Von Neumann Computers And An Example

CS 350: Computer Organization & Assembler Language Programming

9/30: Solved; 9/24: pp.10+ (add immediate, conditional branch)

A. Why?

- The computers we use follow the von Neumann model/architecture.
- A simple decimal example illustrates how von Neumann computers work without worrying about binary representation and more complicated instruction sets.

B. Outcomes

After this lecture, you should know

- The von Neumann architecture and how it differs from other architectures.
- The basic design of a von Neumann computer and CPU.
- The different parts of the instruction cycle and what happens during them.
- How to trace instruction execution for a simple computer.

C. Mechanical Computation Devices

- **Jacquard's Loom** (early 1800s): First machine to perfect use of punch cards to direct its activity (controlling pattern of weave). Not a computer but introduced notion of changeable control of a device.
- **Difference Engine (Charles Babbage, 1820s–1840s)**: Designed to do calculations of values of polynomials. Was technically possible but not actually completed. (Manufacture required stronger tolerances than was easily available, British government cut off funding, and he had become interested in a more general machine.
- **Analytical Engine (Babbage; 1830s on)**: Designed to be a general purpose computer. Instructions and data would've been entered using punch cards. Instructions included tests and loops; machine would've been equivalent to Turing machines.

- Programming language Ada is named after Ada Byron (Countess of Lovelace), who wrote programs for the Analytical Engine.

D. Electro-Mechanical Computation Devices

- **Konrad Zuse (1930s-1940s)**: Built Z machines that did computations using mechanical and electric relays. The Z3 was equivalent to Turing machines and could be programmed using punch tape (not rewiring).
- **Harvard Mark I (1940s)**: also used switches, relays, other mechanical devices. Could execute long instructions automatically. Controlled by punch tape (a loop could be written by connecting the two ends of the tape.

