The Simple Decimal Computer

CS 350: Computer Organization & Assembler Language Programmin

A. Why?

- A simple decimal example illustrates how von Neumann computers work without worrying about binary representation and more complicated instruction sets.

B. Outcomes

After this activity, you should be able to

- Describe the basic design of a von Neumman computer and discuss how it differs from other architectures.
- Describe the parts of the instruction cycle and what happens during them.
- Trace instruction execution for a simple computer.

C. Questions

Questions 1 – 7 refer to the Simple Decimal Computer from the notes. Feel free to use R9, R8, … as temporary registers if you need them. If you need temporary memory locations, use locations 99, 98, …. Unless otherwise specified, assume the code for each question starts at location 00.

1. Write code that takes the contents of memory location 90, doubles it, and stores it back in location 90.
2. Write some code that sets R0 ← R0 – R1. Use memory location 99 as temporary storage. To subtract R1, you'll need to take its negative: R0 ← R0 + NEG R1
3. (a) Write a branch instruction that branches to location 12 if memory location 95 is positive.

   (b) Write code that uses a branch instruction(s) to go to location 12 if M[95] is non-positive (i.e., ≤ 0). Assume your code starts at location 30. (To do a branch-on-not-positive to location 12, first write an unconditional branch

to location 12; then just before it, put a branch-on-positive that jumps over the unconditional branch.)

(c) Write code that goes to location 12 if M[95] ≥ 0. Assume your code starts at location 30. (Hint: M[95] ≥ 0 iff M[1+[95 ≥ 1.)

4. Say location 00 contains 1200. What happens if we execute the instruction at location 00?
5. Write code to implement the following loop; use BRP to jump to the top of the loop. (Version (a): To decrement R0, assume M[90] = 1–. Version (b): To decrement R0, use ADDM with 1–.)

```
R0 ← M[20]; {…; --R0; } while (R0 > 0);
```

6. Write code to implement the following loop; use BRP to exit the loop and BR to jump to the top of the loop. Let XX be the first location after the loop.

```
R0 ← M[20]; while (R0 ≤ 0) {…; ++R0; };
```

7. Write code to implement the following loop; use a BRP and BR to exit the loop and a second BR to jump to the top of the loop. Let XX be the first location after the loop. ~~Assume M[90] = 1–.~~

```
R0 ← M[20]; while (R0 > 0) {…; --R0; };
```


Solution

1. Set M[90] ← 2*M[90]: (Uses R9 as a temporary register.)

```
1990 ; R9 ← M[90]
3990 ; R9 ← R9 + M[90]
2990 ; M[90] ← R9
```

2. Set R0 ← R0 – R1 (uses location 99 as temporary storage; destroys R1):

```
4100 ; R1 ← – R1
2199 ; M[99] ← R1
3099 ; R0 ← R0 – (original value of) R1
```

3. (a) Go to location 12 if memory location 95 is positive:

```
1095 ; R0 ← M[95]
8095 ; go to 95 if M[0 < [95
```

(b) Go to location 12 if M[95] ≤ 0:

```
1995 :00 ; R9 ← M[95]
8903 :01 ; go to 03 if M[0 < [95
7912 :02 ; go to 12 if M[95] ≤ 0
03: (next instruction to execute, if M[0 < [95)
```

(c) Go to location 12 if M[95] ≥ 0:

```
1995 :00 ; R0 ← M[95]
6901 :01 ; R0 ← M[1+[95
7912 :02 ; go to 12 if M[0 < 1+[95 (if M[95] ≥ 0)
03: (next instruction to execute, if M[0 > [95)
```

4. Since M[00] = 1200, executing the instruction at location 00 means we execute 1200 as an instruction, so we loads R2 with the value of M[00], namely 1200.


```
    --R0;
  } while (R0 > 0) {
is
1020 :00  ; R0 ← M[20]
01: …
…
(Version a: … :  3090  ; R0 ← R–0M[90] = R1–0)
(Version b: … : 6001–  ; R0 ← R1–0)
… : 8001  ; continue loop if R0 > 0
…
1–   :90  ; a constant
```

