CS2100 Computer Organization
Help Sheet for Mid-Term Examinations

Data Representation and Number Systems

N bits can store 2^N values. To represent M values, $\lceil log_2 M \rceil$ bits are required, e.g. 32 values require 5 bits.

Data Type in C		32-Bit Processor	64-Bit Processor
`int`	Size	4 bytes (32 bits)	8 bytes (64 bits)
	Range	-2^{31} to $2^{31}-1$	-2^{63} to $2^{63}-1$
`float`	Size	4 bytes (32 bits)	8 bytes (64 bits)
`double`	Size	8 bytes (64 bits)	16 bytes (128 bits)
`char`	Size	1 byte (8 bits)	
	Range	-2^7 to 2^7. ACSII code has the range from 0 to 255.	

Bit	Byte	Word
A single ‘0’ or ‘1’	8 bits	4 bytes (32 bits) / 8 bytes (64 bits)

Decimal-to-Binary Conversion	
Integers	Use **successive division-by-2** until the quotient is 0. The remainders form the answer: first remainder as the least-significant-bit (LSB), last as the most-significant-bit (MSB).
Fractions	Use repeated **multiplication-by-2** until the fractional product is 0. The carried digits form the answer: first carry is the MSB, last is the LSB.

Base Conversion Table

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Powers of 2

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
1	2	4	8	16	32	64	128	256

Sign-and-Magnitude	
Negation	1-bit sign (0 for +, 1 for -), $(n-1)$-bit magnitude. To negate, invert the sign bit.
Range	$[-(2^{n-1}-1), 2^{n-1}-1]$
Zeros	For $n = 8$, $00000000 = +0_{10}$, $10000000 = -0_{10}$.
1s-Complement	
Range	$[-(2^{n-1}-1), 2^{n-1}-1]$
Zeros	For $n = 8$, $00000000 = +0_{10}$, $11111111 = -0_{10}$.
Negation	$-X = 2^n - X - 1$. Alternatively, just invert every bit.
Addition	For $A + B$, (1) perform binary addition; (2) if there is a carry out of the MSB, add 1 to the result; (3) check for overflow.
Overflow	Occurs if result is opposite sign of A and B. For example, positive + positive = negative, or negative + negative = positive.
Subtraction	$A + B = A + (-B)$. Take 1s-complement of B, add to A.
2s-Complement	
Range	$[-2^{n-1}, 2^{n-1}-1]$
Zeros	There is only one representation of zero which is 00000000.
Negation	$-X = 2^n - X$. Alternatively invert all the bits and then add one.
Addition	For $A + B$, (1) perform binary addition; (2) ignore the carry-out of the MSB; (3) check for overflow.
Overflow	Occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result is opposite sign of A and B.
Subtraction	$A + B = A + (-B)$. Take 2s-complement of B, add to A.
Excess-K	
Format	Add K (bias / offset) to X, then represent the result as binary.

IEEE-754 Floating Point Representation

1-bit Sign: 0 for positive number, 1 for negative number.
8-bit Exponent: Excess-127.
23-bit Fraction / Mantissa: Normalized to 1.X and take X only.

$(-39.625)_{10} = (-100111.101)_2 = (-1.00111101)_2 \times 2^5$.
Hence, Sign Bit = 1, Exponent Bit = 5 + 127 = 132, Fraction Bit = 0011 1101 0000…

Functions and Pointers

Declaration of Pointers		
`int x; int *ptr;` `ptr = &x;`	`ptr` is an `int` pointer. All pointer variables store memory addresses. `ptr` points to `x`. “&” gives the address of a variable.	
Manipulation of Pointer Content		
`int *ptr2;` `ptr2 = ptr;`	`ptr2` is also an `int` pointer. Content of `ptr` is copied over to `ptr2`.	
Dereferencing of Pointers		
`*ptr = 1234;` `*ptr = *ptr + 4321;`	Same as `x = 1234;` Same as `x = x + 4321;`	We follow the address stored in the pointer variable and manipulate the destination.

Parameter Passing into Functions	
Pass-By-Value	• Simple data types (`int`, `float`, `char`) and Structures are passed-by-value. • Cannot change the actual parameter.
Pass-By-Address	• Arrays are passed-by-address. • Requires the caller to pass in the address of variables using “&”. • Requires dereferencing of pointers in the function.

