CS 152
Computer Systems Architecture
Summer Session II 2018

Homework 2

Due August 21st, 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
10	10	15	15

Total Score
110

① Given the two 32 bit floating point numbers

$X = 0000\ 0000$

$Y = 813F\ FFFF$

we can convert X & Y from a hexadecimal representation to a binary representation as follows

$X = \overset{s}{0}\ \overset{e}{000\ 0000}\ \overset{f}{1\ 00\ 0000\ 0000\ 0000 \cdots 0}$

$Y = \overset{s}{1}\ \overset{e}{000\ 0001}\ \overset{f}{0\ 011\ 1111\ 1111\ 1111\ 1111\ 1111}$

Given to us are the presumptions that the sign bit is the most significant bit, the 8-bit exponent is unbiased & in 2's complement representation, & the 23-bit mantissa is also in 2's complement representation & there is no implied 1-bit. Thus, this gives us

$$X = f_x \cdot 2^{e_x}$$
$$Y = f_y \cdot 2^{e_y}$$

where

$$f_x = 100\cdots0_2$$
$$f_y = 011\cdots1_2$$
$$e_x = 00000001_2$$
$$e_y = 00000010_2$$

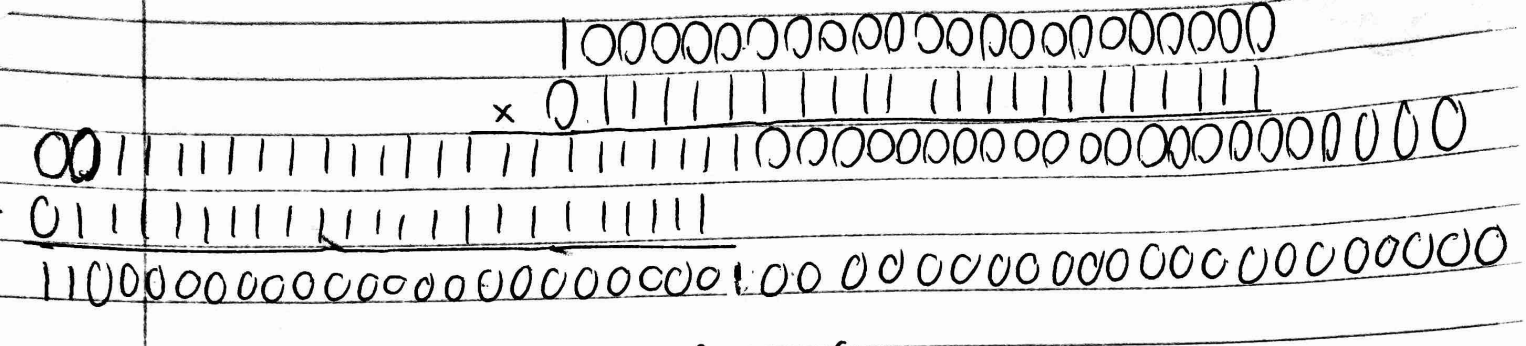
Thus,

(a) $X * Y = (f_x \cdot 2^{e_x}) \cdot (f_y \cdot 2^{e_y}) = 2^{e_x + e_y}(f_x \cdot f_y)$

where $e_x + e_y$ is

$00000001_2 + 00000010_2 = 00000011_2$

and $f_x \cdot f_y$ is



Thus, $x * y = 2^{e_x + e_y}(f_x \cdot f_y)$

$= (010^{00000011} \cdot \overbrace{110\cdots0}^{46b})_2 \doteq 010^{011} \cdot 0.11$

