CS 152
Computer Systems Architecture
Summer Session II 2018

Homework 1

Due August 14th, 2018, EEE 11:59PM in PDF form

Name	Student ID
[redacted]	

IMPORTANT NOTES:

1. **No late submissions.**
2. **Please show your work. Remember that bottom line answers without proper explanations are worth ZERO points.**
3. **Even if you consult with your peers, do make sure the answers herein are yours and that you do not copy or cut-and-paste from other people's work.**

For Grading Purposes Only:

Q1	Q2	Q3	Q4	Q5	Q6
10	10	10	10	10	10

Q7	Q8	Q9	Q10	BONUS
10	10	10	10	10

Total Score
100

Homework #1

① The Von Neumann cycle is illustrated below:

[→IF→ID→EA→OF→EX]

The basic function of each of the states are listed below:

- IF (instruction fetch): The control unit gets an instruction from memory
- ID (instruction decode): Encoded instruction obtained from the IF state (that of which was a binary representation of the same number) was decoded, or translated, from the instruction (said number) to a control word (bit string with any deterministic amount of bits that tells resources what to do in the execution of the instruction) that told all the resources in the CPU what to do, with the same logic gates, to carry out the execution of the instruction
- EA (effective address): This state depends on hardware to compute, in the linear addressable memory, where an operand would be found
- OF (operand fetch): We use the effective address to fetch an operand, & perform the function associated with the mnemonic, for the instruction obtained

From the Von Neumann model, we define two constraints on how we can define instructions for the machine:

(i) Given modern, wide memories, we have enough bits such that we can afford to store fully decoded instructions, or control words, in memory (which effectively eliminates the ID state)

(ii) Instead of fetching an operand from memory, & executing & putting results in memory, all arithmetic instructions will work with registers (only loads & stores will require memory access).

These constraints leaves us with a fetch-execute cycle for a RISC cycle derived from a Von Neumann cycle.

---

(2) • Starting with the J-instruction format, that of which only has one type, the unconditional jump, the 26-bit offset, given by the instruction format, gives us a +/- displacement from the current program counter to find the destination address. The 26-bit offset is logically shifted to the left by 2 bits for byte addressability, which is given by the ISA. Then, this shifted 26-bit offset is sign extended to 32-bits (because it is a 2's complement quantity) & added to the current PC to give the destination address. Thus, the range to jump is $\pm 2^{28}$ bits from the current PC. This can be expressed in RTL notation as

$$PC \Leftarrow PC + (sign\_extend[26\ bit\ offset << 2])$$

• For the I-instruction format, the first type is given by immediate type. The format gives us a 16 bit immediate value, & two 5 bit values that are defined by the RTL. Thus, we load the whole 32 bit value in two instructions, in a "high/low" fashion (high being the most significant 16 bits & low being the least significant 16 bits) into the rt register. The opcode will determine if it is a "high" or "low" instruction.

• For the second I-type, memory access, we are obviously accessing memory here (with loads & stores). In RTL notation, rt can be expressed as the destination register for loads or the source register for stores, & rs will be the the base address for the memory. Thus, the 16bit offset will be a +/- displacement, from the base address, to the sought after data item. The 16 bit offset is sign extended, such that it is a 32 bit value that

can be added to the base address, but it is not logically shifted to the left by 2 because we do not want instructions, but we want data (we want to be able to address the byte within 32 bits). In both loads & stores, the memory address is given by the 32 bit base address (the source register) plus the sign extended 16 bit offset. In RTL, this is notated as

$rt \Leftarrow M[rs + sign\_extend(16\ bit\ offset)]$ (loads)

$M[rs + sign\_extend(16\ bit\ offset)] \Leftarrow rt$ (stores)