Arrays

Declaration of Arrays

```
int myArray[3];
int myArray[3] = {1,2,3};
int myArray[3] = {1};
int myArray[3] = {0};
```

Random values for all elements in array.
Initialize array during declaration using an initialization list.
If initialization list shorter than array size, rest of values is zero.
Use `{0}` to initialize all array items to zero.

Pointers and Arrays

```
int ia[3] = {3,5,7};
int *ptr;
ptr = ia;
ptr[1] = 333;
ptr = &(ia[1]);
ptr[1] = 4444;
```

The array name by itself is the same as the address of the 0th element.
Manipulation of an element at index 1 in the array.

Arrays as a Function Parameter

```
void printLessThan(
int a[], int size, int criteria) {…}
printLessThan(array, 5, 15);
```

To pass an array as an argument into a function, use the name of the array. In effect, the address of `array[0]` is passed into the function.

Declaration of Strings

We use a character array to store multiple characters, and add a special terminator character `‘/0’` at the end.

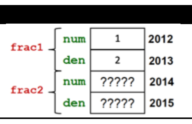
```
char a[6] = {‘H’,’e’,’l’,’l’,’o’,‘!’};
char b[7] = “Hello!”
```

String constants are surrounded by a double quote. Remember to count the terminator when declaring size of the string.

Structure

Defining and Initializing a Structure

```
struct Fraction {
    int num;
    int den;
}
struct Fraction frac1 = {1,2};
struct Fraction frac2;
```



Each structure variable has an independent set of the fields. The fields of a structure is placed in **adjacent locations** in memory.

Joshua Chew Jian Xiang

Structure Passed As a Parameter into a Function

Pass-By-Value:

```
void printFrac(struct Fraction inFrac) {…}
printFraction( myFraction );
```

A copy of the actual argument will be made.

Pass-By-Address:

```
void readFrac(struct Fraction *fptr) {
    int n, d;
    scanf(“%d%d”, &n, &d);
    fptr->num = n; // (*fptr).num = n;
    fptr->den = d; // (*fptr).den = d;
}
readFraction(&myF);
```

To allow function to modify the actual argument, use the **indirect field selector** ‘->’.

Introduction to MIPS

In the MIPS assembly language, there are **32** registers, each 32-bit (4-byte) long. Each word contains 32 bits (4 bytes). Hence, memory registers are 32-bit long.

Several Operations and their Uses	
`addi $s0, $s0, 4`	**Increment / Decrement Operations:** Constant ranges from $[-2^{15}$ to $2^{15}-1]$ and is in **2s-complement**.
`add $s0, $s1, $zero`	**Assignment Operations:** Equivalent to `f = g;` and the pseudo-instruction `move $s0, $s1`
`sll $t2, $s0, 4`	Equivalent to multiplying by 2^n.
`srl $t2, $s0, 4`	Equivalent to getting the quotient when you divide by 2^n. Constant ranges from $[0$ to $2^5-1]$.
`and $s0, $t1, $t2`	**Masking Operations using Bitwise AND:** Place 0s on the places to be ignored → bits will turn into 0s. Place 1s for interested positions → bits will remain the same as original.
`or $t0, $t1, $t2`	Can be used to force certain bits to 1s.
`nor $t0, $t1, $t2`	Can make the **NOT** operation by NOR-ing with `$zero`.
`xor $t0, $t1, $t2`	Can make the **NOT** operation by XOR-ing with all 1s.
`lui $t0, 0xAAAA` `ori $t0, $t0, 0xF0F0`	**Large Constants:** Use “load upper immediate” to set the upper 16-bits. Note that `lui` also clears the lower 16 bits. Use “or immediate” to set the lower-order bits.

Regarding the main memory of a computer,
Each location in the main memory has an address. Given a k-bit address, the address space is of size 2^k.
Using distinct memory address, we can access a single byte (**byte addressable**), or a single word (**word addressable**).
Word Alignment: Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. For example, if a word consists of 4 bytes, then aligned words begin from byte 0, 4, 8, …
The main memory contains 2^{30} memory words.
MIPS uses byte addresses, so consecutive word addresses differ by 4.
Each MIPS instruction is fixed-length 32-bits.

