Lecture 3

COP3402 FALL 2015 – DR. MATTHEW GERBER – 8/31/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

More on the Tiny Computer

- Instruction Format
- Additional Instructions
- Flags and Conditions
- Basic Assembly
- Slightly Less Tiny Computers

The Tiny Computer: Wrapping Up The ISA

The Instruction Format

Like everything else in a computer, an instruction is a binary number.

The Tiny Machine is what's called a *one-address* architecture. Each instruction has:

- An opcode
- Exactly one address as a parameter
- Nothing else

OP	Address
####	#### #### ####
LOAD	0000 0001 0001
0001	0000 0001 0001

Somewhat arbitrarily, we'll use 16-bit instruction words

- 4 bits for the opcode
- 12 bits for the address (we mentioned that this is a *tiny* machine)

Eventually, that LOAD has to become its opcode 0001

- We'll get to assemblers soon enough…

Some New Instructions

04 SUB *Just like ADD, except subtracts instead of adds.*

05 IN *Transfer a value from a given input device into the accumulator.*

- Uses the address parameter to determine the device number.
- Full implementation is beyond the scope of this class.

06 OUT *Transfer a value from the accumulator to a given output device.*

08 JMP *Branches to an address.*

- PC ← ADDRESS

09 SKIPZ *Skips the next instruction if the accumulator contains zero.*

0A SKIPG *Skips the next instruction if the accumulator is greater than zero.*

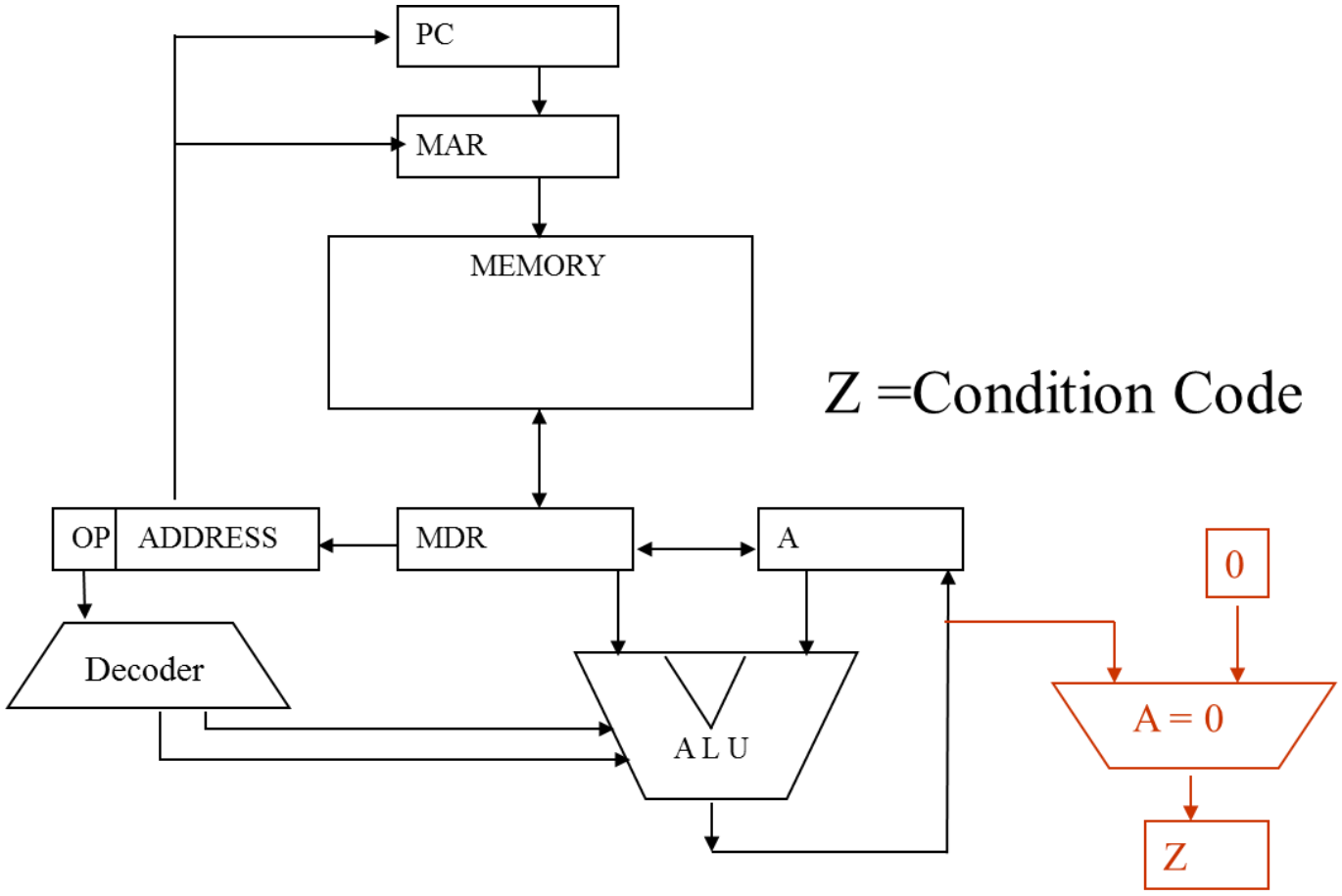
0B SKIPL *Skips the next instruction if the accumulator is less than zero.*