- For the third (& last) I-type conditional branching, a test is initiated between the registers rs & rt (as denoted by the RTL notation), where a subtraction (rs-rt) is performed & whether the result is positive, negative, or zero, the program counter will be updated to the result of the program plus a sign extended 16 bit offset. (as defined by the RTL) logically shifted to the left by 2 (since we are jumping to another instruction). This gives this I-type a potential jumping range of $\pm 2^{17}$. In RTL, this is notated as

if (rs-rt) branch to PC + (sign extend[16 b offset << 2])

- Lastly, for the R-instruction format, there is only one type, which is the R format. Simply, as defined by the RTL for RISC, arithmetic operations can only be performed with registers, so the specified arithmetic operation is performed between the first & second source registers (per the RTL), & written to the destination register. Additionally, a shift amount may be specified in bits 6-10 for logical shift instructions. In RTL notation, this is expressed as

$rd \Leftarrow rs \langle func \rangle rt$

* Note that for all types, the fetch stage is merely obtaining the instruction to be executed (as the instructions are stored in memory).

(3) The case where there is an available register, that may be destroyed, is considered below:

```
; Assume temporary register $t0 to be the available register
; that may be destroyed
add $t0, $rt, $zero ; put the true value of rt into t0
                    ; (the value is added to 0, so
                    ;  it is written as the initial value)
add $rt, $rs, $zero ; put the true value of rs into rt
add $rs, $t0, $zero ; put the true value of rt into rs
```

Now, for the case that there are no available registers:

```
xor $rt, $rt, $rs
xor $rs, $rs, $rt
xor $rt, $rt, $rs
```

For a simple example of the latter case, suppose rt had a value of 0101 & rs had a value of 1010. Thus, the instructions would execute in this fashion, sequentially:

① rt = 1111 = 0101 ⊕ 1010
② rs = 0101 = 1010 ⊕ 1111
③ rt = 1010 = 0101 ⊕ 1111 (& thus the values are swapped WLOG)

To implement the swap function in hardware, suppose there is a program with x instructions & k cycles. Of the x instructions, k are swaps, & each swap is 3 cycles (as demonstrated above). Thus, there are two cases:

① $(x-k)+3k = x+2k$
② $0.1x + x = 1.1x$ (where the clock period of each instruction is increased by 10%)

Thus, the relation

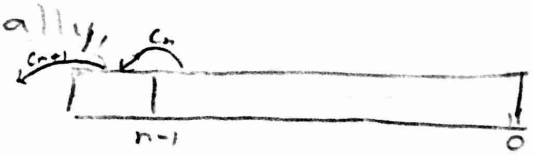
$$x+2k = 1.1x \Rightarrow 2k = 0.1x \Rightarrow k = 0.05x$$

indicates that at least 5% of x instructions being swapped in a given program justifies the hardware implementation of the function.

(4) a) We will prove 2's complement addition overflow (denoted as OVF below) with an exhaustive proof. Succinctly put, overflow occurs when addition of any two n-bit numbers of the same sign ends up in a result of the same sign. Overflow can be detected by the XOR of the carry in to the most significant bit & the carry out of the most significant bit. This can be denoted as

$$OVF = C_n \oplus C_{n+1}$$

or visually,



The independent variables in our proof will be C_n & the signs of x & y, S_x & S_y. These are the independent variables to be considered when adding two two's complement numbers. The dependent variables include C_{n+1}, S_{x-y}, & OVF. Thus, the truth table is shown below

C_n	S_x	S_y	C_{n+1}	$C_n \oplus C_{n+1}$	S_{x-y}	OVF=
0	0	0	0	0	0	0
0	0	1	0	0	1	0
0	1	0	0	0	1	0
0	1	1	1	1	0	1
1	0	1	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	0
1	1	0	1	0	1	0

By the tautology $OVF \equiv C_n \oplus C_{n+1}$ shown in the truth table above, addition overflow between two n-bit two's complement numbers, addition overflow occurs iff the carry into the most significant bit position & the carry out of the most significant bit position are different. Q.E.D.

b) Overflow in sign magnitude representation can be defined as a sign change if the two integers being added are both positive, or if the result is negative & there is overflow when the two numbers being added are both negative (overflow will never occur when the signs are different). Overflow detection in signed-magnitude representation, simply, is detecting a carry-out of the most significant bit of the magnitude. Note that overflow will never happen when subtracting two magnitudes, so there is no way to detect it.