$\doteq 0110.0_2$

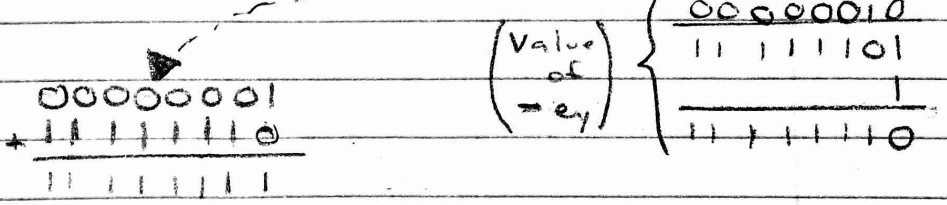
QED

(b) $x + y = 2^{e_x} f_x + 2^{e_y} f_y$

$= 2^{e_y} f_y + 2^{e_x} f_x$

$= 2^{e_y}[f_y + f_x 2^{e_x - e_y}]$ (where $e_y \geq e_x$)

where $e_x - e_y$ (or $e_x + (-e_y)$) is,



Finding $2^{e_y}[f_y + f_x 2^{e_x - e_y}]$,

$x + y = 010^{010}[0.01 + (0.1 \cdot 010^{11111111})]$

$= 0100[0.01 + (0.1 \cdot 0.1)]$

$= 0100[0.01 + 0.01]$

$= 010 + 01 = 011$ (but since the sign bit is 1, the answer is 101) QED

② Given that the clock rate for M1 is 1 GHz and for M2 is 800 MHz, we see that

$$M1_{\text{clock cycle time}} = 10^{-9}\,s = 1\,ns$$

$$M2_{\text{clock cycle time}} = \frac{1}{800 \times 10^6}\,s = 0.00125 \times 10^{-6}\,s$$

$$= 1.25 \times 10^{-3}\,s = 1.25\,ms$$

Now, for x instructions in program P, we find the total amount of cycles for each machine is as follows

$$M1_{\text{clock cycles}} = 5(0.25x) + 7(0.1x) + 19(0.1x) + 2(.55x)$$
$$= 1.25x + 0.7x + 1.9x + 1.1x$$
$$= 4.95x$$

$$M2_{\text{clock cycle}} = 15(0.25x) + 30(0.1x) + 50(0.1x) + 2(.55x)$$
$$= 3.75x + 3x + 5x + 1.1x$$
$$= 12.85x$$

Thus, to obtain the total execution time for each machine

$$M1_{\text{execution time}} = (4.95x) \cdot (10^{-9}\,s)$$
$$= 0.00000000495(x)\,s$$

$$M2_{\text{execution time}} = (12.85x) \cdot (1.25 \times 10^{-3}\,s)$$
$$= (16.0625x) \cdot (10^{-3}\,s)$$
$$= 0.0160625(x)\,s$$

Using the performance ratio

$$\frac{\text{Performance}_{M1}}{\text{Performance}_{M2}} = \frac{\text{Execution time}_{M2}}{\text{Execution time}_{M1}} = \frac{0.0160625(x)\,s}{0.00000000495(x)\,s}$$
$$= 3.2449 \times 10^6$$

Thus, M1 is faster in executing program P by a factor of 3.2449×10^6

③ SINGLE CYCLE IMPLEMENTATION OF A MIPS PROCESSOR

(i) R-Type

The R-Type instruction performs logical or arithmetic operations between two registers. The R-Type is strictly one type of operation only. Starting with the fetch cycle, as in all instruction types, the rising edge of the clock indicates that the program counter has been updated (the value contained in the program counter could be PC+4 or a destination address). The program counter points to a 32 bit instruction in memory (in this case, an R-type instruction). Succinctly, the register file will read in two register operands, & write the results back into a destination register as instructed by the appropriate control lines. Thus, registers rs & rt will be read registers 1 & 2, & rd will assume the role of the write register. The ALU will compute the result of the data in operands rs & rt, & the result is presented back as write data to be written to rd. Thus, data will be written into the write register at the rising edge of the clock (as state changes only happen at this time).