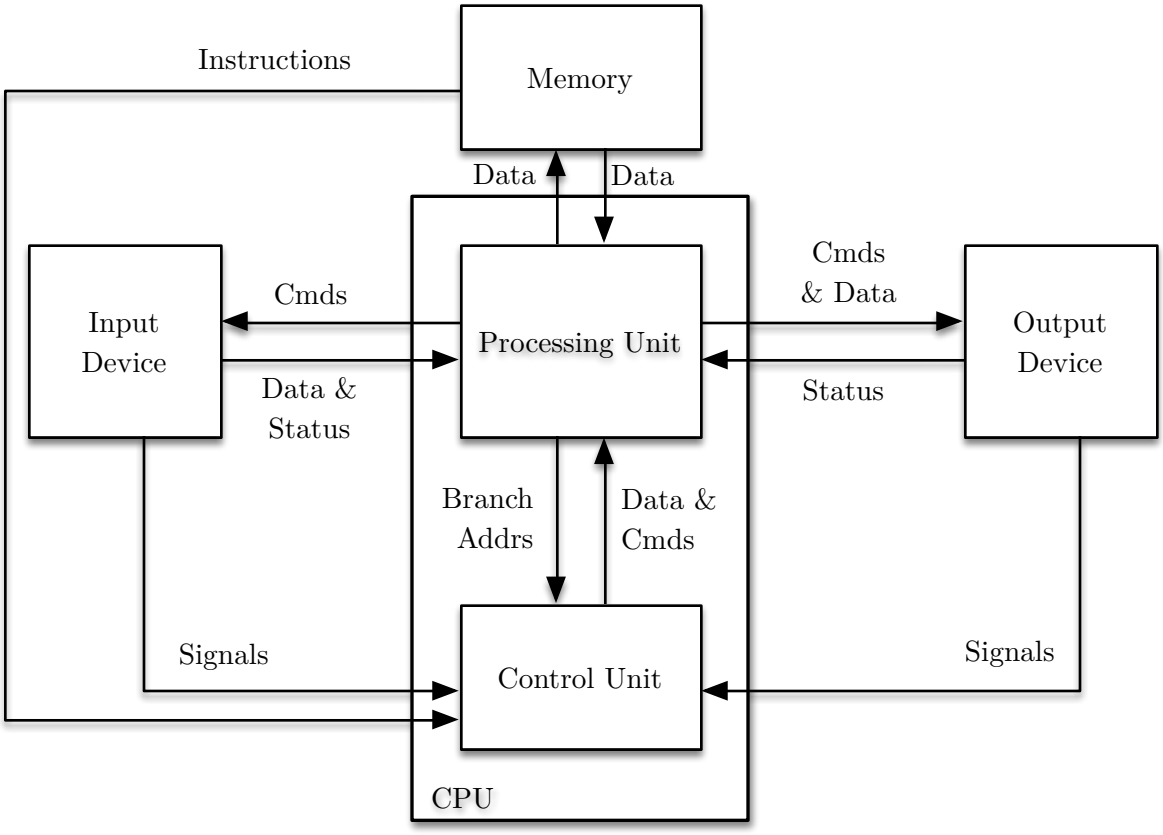
E. Early Electronic Computation Devices

Fixed-Program Computer

- In a fixed-program computer, the program is not changeable at runtime.
- 1939: ABC (Atanasoff-Berry Computer) [Wikipedia photo]
  - Electronic machine. No CPU, but did mathematical computations & binary logic. Not programmable (built for one kind of problem — solving linear equations).
  - Adapted from Wikipedia: The ABC weighed > 700 lb. and was 800 feet². It contained ~1 mile of wire, 280 dual-triode vacuum tubes, 31 [vacuum tube diodes], and was about the size of a desk.
- 1943: ENIAC (Electronic Numerical Integrator And Computer) [Wikipedia photo]
  - Presper Eckert and John Mauchly -- first general purpose electronic computer. Decimal computer, not binary. Program hard-wired through dials & switches.
  - Adapted from Wikipedia: ENIAC contained 17,468 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors and around 5 million hand-soldered joints. It weighed 27 tons, was roughly 8.5'×3'×80', took up 1800 feet², and consumed 150 kW of power. On average a tube failed every 2 days and took 15 min to find.

Stored-Program Computer (von Neumann Computer)

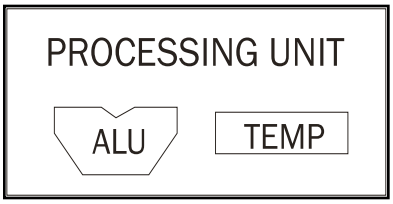
- A stored-program computer has modifiable programs, stored in read-write memory along with data.
- 1944, EDVAC (Electronic Discrete Variable Automatic Computer) [Wikipedia photo], John von Neumann co-authors report on stored program concept.
- Adapted from Wikipedia: The EDVAC contained ~6,000 vacuum tubes, 12,000 diodes, 1,500 relays. It weighed 8.5 tons, took 490 feet², and consumed 56 kW of power. The EDVAC had ~ 5.5 kB of *ultrasonic serial memory*: Electrical pulses were transformed by a crystal speaker into pressure pulses that traveled through a column of mercury. Pulses received at the other end were converted back to electricity using crystal microphones, amplified, and sent back to repeat.



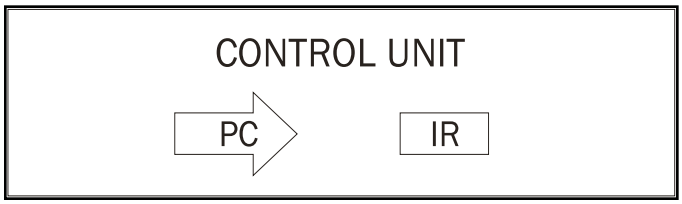
Typical von Neumann Computer

F. Basic von Neumann Architecture

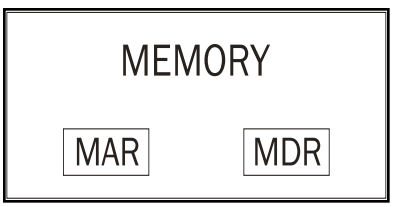
- Under the von Neumann architecture, a computer has three main parts: **Read-Write Memory**, containing instructions and data; **Input and Output Devices**; and a **CPU (Central Processing Unit)**, which includes a **Control unit**, which interprets instructions and a **Processing unit**, which does arithmetic/logical operations.
- The traditional processing unit is an **ALU (Arithmetic Logical Unit)**. It contains