(5) $x = 01010101_2 = 85_{10}$
$y = 11101111_2 = -17_{10}$

a) $x + y =$

$$\begin{array}{rll} 1\,1\,1\,1\,1\,1\,1 & & \\ 0\,1\,0\,1\,0\,1\,0\,1 & (85_{10}) & \\ +\,1\,1\,1\,0\,1\,1\,1\,1 & (-17_{10}) & \\ \hline 0\,1\,0\,0\,0\,1\,0\,0 & (68_{10}) & \end{array}$$

b) $x - y = x + (-y)$, where $-y$ is

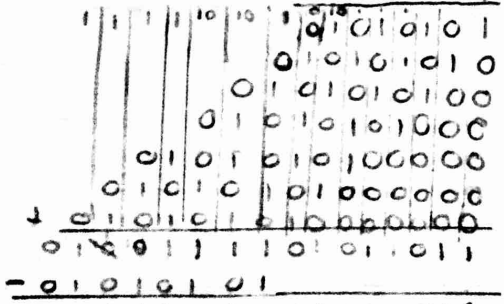
$$\begin{array}{l} 1\,1\,1\,0\,1\,1\,1\,1 \\ \hline 0\,0\,0\,1\,0\,0\,0\,0 \\ \qquad\qquad\quad 1 \\ \hline 0\,0\,0\,1\,0\,0\,0\,1 \end{array}$$

Thus, $x + (-y)$ is

$$\begin{array}{rll} 0\,1\,0\,1\,0\,1\,0\,1 & (85_{10}) & \\ +\,0\,0\,0\,1\,0\,0\,0\,1 & (17_{10}) & \\ \hline 0\,1\,1\,0\,0\,1\,1\,0 & (102_{10}) & \end{array}$$

c) $x \cdot y =$

$$\begin{array}{rll} 0\,1\,0\,1\,0\,1\,0\,1 & (85_{10}) & \\ \times\,1\,1\,1\,0\,1\,1\,1\,1 & (-17_{10}) & \end{array}$$



$$\begin{array}{rl} 1\,1\,1\,1\,1\,0\,1\,0\,0\,1\,0\,1\,1\,0\,1\,1 & (-1445_{10}) \end{array}$$

d) $x \div y = (85_{10}) \div (-17_{10}) = -5_{10}$

Now, to turn -5_{10} to a base -2 integer, we will integer divide & take the two's complement of the raw binary integer.

x	x // 2
5	1
2	0
1	1

Now, taking the two's complement of the unsigned binary integer 101_2:

$$\begin{array}{r} 0101 \\ 1010 \\ +\quad\ 1 \\ \hline 1011 \end{array}$$

Thus, $x \div y = 1011_2$

⑥ a) With the given bit pattern

1010 1111 1010 1000 0000 0000 0000 1000

this equates to a base-10 integer of
$-(2^{31}) \cdot 2^{29} + 2^{27} + 2^{26} + 2^{25} \cdot 2^{24} \cdot 2^{23} + 2^{21} + 2^{19} \cdot 2^3$
$= -1347944440_{10}$
assuming this represents a two's complement integer.

b) Assuming that the given bit pattern represents an unsigned integer, this equates to a base-10 integer of $2^{31} + 2^{29} + 2^{27} + 2^{26} \cdot 2^{25} + 2^{24} \cdot 2^{23} + 2^{21} \cdot 2^{19} \cdot 2^3$
$= 2947022856_{10}$.

