NATIONAL UNIVERSITY OF SINGAPORE

CS2100 – COMPUTER ORGANISATION

(Semester 2: AY2016/17)

Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

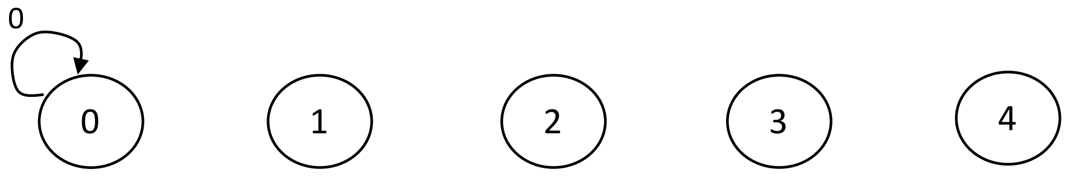
1. This assessment paper consists of **SIX (6)** questions and comprises **TWELVE (12)** printed pages.
2. This is a **CLOSED BOOK** assessment. One handwritten double-sided A4 reference sheet is allowed. Calculators are not allowed.
3. Answer all questions and write your answers in the **ANSWER BOOKLET** provided.
4. Fill in your Student Number with a pen clearly on your ANSWER BOOKLET.
5. You may use pencil to write your answers.
6. Page 9 contains the MIPS Reference Data sheet and page 10 contains the MIPS Datapath.
7. Pages 11 and 12 are for your rough work.
8. You are to submit only the **ANSWER BOOKLET** and no other document.

1. **[10 marks]**
A theme park offers locker rental to its visitors. To use a locker, a visitor deposits 4 tokens, one at a time, into the locker's token slot.

Design a sequential circuit with states *ABC* for the locker's door using a *D* flip-flop for *A*, a *T* flip-flop for *B*, and a *JK* flip-flop for *C*. The circuit consists of 5 states representing the number of tokens a visitor has deposited: 0, 1, 2, 3 and 4 (or, in binary, *ABC* = 000, 001, 010, 011 and 100). The visitor can deposit only one token at a time. When the circuit reaches the final state 4, it remains in state 4 even if the visitor continues to put tokens into the slot.

Let the external input *t* denotes a token.

a. Complete the given state diagram on the Answer Booklet. The state values are shown in decimal. The value on the arrow represents *t*. [2 marks]



b. Write the **simplified SOP expressions** for the flip-flop inputs. [8 marks]

2. **[10 marks]**

a. Given the following Boolean function:

$$\boldsymbol{F(A,B,C,D) = \Sigma m(1, 4, 5, 6, 7, 13)}$$

You are to implement *F* using a single **2-bit magnitude comparator** with no additional logic gates. Note that complemented literals are not available. [5 marks]

b. Given the following Boolean function:

$$\boldsymbol{G(A,B,C,D) = \Sigma m(2, 11)}$$

You are to implement *G* using a single **2×4 decoder** with one-enable and active high outputs, and one 2-input exclusive-OR gate. Note that complemented literals are not available. [5 marks]

3. **[10 marks]**
Study the following MIPS program. Arrays *A* and *B* are integer arrays.

```
     # $s0 contains the starting address of array A
     # $s1 contains the starting address of array B
     add  $s2, $s0, $zero     #inst 1
     add  $s3, $s1, $zero     #inst 2
L1:  lw   $t0, 0($s2)         #inst 3
     lw   $t1, 0($s3)         #inst 4
     bne  $t0, $zero, L2      #inst 5
     beq  $t1, $zero, done    #inst 6
L2:  slt  $t2, $t0, $t1       #inst 7
     beq  $t2, $zero, L3      #inst 8
     sw   $t0, 0($s3)         #inst 9
     sw   $t1, 0($s2)         #inst 10
L3:  addi $s2, $s2, 4         #inst 11
     addi $s3, $s3, 4         #inst 12
     j    L1                  #inst 13
done:
```

a. Give the instruction encoding in hexadecimal for instruction 1 (**add $s2, $s0, $zero**). The opcode for **add** is 0 and the funct value for **add** is 0x20. [2 marks]

b. Give the instruction encoding in hexadecimal for instruction 6 (**beq $t1, $zero, done**). The opcode for **beq** is 0x04. [2 marks]

c. If instruction 13 (**j L1**) is at memory address **0xE0480030**, give the instruction encoding in hexadecimal for instruction 13. The opcode for jump is 0x02. [2 marks]

d. The following are the initial values of the array elements in arrays *A* and *B*.