  - "General-purpose" **Registers**: Small fast temporary storage.
    - Limited number (expensive)
    - The machine's **Word Size** is the width of the registers.
  - Circuitry to perform calculations on data
  - ALU action is directed by the **Control Unit**
    - ALU can get data from control unit, may send addresses to the control unit for branches or jumps (i.e., goto's)
    - ALU also sends data to memory/receives data from memory. ← Fri 9/23
- The **Control Unit (CU)** directs execution of the program. The **Instruction Register** (`IR`) contains the current instruction. The **Program Counter** (`PC`) contains the memory address of the next instruction to be executed. (**"Instruction Pointer"** would probably be a better name, but `PC` is traditional.)



- **Memory:** Each address is K bits long and stores M bits. In theory K and M are independent; in practice $K \leq M$.



  - **MAR: Memory Address Register** (K bits wide): Contains address to read/write
  - **MDR: Memory Data Register** (M bits wide): Contains value read/written

- **Input & Output Devices**
  - Keyboard, mouse, disk, video, printer, etc. Receive commands from processing unit (or control unit; depends on design). Send/Receive data. Sends **interrupt signals** to control unit when done with operation.
- Actual modern computers are more sophisticated than the original von Neumann design
  - Many processing units (e.g. floating-point)
  - I/O processing off-loaded to special hardware (e.g. video cards)
  - I/O goes directly into/out of memory (avoids CPU bottleneck)
  - Control unit works on > 1 instruction at a time (execute current instruction while decoding next instruction while loading one after that into memory).
  - Can have multiple control unit/processing unit pairs (multi-core CPUs)

G. Instruction Format, Instruction Set Architecture

- An instruction has an identifying **Opcode** (Operation code) and some **fields** (bit strings) that specify some numbers (possibly 0) of operands and results. It might also contain extra flags (like to specify when to a go to), or to differentiate between related operations (read a character? A string?)
- **Instruction length**
  - **Fixed-length** instruction machine: Easier to design, limits instruction designs.
  - **Variable-length** instruction machine: More flexible instruction sets, more compact use of memory, hardware more complicated.
- **Instruction Set Architecture (ISA)**
  - The set of instructions for a computer, their formats, the way they specify opcodes and operands and results.
  - It's the fundamental interface between programmers and hardware designers.

H. Addressing Modes

- A large part of how instructions work has to do with accessing operands: they might be in memory or a data register or they might be contained within the instruction

itself. An **addressing mode** is a technique for specifying the location of an operand or result.

- First off, an operand might be part of the actual instruction; these are **immediate** operands. (E.g., the 1 implicit in `++x`.)
- If the data is in a **register**, then you need some bits to specify which one.
- The most obvious way for an instruction to indicate a memory address is to just have the address as a bitstring within the instruction. This is called **absolute addressing** and actually isn't as useful as you might think. First, you can only use it if (instruction size ≥ the opcode width + address width); second, you can't change what address you're accessing unless you change the instruction itself. (Makes going through an array trickier.)
- In practice, a data register can contain an address, so there are a number of techniques that use the value of a data register (a **base** register) or the program counter. In addition, we might have a constant "offset" — that way we can handle a bunch of addresses that are near each other without using multiple registers.
- In addition, there are addressing modes that use pointers for indirection (which we will see) and auxiliary index registers (which we won't).

I. The Instruction Cycle

- A computer executes a program by executing a sequence of instructions: Get instruction from memory, execute it; get instruction from memory, execute it, etc. Whether a bitstring is an instruction or data depends on how we use it.
- The **instruction cycle** is how one instruction is executed. Different people break up the cycle into different numbers of phases; the Patt & Patel book uses 6:

  (1) Fetch instruction

  (2) Decode instruction

  (3) Evaluate address(es) of operands

  (4) Fetch operand(s)

  (5) Execute instruction

  (6) Store results of execution.