(RTL: rd ⇐ rs <func> rt)

(ii) I-Type Memory Access (Store)

There is a rising edge of the clock that indicates that one instruction has finished & we are starting a new instruction (thus, the PC is updated). Since this is a store instruction, registers will not be written to, as "Write register" in the register file is not active; though, in Data Memory, "Write data" is active & corresponds to register rt. Thus, we add the sign extended 16-bit offset to the contents of rs to get the desired memory location, where we will store the contents of rt. The multiplexer before the ALU chooses the sign extended 16 bit offset instead of rt such that the 16 bit offset can be added to the operand rs (this becomes the sought after memory address), & rt corresponds to write data in the data memory. Thus, at the rising edge of the clock, the data in rt will be stored in the computed memory address (rs + 16 bit offset). (RTL: M[rs + sign_extend(16b offset)] ⇐ rt)

(iii) I-Type Memory Access (Load)

This instruction uses the memory address of rs plus the 16 bit offset. Obviously, at the rising edge of the clock, we fetch the instruction. Because it is a load, "Write data" in data memory will be inactive but the "Read data" in data memory will be active. Both rs & rt operands are read, & data that comes from rt will be presented as write data to the memory (but we shall ignore this data). rs & the

sign extended offset are added by the ALU to give the address in memory from which we read 32 bits to put into rt (the write register). The multiplexer after the data memory determines which 32 bit data item is written back into the write register of the register file (it can be the data read from memory or the value from the ALU as a result of formulating rs & rt and producing rd). Controls will decode the opcode & decide on whether the ALU result gets returned to the register file when the instruction is an R-Type, or whether the read data gets returned to the register file when the instruction is a load (the latter is true in this case). Thus, the final data value is written to rt at the rising edge of the clock.

(RTL: rt ⇐ M[rs + sign_extend(16-bit offset)])

(iv) J-Type

At the rising edge of the clock, we fetch the instruction & the program counter is updated to point at the instruction (indicative of an unconditional branch, or jump instruction). All the control lines that go to the data memory do nothing, & we don't use the register file at all. All that is done is the program counter is added with 4, and that sum is added with the

sign-extended 26 bit offset shifted to the left by 2, to obtain a sum that is fed to the input 1 of a multiplexer (the control line, from the control unit, uses combinational logic to tell the multiplexer to choose 1). The program counter then points at this instruction to be the next instruction during the Instruction Fetch (IF) stage.

(RTL: PC <= PC + (sign_extend[(26 bit offset) << 2]))

(v) I-Type Conditional Branch

At the rising edge of the clock, we will fetch an an instruction, that of which is a branch. Succinctly, we compute the destination address based on the condition (rs-rt). With this condition, we are interested in whether the result is 0 or less than 0. The instruction tells us which registers are the two registers, rs & rt, & will tell the ALU to do the subtraction. The "Zero" valve computed by the ALU (either true or false) will decicle if the register destination branch should be taken or not, & this is dependent on the condition. Thus, the destination address is the program counter summed with 4 & the 16 bit offset shifted by two if the branch is to be taken. If this branch is to be taken, the multiplexor connected to the combinatorial logic (by a control line) will

choose this destination as the next program counter, & if this branch is not to be taken, this multiplexor will choose the sum of the program counter & 4 as the next program counter & continue to the execution of the next instruction.

(RTL: if (rs - rt) branch to PC + (sign_extend[(16-bit offset) << 2]))

MULTI-CYCLE IMPLEMENTATION OF A MIPS PROCESSOR

Note that for both the Instruction Fetch (IF) & Instruction Decode/Register Fetch (ID/RF) stages, the process is the same across all 5 instruction types. For the IF stage, we fetch an instruction with the program counter & put it in the instruction register, & then increment the program counter by 4 to be put back into the program counter. In RTL, this can be denoted in RTL as

```
IR = Memory[PC];
PC = PC + 4;
```

Next, for the ID/RF stage, we will have the control tell us what the instruction is by setting the control lines that tell the resources what to do. Furthermore, for all instructions except for the jump, we need to read two registers from the register file. We free resources by putting the values of the two registers into intermediary registers A & B.