c) Assuming that the given bit pattern represents a single precision IEEE standard floating point number, we will use the definition of a single-precision IEEE floating point representation for some base-2 integer x, which is

$$x = (-1)^{\mathcal{S}}(1 + \mathcal{F}) \cdot 2^{(\mathcal{E}-\mathcal{B})}$$

where $\mathcal{S}$ represents the sign bit (the 31st bit in a given bitstring), $\mathcal{F}$ represents the fraction (of the bits 0-22), $\mathcal{E}$ represents the exponent (of the bits 23-30), and $\mathcal{B}$ represents the bias (for single-point precision, this is 127). Thus,

$\mathcal{S}$ $\mathcal{E}$ $\mathcal{F}$
1010 1111 1010 1000 0000 0000 0000 1000 = x

$\mathcal{S} = 1$, $\mathcal{E} = 01011111_2 = 95_{10}$, & $\mathcal{F} = 2^{-2} + 2^{-4} + 2^{-20} \approx 0.3125$

$\therefore x = (-1)^1(1 + 0.3125) \cdot 2^{(95-127)} \doteq -1.3125 \cdot 2^{-32}$
$\doteq -3.055902 \cdot 10^{-10}$
Q.E.D.

d) Assuming that the given bit pattern represents a MIPS instruction, the bit pattern can be interpreted as follows:

opcode rs rt immediate

1010 1111 1010 1000/0000 0000 0000 1000

The instruction is an I-type memory access instruction (as indicated by opcode 101011), where the word in register 29 is being stored at the memory address in register 8, offset by 8. In RTL notation, this can be expressed as

$$M[r8 + \text{sign-extend}(8)] \Leftarrow r29$$

(7) a) To convert 752_9 to a base-3 representation, we will first take the base-10 "intermediary" representation of 752_9, and convert this result to base-3 by integer division. As such,

$$752_9 = 7 \cdot 9^2 + 5 \cdot 9^1 + 2 \cdot 9^0 = 614_{10}$$

Now, for the integer division.

x	x //3
614	2
204	0
68	2
22	1
7	1
2	2

Thus, $752_9 = 211202_3$

b) Representing $-\frac{1}{3}$ in IEEE 754 floating point representation, we will first convert $-\frac{1}{3}$ to binary representation using integer multiplication, as follows

Product	Result	Binary Digit
$0.\overline{3} \cdot 2$	$0.\overline{6}$	0
$0.\overline{6} \cdot 2$	$1.\overline{3}$	1
$0.\overline{3} \cdot 2$	$0.\overline{6}$	0
$0.\overline{6} \cdot 2$	$1.\overline{3}$	1
$0.\overline{3} \cdot 2$	$0.\overline{6}$	0
$0.\overline{6} \cdot 2$	$1.\overline{3}$	1
⋮	⋮	⋮

So, $(\frac{1}{3})_{10}$ can be represented as $0.\overline{01}_2$, & this can be expressed in scientific notation as $1.\overline{01} \times 2^{-2}$. From this, we may deduce that the sign bit S can be expressed as 1 (to denote the negative sign). To get the mantissa $\mathcal{F}$, we simply take 23 digits of $\overline{01}$ for single point precision & 52 digits for double point precision. For the exponent value $\mathcal{E}$, we get $127 - 2 = 125_{10} = 01111101_2$ for single point precision, & $1023 - 2 = 1021_{10} = 01111111101_2$ for double point precision. Altogether, we get the following representations for $-\frac{1}{3}$:

Single point precision:
1011 1110 1010 1010 1010 1010 1010 1010

Double point precision:
1011 1111 1101 1010 1010 1010 1010 1010 1010 1010
1010 1010 1010 1010 1010 1010