- Details vary by instruction; not all instructions have phases 3, 4, 6 (depends on the numbers of operands and results).
- Look at generic instruction cycle first; study a specific version in a bit.
- **Program Counter (`PC`)**: The `PC` is a special register that (most of the time) holds the address of the next instruction to execute. It only holds the address of the current instruction at the very beginning of the Fetch Instruction phase of the instruction cycle.

1. Fetch Instruction

- Read instruction into Instruction Register: Treat the Program Counter value as a memory address; get the contents of that address; copy those contents into the Instruction Register. Microcode (shorthand): `IR` ← `Mem[PC]`
  - Increment Program Counter (so that it points to next instruction in memory). Microcode: `PC` ← `PC+1`

2. Decode Instruction

- Feed opcode bits to decoder to figure out which instruction we have. (E.g. `IR[15:12]`, for a 4-bit opcode stored at the left end of a 16-bit instruction.) Depending on the instruction, retrieve bits from other parts of the `IR`.

3. Evaluate Addresses (= Get effective addresses of operands)

- If the instruction has operands, then for each operand, figure out which register or memory location the operand is stored at. E.g., `IR[7:4]` could be a 3-bit register number, or `PC+IR[15:7]` could be an address. Not all instructions have this phase.

4. Fetch Operands

- If the instruction has operands then retrieve their values. They might be in registers, in memory, or encoded within the instruction itself. Not all instructions have this phase.

5. Execute Instruction

- There are 3 general kinds of instructions.
  - **Data movement instruction**: Load value from/Store value to memory.

- **Calculation instruction**: Perform some operation on data (add, etc).
- **Control Instruction** (Branch/Jump): Go to a different part of the program (used in decision and iteration statements).

6. Store Results

- Take result(s) of load-from-memory instructions and calculation instructions and put them somewhere. (Might be register(s), memory locations.) Instructions that don't produce results (like store to memory, or go to location) don't have this phase.

J. Simple Decimal Computer

- Eventually we'll be looking at the Patt & Patel's LC-3 computer, but before that, let's look at a simpler computer. To make it simple, we'll use decimal data, the instruction set will be very limited and we'll use absolute addressing.
- The SDC (a Simple Decimal Computer) has a hundred memory locations numbered `00` – `99` and ten arithmetic registers named `R0`, `R1`, …, `R9`. Each memory location and register holds a signed 4-digit number. Below, we indicate the contents of register *R* using `Reg`[*R*] and the contents of memory location *MM* using `Mem`[*MM*].
- Instructions are 4 digit numbers and have the form ±*NRMM*, where *N* is an opcode (`0-9`), *R* is a register number (`0-9`), and *MM* is a memory location (`00-99`).
  - The instruction *Sign* is `-1` or `1` for negative/nonnegative instructions. *Sign* is generally ignored (with opcodes 5, 6, and 8 below as exceptions).
- It's important to differentiate between opcodes 1 (`LD`) and 5 (`LDM`). Both set `Reg`[*R*] to a value: *Load* (`LD`) uses `Mem`[*MM*], the value at memory location *MM*, but *Load Immediate* (`LDM`) uses *MM* literally (we say it's an **immediate operand** when it's part of the instruction this way). Opcodes 3 (`ADD`) and 6 (`ADDM`) are similar: Add sets `Reg`[*R*]←`Reg`[*R*] + `Mem`[*MM*] but *Add Immediate* sets `Reg`[*R*] ← `Reg`[*R*] + *MM*.
- Note: For `LDM` and `ADDM`, we take the sign of *MM* to be the sign of the overall instruction: `5123` says to load immediate `R1` with `23`; `-5123` says to load

immediate `R1` with `-23`. This difference is handy if you want to decrement a register. (E.g., `-6201` sets `R2` ← `R2 - 1`.)