All instructions do the same thing in this stage - we don't set control lines based on instruction (it is decoded in the control logic). Thus, the RTL for this stage is as follows:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign_extend(IR[15-0])<< 2);
```

The next three stages deviate from each other based on the instruction type:

(i) R-Type

In the execution stage (EX) the two registers are stored in registers A & B outside of the register file, so A & B are the main operands used (where A & B are interpreted as rs & rt). Thus, this can be illustrated as the operation between the two operands, or in RTL,

```
ALUOut = A op B;
```

In the R-Type 4th stage, we simply write the result of the operation of the two operands (which is now stored in an intermediate ALUOut register to free the ALU resource for other computations) back to the destination register rd. In RTL, this is

```
Reg[IR[15-11]] = ALUOut;
```

(ii) I-Type Memory Access (Loads & Stores)

In the EX stage, given the instruction & the register operands, we can do a memory access computation by using the ALU to compute the address (base address plus

the offset given by the instruction in the IR). This address is stored in the intermediary register ALUOut, so the ALU is free to do other computations. This stage, in RTL, is denoted as

ALUOut = A + sign_extend(IR[15-0]);

In the memory access (MEM) stage, we access memory since we have the address in ALUOut, where loads write data from memory to an intermediary memory data register, & stores write data to memory from the B operand (rt). In RTL, this is denoted as

MDR = Memory[ALUOut]; (Loads)
Memory[ALUOut] = B; (Stores)

In the final write back stage (WB), we write the value from memory (now stored in MDR) into the destination address. This can be denoted as

Reg[IR[20-16]] = MDR;

(iii) J-Type Unconditional Jump & I-Type Conditional Branch

For the J-Type, unconditionally, we do the address computation in the ALU in the ID/RF stage (the program counter plus the offset), & now, in the EX stage, all we need to do is make the program counter equal to the value of this address. For conditional branches in the same stage, we perform the computation A - B in the ALU, & if this stands true, then we make the program counter equal to the value of this address. In RTL, this is denoted as

if (A == B) then PC = ALUOut; (I-Type)
PC = ALUOut; (J-Type)

Referring to the two figures, figure 1 is the implementation of a single cycle processor because one instruction is executed per clock cycle. Additionally, note that the I-type immediate is not implemented, & therefore, the explanation has been omitted.

(4)

Instruction Class	Instruction memory	Register Read	ALU/ Adder	Data Memory	Register write	Total
Register format	300 ps	200 ps	100 ps			600 ps
Conditional Branch	300 ps	200 ps	100 ps			600 ps
Memory Load	300 ps	200 ps	100 ps	300 ps		900 ps
Memory Store	300 ps	200 ps	100 ps			600 ps
Unconditional Jump	300 ps					300 ps

Note here that state changes only happen at the rising edge of the clock, so we are only concerned about a single clock cycle from rising edge to rising edge (in other words, the latency resulting from the state changes of a memory load (register write), memory store (data memory), & register-format (register write) are omitted). Thus, within the bands of the rising edges (non-inclusive), the critical paths may be defined as follows:

- The critical path for register format includes instruction memory, register read, & the ALU operation
- The critical path for conditional branching includes instruction memory, register read, & the adder operation
- The critical path for memory loads include instruction memory, register read, the ALU operation, & data memory
- The critical path for memory stores include instruction memory, register read, & the ALU operation
- The critical path for unconditional jumps include instruction memory