(8) a) For the execution rate in terms of MIPS, we can simply use the ratio

$$\text{Execution rate} = \frac{\text{Instructions executed}}{\text{Execution time}}$$

$$A_{\text{execution rate}} = \frac{5.4 \times 10^6}{5} = 1.08 \times 10^8 = 108000000 \text{ MIPS}$$

$$B_{\text{execution rate}} = \frac{115 \cdot 10^6}{6.4} = 17.90875 \times 10^6 = 17906750 \text{ MIPS}$$

b) Using the relation

$$\text{Clock cycle} = \frac{\left(\frac{\text{Execution time}}{\text{Instructions executed}}\right)}{\text{CPI}} = \frac{\text{Execution time}}{(\text{Instructions executed})(\text{CPI})}$$

we get

$$A_{\text{clock cycle}} = \frac{5}{(1.8)(5.4 \times 10^8)} \doteq 0.5144 \times 10^{-8} \text{ s} \doteq 5.144 \text{ ns}$$

$$B_{\text{clock cycle}} = \frac{6.4}{(2.0)(115 \times 10^6)} \doteq 0.0278 \times 10^{-6} \text{ s} \doteq 0.0278 \ \mu\text{s}$$

c) Using the definition of measuring performance,

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{6.4 \text{ s}}{5 \text{ s}} = 1.28$$

Thus, we find that A is 1.28 times faster than B.

(9) For an instruction count i, a clock cycle time t, & a cycles per instruction count of C, this program will assume a CPU time of $i \cdot c \cdot t$. Thus, for 15% of the instructions (those of which are multiplication), the CPU time will be

$$\text{CPU time}_{15} = (0.15i)(12)(t) = 1.8i \cdot t$$

where a multiply instruction takes 12 cycles. As for the other 85% of instructions, the CPU time will be

$$\text{CPU time}_{85} = (0.85i)(4)(t) = 3.4i \cdot t$$

Thus, the total CPU time is simply the sum of the CPU times of 85% & 15% of the instructions, respectively. Thus,

$$\text{CPU time}_{100} = \text{CPU time}_{85} + \text{CPU time}_{15} = 1.8it + 3.4it$$

$$= 5.2i \cdot t$$

Without loss of generality, we just need to find the total time that the CPU spends doing multiplication with respect to the total CPU time over all instructions, so

$$\frac{\text{CPU time}_{15}}{\text{CPU time}_{100}} = \frac{1.8it}{5.2it} \doteq 0.34615$$

or 34.615% is the percentage of time that the CPU spends doing multiplication.

(10) a) Given that there is only one adder & one multiplier, all the multiplication operations must be done one at a time first before the two terms $(a*b*c*d)$ and $(a*e)$ may be added, since nothing here can potentially be pipelined (this would result in a data hazard). Thus, we have a total of 4 multiplication operations (each performed in 9ns) & 1 addition operation (performed at 1ns), so the best case would be

$$C(x) = 4(9ns) + 1(1ns) = 36ns + 1ns = 37ns$$

b) The optimization of x would be

$$a*(b*c*d + e)$$

where the best case running time, with 3 multiplication operations and one addition operation, would be

$$C(x) = 3(9ns) + 1(1ns) = 27ns + 1ns = 28ns$$

If there were two multipliers, the best case for the optimization remains the same because nothing can be pipelined or ran in parallel — there are data dependencies in the product of $b*c*d$, and the product of a & the rest of the terms in the parentheses. Though, for the original unoptimized equation $(x = a*b*c*d + a*e)$, $a*b$ & $c*d$ can be pipelined, as well as resulting product of $(a*b)$ & $(c*d)$, and $a*e$. This would result in a runtime of

$$C(x) = 2(9ns) + 1(1ns) = 19ns$$

Bonus: The pseudocode for the addition of two natural numbers is as follows:

```
/* Helper function to add two natural numbers.
 *
 * @param  x: The first natural number
 * @param  y: The second natural number
 * @return: A sum of the two natural numbers, as an integer
 */

public int addition(int x, int y)
{
        //Increment x by 1 until y reaches 0
        //(thus, we add the true value of y to
        // x). If we do not receive a natural number
        // for y, then y will never enter the loop
        // & simply return the passed in value for
        // x.
        while (y > 0)
        {
                x = x + 1;
                y = y - 1;
        }
        //Return the sum contained in x
        return x;
}
```