Array *A*:	3	-1	7	0	2	-5	9	0	-9	1
Array *B*:	2	5	7	3	6	0	8	0	-3	2

Fill in the final values of the elements. [4 marks]

4. **[35 marks]**

Zephyr is a 32-bit stack-based processor with the specifications shown below. In this table, registers "x" and "y" serve as placeholders for actual general purpose registers \$1, \$2, …, \$8, the capital letter "V" refers to a single variable, the capital letter "A" refers to the first element of an array, and the small letter "c" refers to a constant. The capital letters "X" and "Y" refer to the top-most and second elements of the stack respectively. All constants and displacements are 2's complement signed values.

Addressing Architecture:	Stack based.
Number of General Purpose Registers:	Eight (\$1, \$2, …, \$8)
Special registers:	Stack pointer (\$sp) Program counter (\$pc)
Instruction formats:	Fixed length 32-bit instructions
Arithmetic instructions: Arithmetic instructions. X is the top-most operand on the stack, Y is the next operand in the stack.	**ADD**: X and Y are popped off the stack, X+Y are pushed back onto the stack. **SUB**: X and Y are popped off the stack, X-Y is pushed back onto the stack. **MUL**: X and Y are popped off the stack, X*Y is pushed back onto the stack. **DIV**: X and Y are popped off the stack. X/Y is pushed back onto the stack.
Stack instructions: Stack manipulation instructions. Special register \$sp points to the top-most element of the stack. Stacks are assumed to be arbitrarily large, while popping an empty stack will cause an error, but we WILL NOT consider that here. We will assume that the stack is never empty nor full.	**PUSHI c:** Push immediate value c onto the stack. **PUSH \$y**: Push register y onto the stack. **POP \$y**: Pop topmost item on the stack into register y. **ZERO:** Reset \$sp to bottom of stack.

4. (continued)

Load/Store Instructions: These are load and store instructions that get data from memory to registers and vice versa. All addresses are byte addresses.	**LW $y, $x**: Load 32-bit word stored in address pointed to by register x into register y. **SW $y, $x**: Store 32-bit word in register y, into the address pointed to by register x. **LB $y, $x:** Load a single byte stored in the address indicated by register x, into the lowest (bits 7-0) bits of register y. **SB $y, $x:** Store the lowest 8-bits (bits 7-0) of register y into the byte address indicated by register x. **LDI $y, c**: Store immediate constant c into register y. **LDI $y, V**: Store address of variable V into register y. **LDI $y, A**: Store base address of array A into register y. **INCW $y:** Register y is incremented by 4. **DECW $y:** Register y is decremented by 4. **INC $y:** Register y is incremented by 1. **DEC $y:** Register y is decremented by 1.
B-type instructions: These are compare and branch instructions.	**BEQ $x, $y, displ:** Jump to address ($pc+4) + 4*displ if register x == register y. **BNE $x, $y, displ:** Jump to address ($pc+4) + 4*displ if register x != register y **BLT $x, $y, displ:** Jump to address ($pc+4) + 4*displ if x < y **BGT $x, $y, displ:** Jump to address ($pc+4) + 4*displ if x>y

4. (continued)

a. Using the instruction set given above, write the Zephyr assembly language equivalent of this program. Ensure that your code is properly commented. [5 marks]

```
for (i=0; i<5; i++)
   if (x[i] < 3)
      x[i] = x[i] + 5;
```

All offsets in Zephyr are expressed as 16-bit word addresses, while registers are expressed as 3-bit register numbers (000_2=\$1, 001_2=\$2, …, 111_2=\$8). Similarly, all constants in Zephyr are 16-bit long.

There are six classes of instructions:

A: No operands (e.g. ADD)

B: One register operand (e.g. PUSH \$1)

C: One constant operand (e.g. PUSHI c)

D: One register and one constant operand (e.g. LDI \$1, c)

E: Two registers (e.g. LW \$1, \$2)