6. The loop

```
R0 ← M[20];
while (R0 ≤ 0) {
   … ;
   ++R0;
} // assume instruction after loop is at XX
```

is

```
1020 :00   ; R0 ← M[20]
80 :01XX   ; Exit loop if R0 > 0
…
… : 6001   ; R++0
… : 7001   ; continue loop
XX: (first location after the loop)
```

7. The loop

```
R0 ← M[20];
while (R0 > 0) {
   … ;
   --R0;
} // assume instruction after loop is at XX
```

is


```
1020 :00  ; R0 ← M[20]
8001 :01  ; continue loop if R0 > 0
7030 :02  ; exit loop if R0 ≤ 0
03: …     ; (top of loop body)
… : …
… :6001–  ; R0 ← R1–0
… : 7002  ; continue loop
XX: (first location after the loop)
```


Von Neumann Computers

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- The von Neumann architecture is the one used by modern computers

B. Outcomes

After this activity, you should be able to

- Describe the basic design of a von Neumman computer and discuss how it differs from other architectures.
- Describe the parts of the instruction cycle and what happens during them.

C. Questions

1. (a) What are the three main parts of a von Neumann computer?

   (b) What makes the von Neumann architecture different from earlier computer architectures. (Hint: "Stored program" architecture.)

   (c) What are the parts of the CPU?
2. (a) What is the Memory Address Register?

   (b) What is the Memory Data Register?

   (c) What are the steps involved in reading memory?

   (d) What are the steps involved in writing memory?
3. What does the Program Counter point to? Why is it badly named?
4. When we look at how instructions execute, we find three basic kinds of instructions. What are they and how do they differ?
5. (a) What are the phases of the instruction cycle?

   (b) What are the steps of the Fetch Instruction phase?

   (c) During the Decode Instruction phase of the instruction cycle, where is the instruction being decoded?

Solution

1. (a) CPU, Memory, and I/O devices

   (b) Programs are stored as data in memory.

   (c) Control unit and (Arithmetical/Logical) Processing Unit
2. (a) The Memory Address Register (MAR) holds the location to read or write.

   (b) The Memory Data Register (MDR) either holds the value read from memory or the value to write to memory.

   (c) To read: Set MAR to the address to read; signal Read; Find the value in the MDR and copy it out to wherever.

   (g) To write: Set MAR to the address to read; Set MDR value to write; Signal Write.
3. The PC points to the next instruction to execute. It's badly named in that it doesn't actually count anything.
4. Calculation instructions use and create values; Data movement instructions move data to/from memory; Control instructions perform goto operations.
5. (a) Fetch Instruction, Decode Instruction, Evaluate Addresses, Fetch Operands, Execute Instruction, Store Results.

   (b) Read the instruction pointed to the program counter from memory into the instruction register; then increment the program counter.

   (c) During Decode Instruction, the instruction is in the instruction register.

   (d) During Evaluate Addresses we figure out where the operands are stored; this could be a register number or a memory address. This phase is skipped if there are no operands.

   (e) During Fetch Operands we actually retrieve the operand values from a register, from memory, or from part of the instruction register. This phase is skipped if there are no operands.

   (f) During Store Results, values calculated during Execute Instruction are moved to memory or CPU registers. This phase is skipped if there are no instructions

Sequential Logic Circuits & Finite State Machines

CS 350: Computer Organization & Assembler Language Programming

I. Why?

- Sequential logic units are circuits that combine calculation with a fixed amount of stored information.
- Finite state machines are a common model for simple programs over a fixed amount of memory.

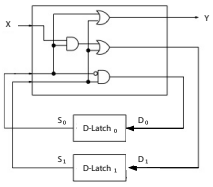
J. Objectives

At the end of this activity, you should:

- Be able to trace the execution of a simple sequential logic circuit.
- Be able to trace the execution of a finite state machine using its state transition table or state transition diagram.
- Be able to translate a state diagram for a finite state machine to an equivalent state transition table.