- For I/O, instead of using *R* as a register, we use it to specify the kind of I/O to do. For operations `90`, `91`, `93`, and `94`, we ignore *MM*.
  - Operation `90` reads a character from the keyboard and copies its ASCII numeric representation to `R0`.
  - Operation `91` is the opposite of operation `90`: It prints the character whose ASCII numeric representation is in `R0`.
  - Operation `92` prints out a string. The string should be represented as a sequence of characters in memory (one character per location) with one extra trailing memory location containing 0. The location of the first character is specified by *MM*. E.g., for instruction `9225`, if locations 25 – 28 contain `65`, `66`, `67`, `0`, then we print `ABC`.
  - Operation `93` dumps the contents of the control unit (`PC`, `IR`, and data registers) to the display.
  - Operation `94` dumps the contents of memory to the display.
- **SDC Execution**: When the SDC powers up, it reads a sequence of 4-digit numbers into memory locations `00`, `01`, …. When it reaches the end of the numbers, it sets all registers to `0`, sets the `PC` to location `00`, sets the *Running* flag to *true*, and begins the instruction cycle, which is a loop implemented via:

  while *Running and* `0` ≤ `PC` ≤ `99` {
  Fetch instruction: `IR` ← `Mem[PC]`*;* `PC++`
  Decode instruction: Set Op, R, and MM to `IR`[3]*,* `IR`[2]*,* `IR`[1:0]*;*
  set Sign to `1`*,* `-1` *or* `0` *depending on* `IR` >, <, or = 0.
  Get operands, execute instruction, store results:
  if *Op* = `0` **then** *Running* ← *False*
  else if *Op* = `1` **then** ….
  }

Opcode	*Semantics*	*Implementation*
0	HALT execution. (Ignore *R* and *MM*.)	*Running* ← *false*
1	LD (Load) Reg[*R*] with the value of memory location *MM*.	Reg[*R*] ← Mem[*MM*]
2	ST (Store) Copy the value of Reg[*R*] into memory location *MM*	Mem[*MM*] ← Reg[*R*]
3	ADD contents of location *MM* to Reg[*R*]	Reg[*R*] += Mem[*MM*]
4	NEG: Set Reg[*R*] to its arithmetic negative (ignore *MM*)	Reg[*R*] ← (– Reg[*R*])
5	LDM (Load immediate): Load Reg[*R*] with *MM*	Reg[*R*] ← *MM*
–5	LDM (Load immediate): Load Reg[*R*] with –*MM*	Reg[*R*] ← –*MM*
6	ADDM (Add immediate): Add *MM* to Reg[*R*].	Reg[*R*] += *MM*
–6	SUBM (Subtract immediate): Add –*MM* to Reg[*R*].	Reg[*R*] += –*MM*
7	BR (branch): Go to location *MM* (ignore *R*)	PC ← *MM*
8	BRGE (branch ≥ 0): If Reg{*R*] ≥ 0, go to location *MM*	**if** Reg[*R*] ≥ 0 **then** PC ← *MM*
–8	BRLE (branch ≤ 0): If Reg{*R*] ≤ 0, go to location *MM*	**if** Reg[*R*] ≤ 0 **then** PC ← *MM*
90	GETC Read a character and copy its ASCII representation into R0. ('A' sets R0 = 65, etc.)	Reg[0] ← *Char from Keyboard*
91	OUT Print the character whose ASCII representation is in R0. ('A' for 65, etc.)	*Print* Reg[0] as a char

92	PUTS Print a string (i.e., the characters) at locations *MM*, *MM+1*, …. Stop (and don't print) when we get to a location that contains 0. (Don't print the zero.)	temp ← *MM*
		while Mem[temp] ≠ 0
		  Print Mem[temp];
		  temp++

93	DMP Print out the values of the control unit registers (the PC, IR, and the registers)	*Dump Control Unit*
94	MEM Print the non-zero values in memory	*Dump Memory*
95–99	Ignore instruction	

SDC Instruction Set Architecture

- The SUBM (subtract immediate), BRGE (branch if ≥ 0), and BRLE (branch if ≤ 0) differ from the Spring 2016 version of the SDC.