R-format (`op $r1, $r2, $r3`)

Case 1: For `add, sub, and, slt, etc.` the format is `add $rd, $rs, $rt`.
Case 2: For `srl, sll, etc.` the format is `sll $rd, $rt, shamt` and the `rs` field is left as zero.

`opcode`	6	Partially specifies the instruction, equals to 0 for all R-format instructions.
`rs`	5	**Source Register:** Specify register containing the first operand.
`rt`	5	**Target Register:** Specify register containing the second operand.
`rd`	5	**Destination Register:** Specify register which will receive result of computation.
`shamt`	5	**Shift Amount:** Amount a shift instruction will shift by. Set to zero if no shifting.
`funct`	6	Combined with `opcode`, will exactly specify the instruction.

I-format (`op $r1, $r2, Immd`)

Case 1: For instructions involving signed integers or just 16-bit patterns as the Immediate, the format is `addi $rt, $rs, Immd`.
Case 2: For memory instructions, the format is `lw $rt, Immd($rs)`.

`opcode`	6	No `funct` field, so `opcode` uniquely specifies an instruction.
`rs`	5	**Source Register:** Specifies the source register operand (if any).
`rt`	5	**Target Register:** Specifies the register to receive result.
`Immd`	16	**Immediate Value:** 16 bits → can represent a constant of up to

The **Program Counter (PC)** is a special register that keeps address of instruction being executed in the processor. Instructions are word-aligned, and the `Immediate` value in branch instructions is interpreted as the number of words. We can branch to $\pm 2^{15}$ words from the PC.

$$PC = \begin{cases} PC + 4, if\ the\ branch\ is\ not\ taken \\ (PC + 4) + (Immediate \times 4), if\ the\ branch\ is\ not\ taken \end{cases}$$

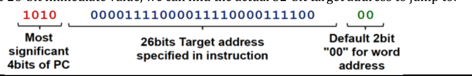
PC-Relative Addressing

For branch instructions, the format is `beq $rs, $rt, Label`.
The `Immd` value is equal to the number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.

J-format (`op, Immd`)

`opcode`	6	The `opcode` for the `j` instruction.
`target address`	26	We can only specify 26 bits of a 32-bit address.

From the 26-bit Immediate value, we can find the actual 32-bit target address to jump to:


1010
00001111000011110000111100
00
Most significant 4bits of PC
26bits Target address specified in instruction
Default 2bit "00" for word address

Maximum jump range: 0x0FFFFFFC

Processor: Datapath

5-Stage MIPS Instruction Execution Cycle

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, label`
Instruction **F**etch	Use the PC to fetch the instruction from memory. Then, increment the PC by 4 to get the address of the next instruction.		
Decode and Operand Fetch	Read `[$1]` as `opr1` Read `[$2]` as `opr2`	Read `[$1]` as `opr1` Use `20` as `opr2`	Read `[$1]` as `opr1` Read `[$2]` as `opr2`
Execute	`Result = opr1 + opr2`	`MemAddr = opr1 + opr2`	`Taken = (opr1 == opr2)?` `Target = PC + Label*`
Memory Access		Use `MemAddr` to read from memory	
Result **W**rite	`Result` stored in `$3`	Memory data stored in `$3`	`if Taken: PC = Target`