F: Two registers and a displacement (e.g. BEQ \$1, \$2, displ)

b. Sketch the instruction formats for all 6 classes, assuming that all 32 bits of a Zephyr instruction word are utilized fully, and that we maximize the number of opcode bits possible each time. [12 marks]

c. If we utilize an expanding opcode scheme for Zephyr, what is the maximum number of opcodes possible, assuming that there are at least one instruction in each class? Show your working and reasoning process. Where convenient you may leave your answers in terms of powers of 2. [5 marks]

d. What is the minimum number of opcodes possible, assuming that there are at least one instruction in each class? Show your working and reasoning process. Again, where convenient you may leave your answers in terms of powers of 2. [5 marks]

e. What is the furthest forward distance that you can branch to, in the BEQ, BNE, BLT and BGT instructions? Express your answer in number of instructions. [2 marks]

f. Suppose that we have an array *A* of words (i.e. we access elements of *A* one word at a time). What is the maximum size of *A*, expressed in words? [3 marks]

g. Suppose again that we have an array *B* of bytes (i.e. we access elements of *B* one byte at a time). What is the maximum size of *B*, expressed in bytes? [3 marks]

5. **[15 marks]**
In this question we want to modify the (non-pipelined) MIPS datapath to support two new instructions: BLT and BGT – "branch on less than" and "branch on greater than".

The BLT and BGT instructions are shown below:

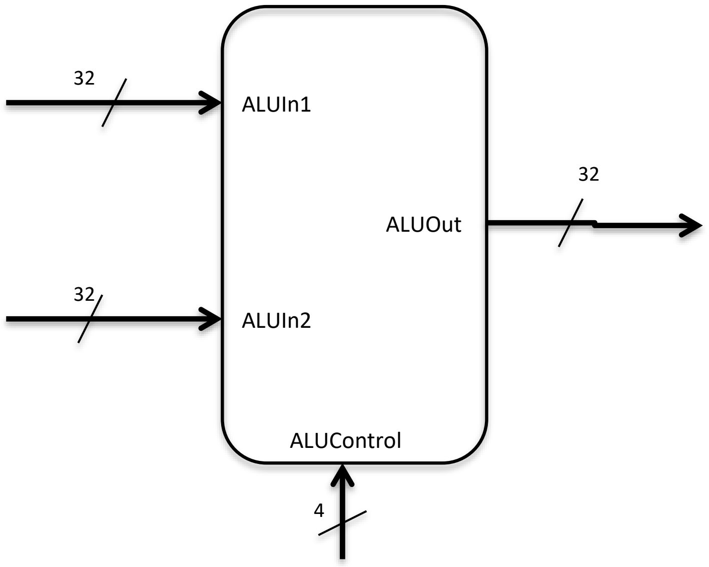
BLT:

0x08	rs	rt	displ

BGT:

0x12	rs	rt	displ

a. The ALU for the MIPS processor is shown below as a single block with two 32-bit inputs, and one 32-bit output. Show, by adding AT MOST ONE 32-input logic gate and any additional wires, how to generate the **IsZero** and **IsNegative** signals. The IsZero signal is 1 when ALUIn1 – ALUIn2 is zero, and the IsNegative signal is 1 when ALUIn1 – ALUIn2 is negative. [4 marks]



b. The CONTROL unit in the datapath must now generate **BranchLess** and **BranchGreater** signals from the instruction bits. Sketch the combinational circuits to generate these signals. [5 marks]

c. For your convenience the MIPS datapath is shown in page 10. Sketch the combinational logic circuit needed to generate the **PCSrc** control signal to support the BEQ, BLT and BGT instructions. [6 marks]

6. **[20 marks]**
Suppose we have a cache that has an access time of 5ns, and a main memory with an access time of 80ns.

a. What is the memory access time when you have a cache hit? [2 marks]

b. What is the memory access time when you have a cache miss? [3 marks]

c. You run some benchmarks on your system and find that 10,000 accesses take a total of 70 microseconds (1 microsecond = 1000 nanoseconds). What is the miss rate of your cache? [5 marks]

Your cache is implemented as a 4-way set associative write-back cache totaling 64KB. Each cache block holds 8 words of 4 bytes each. CPU addresses are 32 bits long.

d. How many bits per set do you require to store the tags? [4 marks]

e. Assuming that the 64KB of cache refers purely to "usable cache" – i.e. cache that is used only to store data or instructions, and not overheads like tag bits, what is the total amount of static RAM that you require to implement this cache? [6 marks]

~~ END OF PAPER ~~~

(The next two pages contain the MIPS Reference Data sheet and the MIPS Datapath.)

