CS2100 Midterms Cheatsheet 23/24 S1

C programming

- `%5d`: width of 5, right justified (123 → 00123)
- `%.3f`: 3 decimal places
- no semicolons behind defined constants: `define`

Operator type	Operator	Associativity
Primary expression	(), expr++, expr--	Left to right
Unary	*, &, +, -, ++expr, --expr	Right to left
Binary	*, /, %	Left to right
	+, -	
	<, > ,<=, >=	
	==, !=	
	&&	
	||	
Ternary	?, :	
Assignment	=, +=, -=, *=, /=, %=	Right to left

Pointers

- `%p` for pointers, address in hexadecimal(0x) format
- p = p + n is incrementing the address by the size of the data type multiplied by `n`.
- a_ptr = &a, hence `*a_ptr = a`
- (*player_ptr).name = `player_ptr->name`
- *player_ptr.name is **wrong** as it's treated as `*(player_ptr.name)`
- Declaring an array of pointers: `type *a[]`

Struct

```
typedef struct {
    int numer[1];
    int *denom;
} rational;
```

- If passed into **function**: entire stucture is **copied**
  `*x.numer = 3` won't change original `numer`
  `*x.denom = 5` change original `denom` as it is a pointer

Number Systems & Data Representation

Sizes of data/types

- byte: 8 bits; nibble: 4 bits (half-byte)
- word: multiple bytes (1, 2, 4) (for MIPS it's 4)
- `int`: 4 bytes (1 bit for sign, 31 for magnitude)
- `float`: 4 bytes; `double`: 8 bytes; `char`: 1 byte

Representation & Complements

- Convert decimal whole numbers to base R : divide by R, first remainder is LSB, last is MSB
- Convert decimal fractions to base R : multiply by R, first carry is MSB, last is LSB
- base R to base R^N: partition in groups of N e.g groups of 4 for base 2 to base 16
- 1s & 2s complement: MSB 0 = +ve, 1 = -ve
- Convert to R-1s complement : Flip the digits; `digit = R - digit`
- Convert to R-2s complement : Flip the digits, then add 1 to the number
- Shortcut trick: right to left, copy all 0s until hit the first '1', then invert everything else to the left after
- 1s complement range:$-(2^{n-1}-1)$ to $(2^{n-1}-1)$
- 2s complement range:$-(2^{n-1})$ to $(2^{n-1}-1)$
- Sign Extension: fill up front part with sign bit

Excess & IEEE 754

- Convert to excess X: Take number minus X (0 refers to -x)
- IEEE 754 Representation: $sign|exponent|mantissa$
- Single-precision float has 1 bit sign, 8 bit excess-127 exponent, 23 bit mantissa (normalized with a leading bit 1 i.e the mantissa is the X in 1.X)
- Double has 1 bit sign, 11 bit excess-1023 exponent, 52 bit mantissa
- Example: $-6.5_{10} = -110.1_2 = -1.101_2 * 2^2$
  sign=1; Exponent = 2+127 = 129;
  mantissa = 10100000000000000000000

Operations with binary numbers

- 2s complement addition: Simply add & ignore carry out of MSB
- 2s complement subtraction: take 2s complement of number to be subtracted, then do 2s addition.
- 1s complement addition: Add; If there is a carry out, add 1 to the result
- 1s complement subtraction: take 1s complement of number to be subtracted, then do 1s addition.
- check for **overflow: If result is opposite sign of both operands (that have the same sign)**

MIPS

logical Operations

- `sll/rll`: shifting by n bits; multiply/divide by 2^n
- `and`: used for masking operations , 0 for positions to be ignored, 1 for interested positions
- `or`: forcing certain bits to be 1
  note: ori is **not** sign-extended:
  e.g. `ori $t0, $t1, 0xFFFF` : upper 16-bites are all 0s!
- `nor`: `nor $t0, $t0, $zero` = `NOT` operation
- `xor`: `xor $t0, $t0, $t2` = `NOT` operation ($t2 contains all 1s)
- `lui & ori`: `lui` upper 16-bits, `ori` lower-order bits
- note: only `addi` immediate use 2s complement (range: $-2^{15} to -2^{15}-1$)
- `nop`: as long as RegWrite, MemWrite are both 0. *or* `sll $zero, $zero, 0`

Memory

- Given a k-bit address, can have up to 2^k addresses
- n-bit Architecture: word size is n
- MIPS: word = 4-bytes (32 bits)
- 32 registers, each 32-bit (4-byte) long
- Each word contains 32-bit (4-byte)
- Memory addresses are 32-bit long
- Max byte address is 2^{32} = Mem[4294967292]
- 2^{30} memory words : $\frac{2^{32} bytes}{2^2}$