Clock Signal

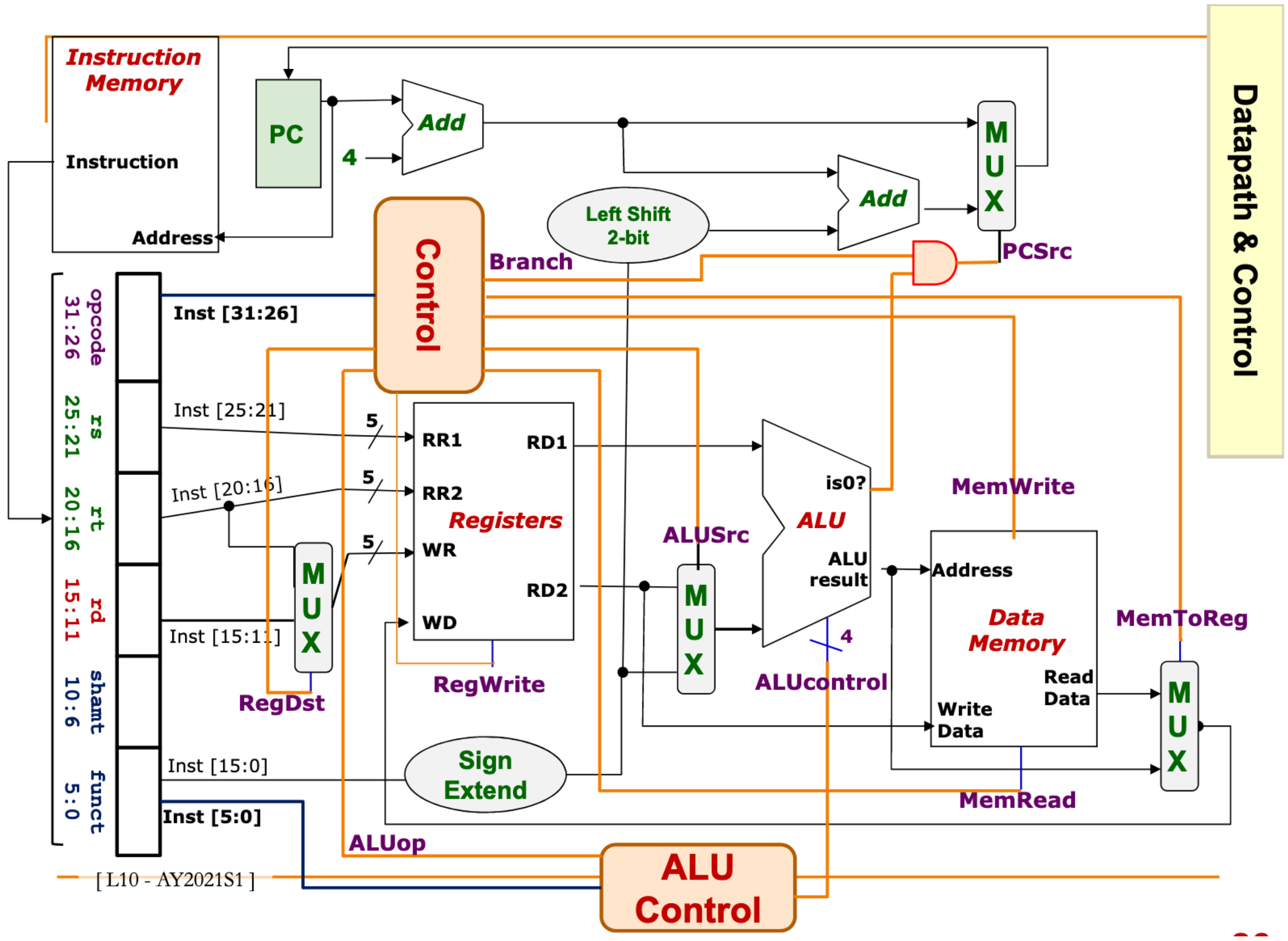
Processors utilize a stable clock signal (square wave) for instruction execution.
Cycle Time / Clock Period: Duration between two consecutive rising edges, measured in seconds
Clock Rate / Clock Frequency: $\frac{1}{cycleTime} = \frac{\#cycles}{second}$, measured in Hz (1 Hz = $\frac{1\ cycle}{second}$)
The PC is read during the first half of the clock period, and the next rising clock edge

Processor: Control

Signal	False (0)	True (1)
`RegDst`	Write register = Inst[20:16]	Write register = Inst[15:11]
`RegWrite`	No register write	New value will be written
`ALUSrc`	opr2 = Register Read Data 2	opr2 = SignExt(Inst[15:0])
`MemRead`	Not performing memory read access	Read memory using ***Address***
`MemWrite`	Not performing memory write operation	memory[Address] ← Register Read Data 2
`MemToReg`	Register write data = ALU Result	Register write data = Memory read data
`PCSrc`	Next PC = PC + 4	Next PC = SignExt(Inst[15:0])<<2 + (PC + 4)

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111



While Loop	If Loop

`while (condition) {`
`    …;`
`}`
	`Loop:`
	`    if (!condition) branch to Exit;`
	`    do stuff;`
	`    j Loop;`
	`Exit:`

Joshua Chew Jian Xiang