MIPS Reference Data

①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15		0
J	opcode	address				
	31 26	25				0

ARITHMETIC CORE INSTRUCTION SET

②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

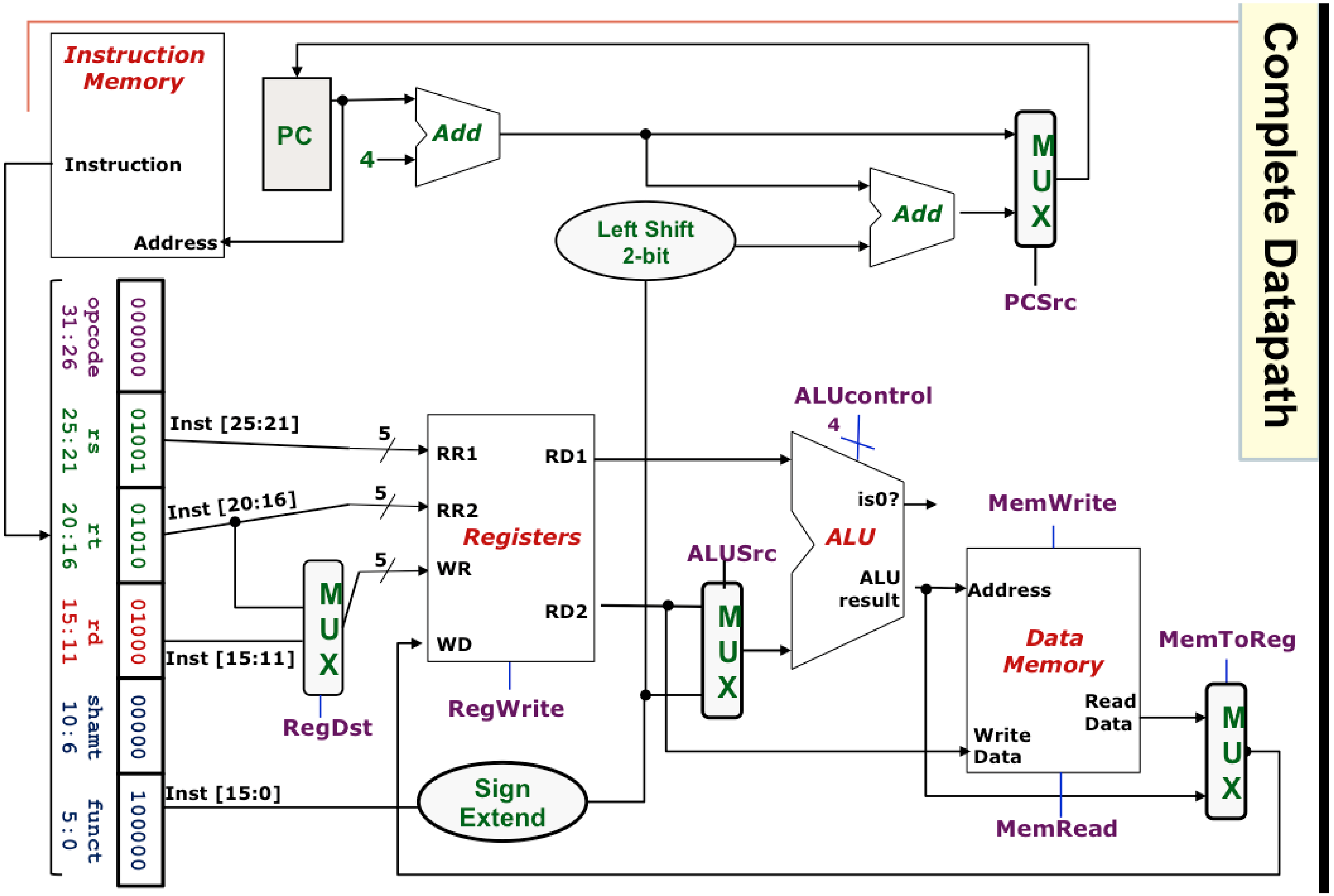
FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15		0

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes


Complete Datapath
Instruction Memory
Instruction
Address
PC
4
Add
Add
Left Shift 2-bit
MUX
PCSrc
opcode 31:26
000000
rs 25:21
01001
rt 20:16
01010
rd 15:11
01000
shamt 10:6
00000
funct 5:0
100000
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
5
5
5
MUX
RegDst
RR1
RR2
WR
WD
RD1
RD2
Registers
RegWrite
Sign Extend
ALUSrc
MUX
ALUcontrol
4
ALU
is0?
ALU result
MemWrite
Address
Data Memory
Read Data
Write Data
MemRead
MemToReg
MUX

(This page is for your rough work.)

(This page is for your rough work.)