(5) (a) Given that every instruction completes in one clock cycle, & the clock cycle is the same for all instructions, we only need to calculate the slowest instruction, as this will determine the clock cycle length. Without loss of generality (see problem (4)), the slowest instruction is a memory load, so the clock cycle length in this case is 900 ps.

(b) Without loss of generality (see problem (4)), given the percentages for the instruction mix, the average clock cycle length is as follows

$$0.25(900) + 0.05(600) + 0.6(600) + 0.06(600) + 0.04(300) = 663$$

or 663 ps for a variable clock cycle.

(c) The performance ratio of a variable clock V to a single clock S, is simply

$$\frac{\text{Performance}_V}{\text{Performance}_S} = \frac{\text{Execution time}_S}{\text{Execution time}_V} = \frac{900\text{ps}}{663\text{ps}} \doteq 1.357$$

Thus, the variable clock is approximately 1.357 times faster than the single clock.

(6) Assuming a multi-cycle architecture & the same instruction mix (for x instructions),

a) The number of clock cycles it will take to execute any one of the instructions is as follows:
→ Load: 5 clock cycles
→ Store: 4 clock cycles
→ ALU operation: 4 clock cycles
→ Branch: 3 clock cycles
→ Jump: 3 clock cycles

b) Thus, the total amount of clock cycles for this mix of x instructions can be given by

$$0.25x(5) + 0.05x(4) + 0.6x(4) + 0.06x(3) + 0.04x(3)$$
$$= 1.25x + 0.2x + 2.4x + 0.18x + 0.12x$$
$$= 4.15x$$

or $4.15x$ total clock cycles. We will then divide this value by the total number of instructions (x) to obtain the CPI rating or

$$CPI = \frac{4.15x}{x} = 4.15 \text{ cycles per instruction}$$

② Note here that state changes only happen at the rising edge of the clock, so we are only concerned about a single clock cycle from rising edge to rising edge (in other words, the latency resulting from the state changes of a memory load (register write) is omitted). Thus, within the bounds of the rising edges (non-inclusive), we will find the clock cycle length that takes the longest, as this will be our fixed clock cycle length for the entire instruction mix for that specific instance. In the most general case, the load instruction will be the largest, with a clock cycle time of (memory fetch) + (register read) + (ALU) + (Data access), or 200ps + 100ps + 200ps + 300ps. = 800ps. Also note that PC+4 is done during the instruction fetch (IF) stage at the rising edge of the clock. In the general case, the conditional branch's execution time is (memory fetch) + (register read) + (ALU), or 200ps + 100ps + 200ps = 500ps, & the branch address computation is done in the ALU operation.

(a) If X = 300 & Y = 300,
Execution time of load =
(execution total time) − (memory fetch) + (adder time) = 800 − 200 + 300 = 900ps
Without loss of generality (as the branch execution time would be too small), the cycle time is 900ps in this instance.

(b) If X = 500 & Y = 500
Load execution time = 800 − 200 + 500 = 1100ps
Branch execution time = 500 − 200 + 500 = 800ps

WLOG, the cycle time is 1100ps in this instance.

(c) If X = 100 & Y = 800,
Load execution time = 800 - 200 + 100 = 700ps
Branch execution time = 500 - 200 + 800 = 1100ps
Here, the branch instruction has the longest cycle time, so the cycle time in this instance is 1100ps.

(8) For our single cycle implementation, we will assume the add-immediate instruction behaves as an R-Type instruction. The instructions used in the MIPS code snippet are the R-Type, Conditional Branch, Memory Load, & Unconditional Jump. Given that state changes only happen at the rising edge of the clock (non-inclusive), our times are as follows:

Instruction class	Instruction Memory	Register read	ALU/ Adder	Data Memory	Register write	Total
Register format	1.5 ns	1 ns	0.5 ns			3 ns
Conditional Branch	1.5 ns	1 ns	0.5 ns			3 ns
Memory Load	1.5 ns	1 ns	0.5 ns	1.5 ns		4.5 ns
Unconditional Jump	1.5 ns					1.5 ns

and given that the loop goes to the label 'Exit' on the 100th iteration after running completely for 99 times, we see that the execution time for each instruction during the duration of the program will look like the following:

