

    add $s1, $zero, $zero
loop:
    beq $s2, $zero, quit

```



<i>Inst</i> : add	<i>Reg</i> : s1	<i>Punc</i> : comma	<i>Reg</i> : zero	<i>Punc</i> : comma	<i>Reg</i> : zero
<i>Id</i> : loop	<i>Punc</i> : colon				
<i>Inst</i> : beq	<i>Reg</i> : s2	<i>Punc</i> : comma	<i>Reg</i> : zero	<i>Punc</i> : Comma	<i>Id</i> : quit



<i>Inst</i> : add	<i>Reg1</i> : s1	<i>Reg2</i> : zero	<i>Reg3</i> : zero
<i>Inst</i> : beq	<i>Reg1</i> : s2	<i>Reg2</i> : zero	<i>Constant(label)</i> : quit



0x00004020

0x10800006

8.2 Symbol Table

For the purpose of keeping track of the addresses of the labels, it is a good idea to maintain a symbol table, which contains the labels seen so far and its address (if available). Since in the assembler you are writing, each label cannot be declared more than once, this symbol table is particularly easy to maintain⁴. Also, as your assembler is not required to be really efficient (as opposed to an assembler for production use) and the programs that are going to be assembled contains not many labels, a simple array (with lookups using linear search) may serve as a good symbol table. But you are encouraged to think about what data

⁴ Think about writing a C compiler, where the same identifier can serve different purposes. In this case, maintaining the symbol table requires a bit more work.

structures are good for symbol table in a production assembler or compiler, where there can be many symbols and lookups have to be more efficient than linear search (e.g. hash table, binary tree,...).

8.3 Forward Referencing

One issue for handling labels is the forward referencing problem. This refers to the case that a label has not been seen at the point where it is used; therefore we do not know its address at the time of assembling the instruction. There are two possible ways to solve this issue.

2-pass assembler

In this case, the assembler goes through the source program twice, in the first pass, the purpose is only in knowing what the labels are and their addresses (so you have to be careful about `.word` directives when counting addresses). Then in the second pass, knowing the addresses of all the labels, the assembler can assemble the instructions easily.

1-pass assembler

In this case, the assembler still goes through the source program once. But when it encounters a label which has not been declared (and therefore its address is not known yet), the assembler continue to assemble the instruction (or skip this altogether and go to the next one), and records that this instruction needs fixing because of the unknown label. Later, when the label is declared and therefore its address is known, the assembler can fix the previous recorded instructions. If, after assembling the whole program, there are still instructions unfixed, some labels are referenced but never declared, which is an error of the source program.