K. SDC Examples

- **Example 1**: Add 1 to memory location 99, using *R3* as temporary storage:

```
1399  ; R3 ← Mem[99] (Load R3 from memory location 99)
6301  ; R3 ← R3 + 1
2399  ; Mem[99] ← R3 (Store R3 into memory location 99)
```

- **Example 2**: Set `R1` to `-1`
  - Version (a): Use load immediate

```
-5101  ; R1 ← -1 (Load immediate R1 with -1)
```

  - Version (b): Load it with `1` and take its negative.

```
5101  ; R1 ← 1 (Load immediate R1 with 1)
4100  ; R1 ← -R1
```

  - Version (c): Load `R1` with a location that contains `-1`

```
1199  ; R1 ← -1 (Load R1 with Mem[99], where Mem[99] = -1)
```

- **Example 3**: Add `R1` to `R0`; if the result is ≥ 0, branch to location `57`; if the result is < 0, branch to location `62`. (We use location `98` as temporary storage).

```
2198  ; Mem[98] ← R1 (Temporarily save R1 into location 98)
3098  ; R0 ← R0 + Mem[98] = R0 + R1
8057  ; Branch conditional to location 57 if current R0 ≥ 0
      ;     (i.e., if old R0 + R1 ≥ 0)
7062  ; (else) Branch to location 62
```

- If the sum is ≥ 0, then the conditional branch to `57` will be done; if the sum is < 0, then we'll fall through to the next instruction, which branches to location `62`. It's an unconditional branch to `62`, but it's okay because we only get to this instruction if the sum was not ≥ 0.

- **Example 4**: Print "`Hello, world!`" Assume the first instruction is at location `00`.

```
9202  ; PUTS string starting at location 02.
0000  ; HALT
0072  ; 'H'
0101  ; 'e'
0108  ; 'l'
0108  ; 'l'
0111  ; 'o'
0044  ; ','
0032  ; ' '
0119  ; 'w'
0111  ; 'o'
0114  ; 'r'
0108  ; 'l'
0100  ; 'd'
0033  ; '!'
0000  ; end of string
```

- The address of the string is part of the `PUTS` instruction, so it's **really** hard-coded in there.
- If we ever change the location of the string (e.g., by making the program longer), we'll need to update the `PUTS` instruction to use the new location of the string.
- A partial workaround is to put the string at an address far away from the program so that lengthening the program is less likely to cause a problem.

L. SDC Simulator

- There's an SDC simulator available on `fourier` at `~sasaki/CS350/sdc`. The simulator starts by reading in a file containing SDC instructions. The filename can be specified on the command line (if omitted, it defaults to `default.sdc`).
- The examples in the previous section show the format of the input file: The instructions should be written as decimal numbers, one per line, with optional comments. The simulator loads these values into location `00`, then `01`, etc.
- On end-of-file, the simulator offers a prompt at which you can type one of a number of commands: `h` for help, `d` to dump the CPU, `q` to the quit the simulator, and an integer ≥ 1 to execute that many instruction cycles (or until the machine has HALTed. All these commands should be followed by a carriage return. In addition, you can just press carriage return (with no input) to execute one instruction cycle.

- We'll go through examples of the simulator in class, but you definitely should play with it yourself.

M. Instructions vs Data

- When we write low-level programs, we may think of some memory locations as having instructions and other locations as having data, but the CPU treats everything as data except for the value inside the **Instruction Register**.
- E.g., in Example 4, we intend the `0000` at location 1 to be a `HALT` instruction, but in a different context it can be treated as data.
- If we were to print the string starting at location `01` (by making the `PRINT` instruction `9201` instead of `9202`), we would print no characters because we'd see the `0000` at location `01` as an end-of-string.
- On the other hand, we can also treat what we think of as data as instructions ("execute our data"):
- E.g., still in Example 4, if we take out the `HALT 0000` at location `01` so that the '`H`' is now at location `01`, the '`e`' is at location `02`, etc., then the `PRINT 9202` instruction will print "`ello, world!`".
- Then we'll execute the next instruction (the one at location `01`). Location `01` contains `0072` (the letter '`H`') but as an instruction, it means `HALT`, so our program will stop.