K. Questions

1. Refer to the modified Figure 3.43 from lecture:

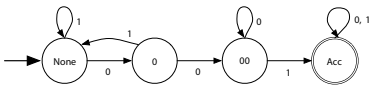


(a) Say $S_1 = 0$ and $X = S_0 = 1$. What are the sequence of Y, S_0, and S_1 values we get if we calculate Y, then D_0, then D_1, and then set S_0 and S_1 (and repeat) ?

(b) Repeat part (a) but assume $X = S_1 = 0$ and $S_0 = 1$.

(c) Repeat (part a) but assume we calculate Y then D_0, then set S_0, then calculate D_1, and then set S_1 (and repeat).

2. Here is a the state diagram for a finite state machine.



(a) Trace the execution of this machine on the input 0100010.

(b) What is the pattern of strings accepted by this machine?

(c) Complete the state transition table below for this machine.

0
00
00
Acc
Acc

3. We'd like a finite state machine that takes an input string of 0's and 1's and has two states E and O (for even and odd), which it uses to keep track of whether the input has an even or odd number of 0's. It accepts strings with even numbers of 0's. (a) Draw a state diagram for the machine. (b) Give a state transition table for the machine.

4. Fill in the following trace of execution table for the flashing warning sign FSM.

C_1 (State hi bit)	0	0						
C_0 (State low bit)	0	0						
S (On-Off Switch)	0	1	1	1	1	1	0	0
L_{12} (Lights 2,1)	0	0						
L_{34} (Lights 4,3)	0	0						
L_5 (Light 5)	0	0						
N_1 (new C_1)	0	0						
N_0 (new C_0)	0	1						

Solution

1a. In general $D_0 = \overline{S}_0 S_1$, $D_1 = X S_0 + S_1$, and $Y = S_0 + S_1$.
We start with $S_0 = 1$, $S_1 = 0$, and $X = 1$. Then

- $Y = S_0 + S_1 = 1 + 0 = 1$
- $D_0 = \overline{S}_0 S_1 = \overline{1}\,0 = 0$ and $D_1 = X S_0 + S_1 = 1 \;\; 1 + 1 = 0$
- $S_0 \leftarrow D_0 = 0$ and $S_1 \leftarrow D_1 = 1$

Repeating, we get

- $Y = S_0 + S_1 = 0 \;\; + 1 = 1$
- $D_0 = \overline{S}_0 S_1 = \overline{0}\,1 = 1$ and $D_1 = X S_0 + S_1 = 1\,0 + 1 = 1$
- $S_0 \leftarrow D_0 = 1$ and $S_1 \leftarrow D_1 = 1$

1b. This time we start with $S_0 = 1$, $S_1 = 0$, and $X = 0$. Then

- $Y = S_0 + S_1 = 1 + 0 = 1$
- $D_0 = \overline{S}_0 S_1 = \overline{1}\,0 = 0$ and $D_1 = X S_0 + S_1 = 0 \;\; 1 + 0 = 0$
- $S_0 \leftarrow D_0 = 0$ and $S_1 \leftarrow D_1 = 0$

Repeating, we get

- $Y = S_0 + S_1 = 0 \;\; + 0 = 0$
- $D_0 = \overline{S}_0 S_1 = \overline{0}\,0 = 0$ and $D_1 = X S_0 + S_1 = 0\,0 + 0 = 0$
- $S_0 \leftarrow D_0 = 0$ and $S_1 \leftarrow D_1 = 0$

1c. We again start with $S_0 = 1$, $S_1 = 0$, and $X = 1$ but change the order of computations.

- $Y = S_0 + S_1 = 1 + 0 = 0$
- $D_0 = \overline{S}_0 S_1 = \overline{1}\,0 = 1$ and then $S_0 \leftarrow D_0 = 0$
- $D_1 = X S_0 + S_1 = 0 \;\; 0 + 0 = 0$ and then $S_1 \leftarrow D_1 = 0$

Repeating, we get

- $Y = S_0 + S_1 = 0 \;\; + 0 = 0$
- $D_0 = \overline{S}_0 S_1 = \overline{0}\,0 = 0$ and then $S_0 \leftarrow D_0 = 0$
- $D_1 = X S_0 + S_1 = 1\,0 + 0 = 0$ and then $S_1 \leftarrow D_1 = 0$

Combinatorial Circuits

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Combinatorial logic circuits correspond to pure (state-free) calculations on booleans.