Memory Instructions

- Only `load` and `store` instructions can access data in memory
- `lw $t0, 4($s0)` : Memory Address = `$s0 + 4`
  Memory word at Mem[`s0 + 4`] is loaded into `$t0`
- `sw $t0, 12($s0)` : Memory Address = `$s0 + 12`
  Content of `$t0` is stored into word at Mem[`s0 + 12`]
- Load byte (`lb`) & store byte (`sb`) uses similar format offset no longer needs to be mulitple of 4

Control Flow instructions

- `beq & bne`
  - `beq $r1, $r2, L1` : go to L1 if R[$r1] == R[$r2]
  - `bne $r1, $r2, L1` : go to L1 if R[$r1] != R[$r2]
  - Condition inversion to write shorter code
- `slt`
  - `slt $t0, $s1, $s2`
    if R[$s1] < R[$s2]: $t0 = 1
    if R[$s1] >= R[$s2]: $t0 = 0
  - `slt $t0, $s2, $s1`
    if R[$s2] < R[$s1] = R[$s1] >= R[$s2] : $t0 = 1
    if R[$s2] >= R[$s1] = R[$s1] < R[$s2], $t0 = 0
- `j`
  - `j L1` == `beq $s0, $s0, L1`
  - labels are **not** counted as instructions

Loops without Pointers

```
Address of A[] → $t0
Result → $t8
i → $t1

      addi $t8, $zero, 0
      addi $t1, $zero, 0
      addi $t2, $zero, 40
loop: bge  $t1, $t2, end
      sll  $t3, $t1, 2
      add  $t4, $t0, $t3
      lw   $t5, 0($t4)
      bne  $t5, $zero, skip
      addi $t8, $t8, 1
skip: addi $t1, $t1, 1
      j loop
end:
```

```
Comments

# end point

# i * 4
# $t4 ← &A[i]
# $t5 ← A[i]

# result++
# i++
```

Loops with Pointers

```
Address of A[] → $t0
Result → $t8
&A[i] → $t1

      addi $t8, $zero, 0
      addi $t1, $t0, 0
      addi $t2, $t0, 160
loop: bge  $t1, $t2, end
      lw   $t3, 0($t1)
      bne  $t3, $zero, skip
      addi $t8, $t8, 1
skip: addi $t1, $t1, 4
      j loop
end:
```

Comments

NOTE:
Consider the code using pointer arith:

```
result = 0;
ptr = A;
end = &A[40];
while ( ptr < end ) {
    if ( *ptr == 0 )
        result++;
    ptr++;
}
```

The resulting MIPS looks like the code on the left.

Encoding: R, I, J format

- **R**: $Opcode(6), rs(5), rt(5), rd(5), shamt(5), funct(6)$
- **I**: $Opcode(6), rs(5), rt(5), Imm(16)$
- For branch, Imm is the relative number of *words* to jump(with respect to $PC+4$), in 2s complement
  Bitwise operations(`andi, ori, xori`) uses raw binary(no negatives)
- **J**: $Opcode(6), Address(26)$ (Pseudo-direct addressing)
- First 4 bits are assumed to be 4 MSBs of PC+4. Last 2 bits assumed to be 0 (because of word addressing)
- Maximum jump range: 2^{28} = 256MB

Instruction Set Architecture

Architectures & Endianness

- Complex Instruction Set Computer(CISC): small prog size, complex implementation
- Reduced Instruction Set Computer(RISC): MIPS, instruction set small, burden on software to combined simpler operations
- Von Neumann: Data(operands) stored in memory
- Stack : operands are on top of stack
- Accumulator: One operator is in the accumulator (a special register) : uses implicit operands
- Memory-memory: all operands in memory
- General-purpose Register: all operands in registers (MIPS) : only explicit operands
- Endianness: relative ordering of bytes in multi-byte word, NOT word itself
  `Big-endian`: Most significant byte stored in lowest address (MIPS)
  `Little-endian`: Least significant byte stored in lowest address

Opcode encoding Question

- To maximize, reserve 1 instruction for lesser-bit instruction types.
- To minimize, reserve all but 1 instruction for lesser-bit instruction types
- Example: 3 types of instructions, A: 4-bit opcode, B: 7-bit opcode, C: 8-bit opcode
  min N (by maxing A): $(2^4-1)+(2^3-1)+2^1$
  max N (by min A): $1+1+((2^4-1)*(2^3-1))*2^1$