Von Neumann Computers And An Example

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 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. Problems

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?
   d. What happens during the Evaluate Addresses (a.k.a Get Effective Addresses)

phase of the instruction cycle? Why is this phase sometimes skipped?

e. What happens during the Fetch Operands phase of the instruction cycle? From where can operands be fetched? Why is this phase sometimes skipped?

f. What happens during the Store Results phase of the instruction cycle? To where can results be stored? Why is the phase sometimes skipped?

Problems 6 – 12 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 `05`.

6. Write code that takes the contents of memory location `90`, doubles it, and stores it back in location `90`.

7. 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`

8. (Conditional branches)

   a. Write some code that goes to (i.e., branches to) location `12` if `M[95]` is ≤ `0`.

   b. Write some code that goes to location `12` if `M[95]` < `0`. (Hint: Test for `M[95]+1` ≤ `0`.)

   c. Write some code that goes to location `12` if `R1 = 0`. (Hint: Use two tests; one to see if `R1` ≥ 0 and another for ≤ 0. If both hold, then `R1` = 0.)

   d. Write some code that goes to location `12` if `R1` ≠ `0`. (Hint: Check for `R1` < `0` or `R1` > `0`; i.e., `R1` not ≥ `0` or `R1` not ≤ `0`.)

9. Say location `00` contains `1200`. What happens if we execute the instruction at location `00`?

10. Write code to implement the following loop; use `BRGE` to jump to the top of the loop: `R0 ← M[20]; { ++R1; --R0; }` **`while`** `(R0 ≥ 0)`;

11. Write code to implement the following loop; use `BRLE` to exit the loop and `BR` to jump to the top of the loop: `R0 ← M[20];` **`while`** `(R0 > 0) { ++R1; --R0; }` ;

Activity 9 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.
   (d) 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

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

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

7. Set R0 ← R0 – R1 (destroying R1 and using location 99 as temporary storage):

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

8a. Go to location 12 if M[95] ≤ 0 (using R9)

```
 1995  ; R9 ← M[95]
-8912  ; go to 12 if R9 ≤ 0
```

8b. Go to location 12 if M[95] ≤ 1 (uses R9)

```
 1995  ; R9 ← M[95]
-6901  ; R9 ← M[95]-1
-8912  ; go to 12 if R9 < 0 (i.e., M[95] ≤ 0)
```

8c. Go to location 12 if R1 = 0. (We use two tests, one to see if R1 ≥ 0 and another for ≤ 0. If it passes both tests, R1 = 0.)

```
 8107  ; if R1 ≥ 0, go to R1 ≤ 0 test     @ 05
 7008  ; R1 < 0 so skip R1 ≥ 0 test       @ 06
-8112  ; (R1 ≥ 0) go to 12 if R1 also ≤ 0 @ 07
```

8d. Go to location 12 if R1 ≠ 0. (We check R1 ≥ 0 and if necessary R1 ≤ 0. If either test fails, then R1 ≠ 0, so we go to 12.)

```
 8107  ; if R1 ≥ 0, go to R1 ≤ 0 test @ 05
 7012  ; R1 < 0 so go to 12           @ 06
-8109  ; (R1 ≥ 0) check for R1 ≤ 0    @ 07
 7012  ; R1 < 0 so go to 12           @ 08
 ....                                 @ 09
```

9. Since M[00] = 1200, executing the instruction at location 00 means we execute the instruction 1200 (load R2 ← M[00] = 1200).

10. The loop: R0 ← M[20]; { --R0; } while (R0 ≥ 0);

```
 1020 ; R0 ← M[20]        @00
 6101 ; R1 ← R1 + 1       @01
-6001 ; R0 ← R0 - 1       @02
 8001 ; continue if R0 ≥ 0 @03
```


11. The loop: `R0 ← M[20]; while (R0 > 0) { ++R1; --R0;}`

```
 1020 ; R0 ← M[20]            @00
-8005 ; exit loop if R0 ≤ 0   @01
 6101 ; R1 ← R1 + 1           @01
-6001 ; R0 ← R0 - 1           @02
 7001 ; go to top of loop     @03
```