```
Loop: addi $t0, $t0, 1      3ns  x 100 = 300 ns
      beq  $t0, $s1, Exit   3ns  x 100 = 300 ns
      add  $t0, $t0, $t0    3ns  x 99  = 297 ns
      add  $t0, $t0, $t0    3ns  x 99  = 297 ns
      add  $t1, $a0, $t0    3ns  x 99  = 297 ns
      lw   $t2, 0($t1)      4.5ns x 99 = 445.5 ns
      add  $s0, $t2, $s0    3ns  x 99  = 297 ns
      j    Loop             1.5ns x 99 = 148.5 ns
Exit:                                   2832 ns
```

The execution time for this program in the single cycle implementation is 2832 ns, or 2.832 μs.

For the multi-cycle implementation, we know that the number of clock cycles for the involved instructions are as follows:
- Register operation: 4 clock cycles
- Branch: 3 clock cycles
- Load: 5 clock cycles
- Jump: 3 clock cycles

Thus, the delays would be as follows for the involved instructions (in nanoseconds):

Instruction class	IF	ID/RF	EX	MEM	WB	Total
Register format	1.5	1	0.5	1.5		4.5
Conditional Branch	1.5	1	0.5			3
Memory Load	1.5	1	0.5	1.5	1	5.5
Unconditional jump	1.5	1	0.5			3

and given the same instructions & their respective assumptions, we get

Loop: addi ... 4.5 × 100 = 450 ns
beq ... 3 × 100 = 300 ns
add ... 4.5 × 99 = 445.5 ns
add ... 4.5 × 99 = 445.5 ns
add ... 4.5 × 99 = 445.5 ns
lw ... 5.5 × 99 = 544.5 ns
add ... 4.5 × 99 = 445.5 ns
j ... 3 × 99 = 297 ns
Exit: 3373.5 ns

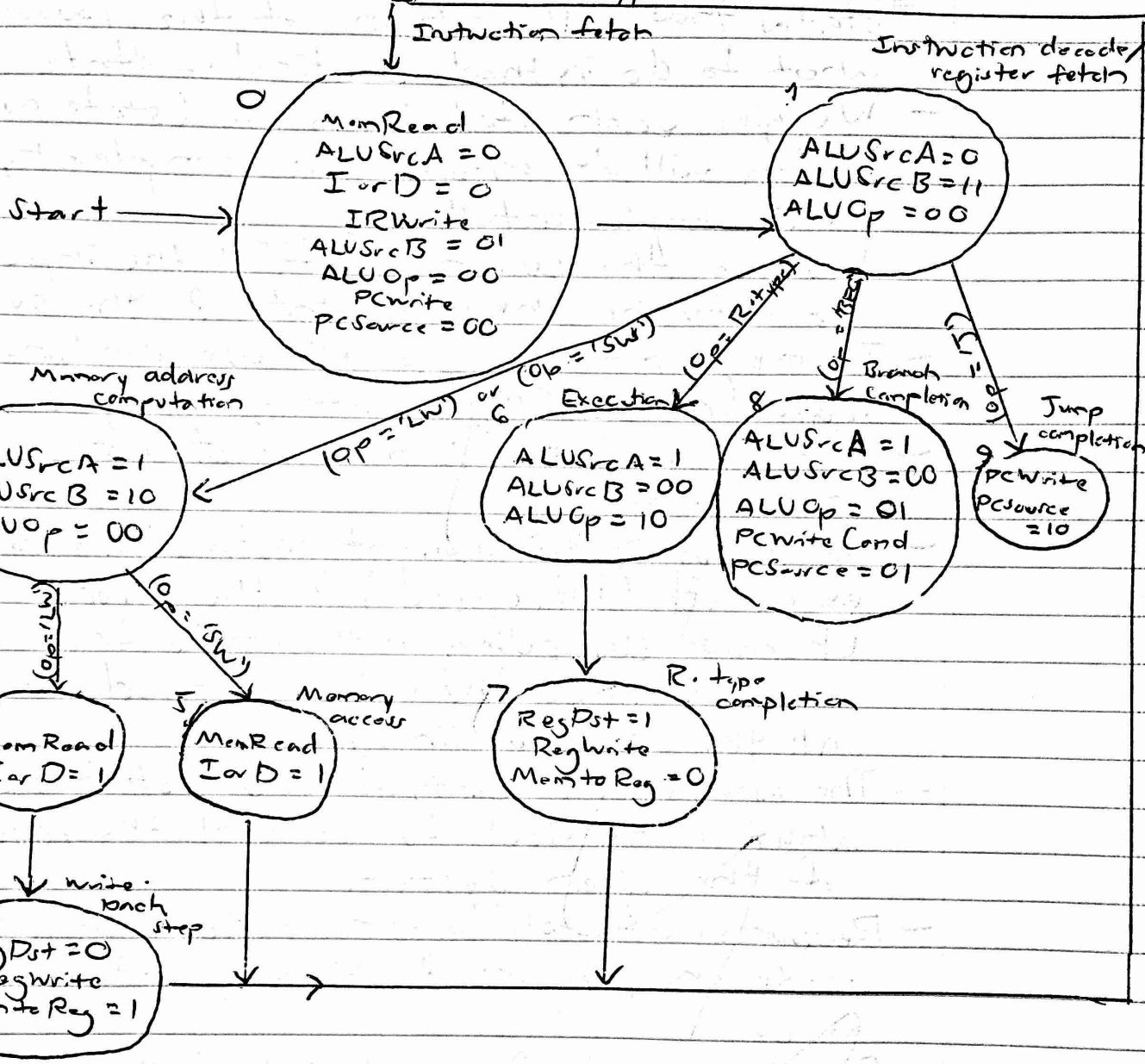
The execution time for the program in the multi cycle implementation is 3373.5 ns, or 3.3735 μs.

(9) a) We need 3 bits to address a microcode controller with 5 control words.

b) The maximum number of bits that are allowed in each control word are as many as we need in order to define the control words & the next state; this microcode controller is implementing the encoding of the opcode to tell the devices what to do at every clock cycle, & thus the maximum allowed is as many needed.