B. Outcomes

After this activity, you should be able to:

- Implement some standard combinatorial logic circuits

C. Questions

1. A half subtractor implements one column of a subtraction, assuming there was no borrow out to the column to its right. We have inputs X_i and Y_i and outputs D_i and B_{i+}; the borrow-in bit B_{i+} and difference D_i is the result of subtracting $X_i - Y_i$. (Hint: we only borrow to calculate 1 – 0.) Create a truth table for this operation with and give a full DNF representation for both D_i and B_{i+}.
2. A -4to2- multiplexer takes 4 bits of data input X [0:3] and uses 1 bit of selector input S to produce 2 bits of output Y [0:1]. If S = 0, then $Y_0 = X_{00}$ and $Y_1 = X_{01}$; if S = 1, then $Y_0 = X_{10}$ and $Y_1 = X_{11}$. Implement a -4to2- multiplexer using two -2to1- multiplexers.
3. Let X [0:2] be a -3bit unsigned number and let Y [0:2] be the -3bit result you get by incrementing X by 1, with variable W = 1 if over flow has occurred. (E.g. if X = 011 then W = 0 and Y = 100.) Write equations for outputs Y_2, Y_1, Y_0, and W from inputs X_2, X_1, and X_0; use logic gates to design a circuit for W, Y_2, Y_1, and Y_0.

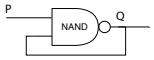
Solution

1. (Half subtractor $X_i - Y_i = D_i$ with borrow-in B_{i+}). We get $D_i = \overline{X}_i Y_i + X_i \overline{Y}_i$ and $B_{i+1} = \overline{X}_i Y_i$.

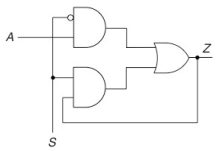
2. (Implement a -4to2- multiplexer with two -2to1- multiplexers): With one multiplexer, use S to select between X_{00} and X_{10} with output Y_0; with the other, use use S to select between X_{01} and X_{11} with output Y_1.

3. (Increment a -3bit unsigned number X [0:2] to get Y [0:2] and over flow W.) $W = X_2 X_1 X_0$, $Y_0 = \overline{X}_0$, $Y_1 = X_1 \overline{X}_0 + \overline{X}_1 X_0$, and $Y_2 = X_2 \overline{X}_1 + X_2 \overline{X}_0 + \overline{X}_2 X_1 X_0$.

3. Say an R-S latch powers up with a and b having random values and R = S = 0. What happens? Is this a problem? If so, suggest a fix.
4. Does this circuit have a stable (non-oscillating) logical value when P = 0? When P = 1? Does this circuit remember a bit?



5. (Exercise 3.27) (a) Describe the output of this logic circuit when the select line S is a logical 0. That is, what is the output Z for each value of A? (b) If the select line is switched from a logical 0 to 1, what will the output be? (c) Is this circuit a storage element?



6. Say we have a machine with k-bit addresses and m-byte addressability. (a) What is the address space and how large is it? (b) How many bytes of memory can we have in total? (c) What, if any, is the relationship between k and m?
7. Say a byte-addressable machine has -32bit memory addresses. How many gigabytes of memory can we access? What if memory addresses are -64bits? Hint: The standard prefixes are kilo-, mega-, giga-, tera-, peta-, exa-, zetta-, and yotta-.

Solution

1. Here's a full truth table for an R-S latch; the rows we wanted are checked off.

	R S	a b	New a = $S + \overline{b}$	New b = $R + \overline{a}$
√	0 0	0 0	1	1
	0 0	0 1	1	1
	0 0	1 0	1	1
	0 0	1 1	1	1
√	0 1	0 0	1	1
	0 1	0 1	0	1
	0 1	1 0	1	1
	0 1	1 1	0	1
√	1 0	0 0	1	1
	1 0	0 1	1	1
	1 0	1 0	1	0
	1 0	1 1	1	0
√	1 1	0 0	1	1
	1 1	0 1	0	1
	1 1	1 0	1	0
√	1 1	1 1	0	0

2. If an R-S latch powers up with R S then a b is stable, which is a problem, since it's not a configuration we're using to denote true or false. We can fix the problem by using the R or S switch ($R\overline{\ }$ S takes a b → a $\overline{b}$, and R $S\overline{\ }$ takes a b → a $\overline{b}$.)