Processor

- Datapath: Collections of components that process data, performs arithmetic, logical and memory operations
- Control: Tels datapath, memory and I/O devices what to do according to control signals
- Fetch Stage:
  - Program Counter(PC) to fetch instruction from Memory
  - PC is read during first half of clock period and is updated with PC+4 ((32 bits → 4 bytes)) at next rising clock edge
  - Output: 32-bit instruction in binary
- Decode Stage:
  - Register file: 32 registers, each 32-bit wide
  - Control signal `RegWrite`: 1 = Write, 0 = No Write
  - Output: Operands 1 and 2 for the ALU
- ALU(Arithmetic-Logic Unit) Stage:
  - calculating (`add,sub,sll,and,or`)
  - Memory operations (`lw,sw`): address calculation
  - Branch (`bne,beq`): Register comparison and target address calculatiion
  - `beq $9, $0, 3` :
    check R[$9] - R[$0] = 0; `isZero?` signal = 1 if true
  - `PSCrc`:
    if `PSCrc` = 0: next instruction (pc+4)
    if `PSCrc` = 1: jump branch (pc+4 + immediate*4)
  - Output: Result computed by the ALU and the 1-bit signal `isZero`

- Memory Stage:
  - Memory instruction: `MemRead, MemWrite`, ALU result gives the address, WriteData from RD2
  - Non-Memory instruction: `MemToReg`: determine if result came from memory or ALU
  - Output: Data read from memory. For some instructions, this stage is not executed
- WriteBack:
  - Write the result of computation back into a register
  - Exceptions are stores, branches, jumps
  - No Output. Only data from ALU or memory is written into WR

Control Signals

Control Signal	Execution Stage	Purpose
`RegDst`	Decode/Operand Fetch	Select the destination register number
`RegWrite`	Decode/Operand Fetch RegWrite	Enable writing of register
`ALUSrc`	ALU	Select the 2nd operand for ALU
`ALUcontrol`	ALU	Select the operation to be performed
`MemRead` / `MemWrite`	Memory	Enable reading/writing of data memory
`MemToReg`	RegWrite	Select the result to be written back to register file
`PCSrc`	Memory/RegWrite	Select the next PC value

Control Unit

- `RegDst`
  False(0): Write register = Inst[20:16] (`rt`)
  True(1): Write register = Inst[15:11] (`rd`)
- `RegWrite`
  False(0): No register write
  True(1): New value(`WD`) will be written into (`WR`)
- `ALUSrc`
  False(0): Operand2 = RD2 (`rt`)
  True(1): Operand2 = signExt(Inst[15:0]); (32-bit immed)
- `MemRead`
  False(0): Not performing memory read access
  True(1): Read memory using ***Address***
- `MemWrite`
  False(0): Not performing memory write operation
  True(1): memory[*Address*] → `RD2`
- `MemToReg`
  True(1): `WD` = Memory read data
  False(0): `WD` = ALU result
  Note: The input of MUX is **swapped**: 1 on top, 0 below
- `PSCrc`
  Idea: If instruction is branch AND taken, then
  False(0): Next PC = PC + 4
  True(1): Next PC = SignExt(Inst[15:0]) << 2 + (PC + 4)

Truth Tables

YES

INPUT A	OUTPUT
0	0
1	1

NOT

INPUT A	OUTPUT
0	1
1	0

AND

A	B	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

OR

A	B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

XOR

A	B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0

NAND

A	B	OUTPUT
0	0	1
1	0	1
0	1	1
1	1	0

NOR

A	B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

A	B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1

PYPs Qns

Expanding Opcode

Section 4. Fill In The Blanks (Expanding Opcodes)

ceil(log2 16) = 4 register bits required

In this section we assume a processor with a fixed 16-bit instruction length, 16 registers, and the following 3 instruction classes:

Class A: Two registers. 8 bits instruction
Class B: One register. 12 bits instruction
Class C: One register, one 8-bit immediate value. 4 bits instruction

In all cases we assume that the encoding space for opcodes is fully utilized.

12. The minimum number of opcodes that can be encoded on this machine is 46.

13. The maximum number of opcodes that can be encoded on this machine is 3826.

No. Register bits = $\lceil log_2 n \rceil = \lceil log_2 16 \rceil = 3$
Min instructions: $(2^4 - 1) + (2^4 - 1) + 2^4 = 46$
Max instructions: $1 + 1 + ((2^4 - 1) * 2^4 - 1) * 2^4$
$= 2 + (15 * 16 - 1) * 2^4 = 3826$
note: qn: address 128 **bytes** of memory = $\lceil log_2(128 * 2^2) \rceil$, cause bytes is 4 bits!