c) The same control function can indeed by performed by a combinational random circuit, because it corresponds to a truth table. From the truth table we may derive logic. (this was seen in the ROM implementation). Thus, without loss of generality, we can suggest a ROM implementation to design a combinational random circuit. For x bits for the opcode, & y bits for state, we have x+y address lines. This means we can address $2^{(x+y)}$ entries in the ROM, where the outputs are the bits of data that the address points to. For outputs, there are z datapath control outputs, & y bits for state, which gives us z+y outputs. Thus, the size of our ROM would be $2^{(x+y)}(z+y)$ bits.

(10) a) The state transition diagram of a sequential machine capable of sequencing the execution of each of the MIPS instruction types is sketched below.



b) A microprogrammed implementation for the sequential control of the multi-cycle implementation of the RISC datapath can be used to implement the sequential machine in (a) by the following:

- For the execution .. of certain paths in the state transition diagram, the state name was just an increment of the previous one
- Thus, we can implement this using a memory with a certain/predetermined amount of entries
- For the first two locations in the microprogrammable memory, we will put the IF & the ID states
- The next location will be a branch, & a jump following (they both only get one line because it is only one state)
- For control, we will implement logic that goes from address to address, depending on the opcode
- Flags to the right will indicate if the sought after instruction has been reached (this is labeled "go to fetch") (it will use this control word & go back to fetch)
- This gives us the control, which is the next state plus the control lines
- An R Type would need 4 states, or two lines
- A store instruction is two more states, or lines
- A load gets three lines, or states
- A "case statement" is performed amongst the entries
- The number of states in the state transition diagram correspond to the number of lines in the microprogram (a total of 11 lines)
- The instruction fetch is decoded into a bunch of ~~[illegible]~~ change of states that can have any number of states inbetween (these states are given by the control word that

says what is being done with the resources of the machine)

- This is illustrated below