3. If an R-S latch powers up with a and b having random values and $R\overline{\ }$ $S\overline{\ }$, then we go to a b as in the previous question. Again, it's a problem and again it

can be fixed using the technique in the previous problem (use $R\overline{\ }$ S or R $S\overline{\ }$ and then R S).

4. When P = 0, the NAND gate produces 1 regardless of its other input, so Q becomes 1 and stays there. When P = 1, the NAND gate produces the NOT of its other input, so Q's logical value flips back and forth and is not stable. This circuit doesn't remember a bit; it's more like a combinatorial function that breaks on certain inputs.

5. The new $Z = A\overline{S} + SZ$, so if S = 0 then the new Z = A. If S = 1 then new Z = Z (whether Z = 0 or 1). This circuit is a storage element, with Z as its state: If S = 1 then Z remains unchanged. To change Z, set A to the value of Z you want, then set S = 0 to change Z, and then set S = 1 to maintain Z.

6. The address space is the set of legal addresses. Ours has size 2^k. The total amount of memory is $m\,2^k$ bytes. In theory, there's no relationship between k and m. A typical computer, however, can fit an address within a word, so in practice, if we have word addressability (i.e., if m > 1), then we probably have $k \le m$.

7. A gigabyte (GB) = 2^{30} bytes, so with -32bit addresses, we can access 2^{32} bytes of memory = 4 gigabytes (4 GB). With -64bit addresses we can address $2^4 * 2^{60}$ bytes = 16 exabytes (16 EB).

Storage Elements and Memory

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Storage elements are the basic circuits that store data, which is used in logic circuits that have a state (i.e., use memory).
- Memory is used to store data for I/O and computations.

B. Outcomes

After this activity, you should be able to:

- Be able to identify and trace the execution of basic storage elements.
- Know how memory combines address decoders, multiplexers, and data storage/update.

C. Questions

1. Complete the table below, which shows the 5 configurations for an R-S latch that didn't appear in the notes.

R S	a b	New a = S $+\overline{b}$	New b = R $+\overline{a}$
0 0	0 0		
0 1	0 0		
1 0	0 0		
1 1	0 0		
1 1	1 1		

2. Say an R-S latch powers up with $\overline{a}$ $\overline{b}$ and R = S = 1. What happens? Is this a problem? If so, suggest a fix.

2a. (Execution trace):

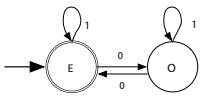
	0	1		0	0	0	1	1	0	
None	0	None		0	00	00	Acc	Acc	Acc	Acc

2b. It accepts any string that contains a 001

2c. (Transition table): Initial state: "None"; accepting state: Acc.

State	Input	New State
None	0	0
None	1	None
0	0	00
0	1	None
00	0	00
00	1	Acc
Acc	0	Acc
Acc	1	Acc

3a. Note that since we're tracking only the 0's, the 1's don't change the state. Each 0 changes the state from even to odd (or vice versa).



3b. Initial and accepting state: E

State	Input	New State
E	0	O
E	1	E
O	0	E
O	1	O

4. Output and transition table:

C_1 (State hi bit)	0	0	0	1	1	0	0	0
C_0 (State low bit)	0	0	1	0	1	0	1	0
S (On-Off Switch)	0	1	1	1	1	1	0	0
L_{12} (Lights 2 & 1)	0	0	1	1	1	0	1	0
L_{34} (Lights 4 & 3)	0	0	0	1	1	0	0	0
L_5 (Light 5)	0	0	0	0	1	0	0	0
N_1 (new C_1)	0	0	1	1	0	0	0	0
N_0 (new C_0)	0	1	0	1	0	1	0	0