Excess-N representation

5. We consider a 24-bit signed number system in excess-1048576. In this number system, there are **1048576** negative numbers and **15728640** non-negative numbers (2 marks).

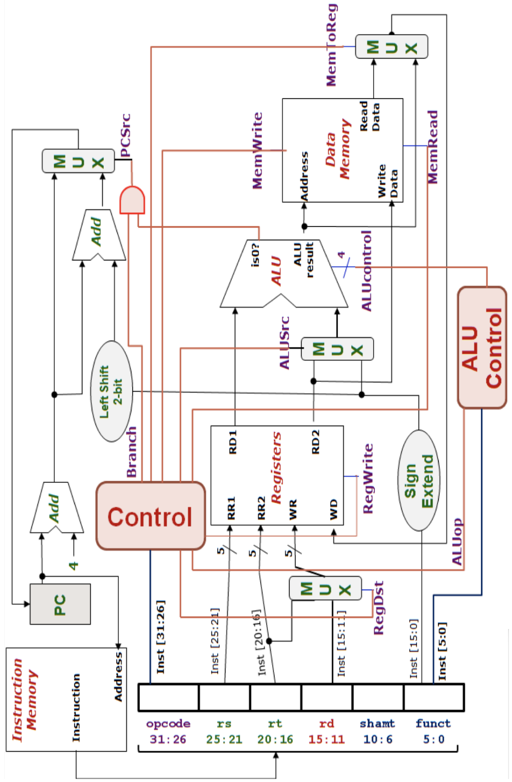
Check: Excess-1048576 means the most negative number is -1048576. From -1048576 to -1 there are 1048576 numbers. With 24 bits we have 2^24 = 16777216 numbers, of which 1048576 are negative, so 16777216 – 1048576 = 15728640 numbers are non-negative.

Diminished Radix (r-1)'s Complement: If we are given a number x in base-r having n digits:
r-1's complement: $r^n - x - 1$
r's complement: $r^n - x$
Convert binary with big powers → decimals
$-1.11111111 * 2^{31} = 111111111 * 2^{23} = 2^{23} + 2^{24} + \ldots + 2^{31}$
Iterate through bits in char(1 char 8 bits)

```
#include <stdio.h>

int main() {

    char *msg = "cs2100";
    char *c = msg;

    int d = 0;

    do {
        char e = *c;
        while(e) { // Q1
            d += (e & 1); // Q2
            e = e >> 1;
        }

        c++;
    } while(*c);

    printf("d is %d\n", d); // Q3
}
```

Iterates over the bits of the character e. It continues until e becomes zero (i.e., all bits have been processed).

iterate over all characters in msg until '\0'



4-bit system

Positive values: Value	Sign-and-Magnitude	1s Comp.	2s Comp.	Negative values: Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111	-0	1000	1111	-
+6	0110	0110	0110	-1	1001	1110	1111
+5	0101	0101	0101	-2	1010	1101	1110
+4	0100	0100	0100	-3	1011	1100	1101
+3	0011	0011	0011	-4	1100	1011	1100
+2	0010	0010	0010	-5	1101	1010	1011
+1	0001	0001	0001	-6	1110	1001	1010
+0	0000	0000	0000	-7	1111	1000	1001
				-8	-	-	1000

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

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

Bit value:	32	16	8	4	2	1
Sum:	63	31	15	7	3	1
Power of 2:	5	4	3	2	1	0
Bit	6	5	4	3	2	1
Power negative:	0.015625	0.03125	0.0625	0.125	0.25	0.5

Bit value:	4096	2048	1024	512	256	128	64
Sum:	8191	4095	2047	1023	511	255	127
Power of 2:	12	11	10	9	8	7	6
Bit	13	12	11	10	9	8	7
Power negative:			.00048828125	.0009765625	.001953125	.00390625	.0078125

	opcode	operand	operand
Type-A	6 bits	5 bits	5 bits

	opcode	operand
Type-B	11 bits	5 bits

Max (1 type A) = $1 + (2^6 - 1) * 2^5$
Min (1 type B) = $(2^6 - 1) + 2^5$

Input	0X DE AD BE EF
Big-Endian	0: DE, 1: AD ...
Little-Endian	0: EF, 1: BE ...