A Note on Conditions

We don't actually compare against the accumulator every time we call a SKIPZ, SKIPG or SKIPL instruction – that would make branching too expensive.

Instead, there's dedicated logic to determine, after every instruction, the state of the accumulator. Based on that state, the logic sets *flags*.

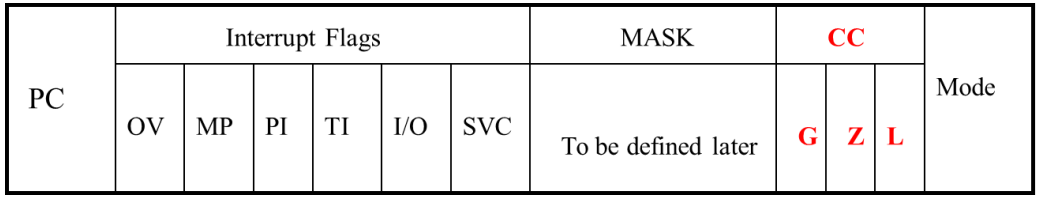
Here we show a notional data path for the Z flag, or zero flag. G and L flags are similar and straightforward.



The Program State Word

The PSW is a special-purpose register that provides the OS – and the computer itself – with information on the status of the running program.

- At least notionally, it includes the program counter
- Here we highlight the G, Z and L flags
- We'll discuss the other flags when we get to interrupt handling
- That's going to be a while



The Complete Tiny ISA

We now have a full, usable instruction set. *With this instruction set, we could write any computer program that can be written.*

- This does not mean it's actually a good architecture in the real world

Thus far, we've been intentionally vague about the distinction between *opcodes* and their associated *mnemonics*.

- That stops now
- The translation between these two is the most essential job of the *assembler*
- Let's talk about those

Op	Mnemonic	Meaning
0	(Fetch cycle)	Fetch next instruction
1	LOAD <addr>	Load <addr> contents into A
2	ADD <addr>	Add <addr> contents into A
3	STORE <addr>	Store A contents into <addr>
4	SUB <addr>	Sub <addr> contents from A
5	IN <device>	Read into A from <device>
6	OUT <device>	Write to <device> from A
7	HALT	End program
8	JMP <addr>	Branch to <addr>
9	SKIPZ	Skip next instruction if Z (A = 0)
A	SKIPG	Skip next instruction if G (A > 0)
B	SKIPL	Skip next instruction if L (A < 0)

The Tiny Computer: Assembly Language

A Very Simple Program

MEMORY BEFORE

Location	Contents
000	1245_{10}
001	1755_{10}
002	0_{10}
003	Load <000>
004	Add <001>
005	Store <002>
006	Halt

MEMORY AFTER

Location	Contents
000	1245_{10}
001	1755_{10}
002	3000_{10}
003	Load <000>
004	Add <001>
005	Store <002>
006	Halt

Assembling a Very Simple Program

Location	Contents
003	Load <000>
004	Add <001>
005	Store <002>
006	Halt
Becomes…	
003	0001 0000 0000 0000
004	0010 0000 0000 0001
005	0011 0000 0000 0010
006	0111 0000 0000 0000

Op	Mnemonic	Meaning
0	(Fetch cycle)	Fetch next instruction
1	LOAD <addr>	Load <addr> contents into A
2	ADD <addr>	Add <addr> contents into A
3	STORE <addr>	Store A contents into <addr>
4	SUB <addr>	Sub <addr> contents from A
5	IN <device>	Read into A from <device>
6	OUT <device>	Write to <device> from A
7	HALT	End program
8	JMP <addr>	Branch to <addr>
9	SKIPZ	Skip next instruction if Z (A = 0)
A	SKIPG	Skip next instruction if G (A > 0)
B	SKIPL	Skip next instruction if L (A < 0)

Assembler Directives

- The core purpose of the assembler is to translate mnemonics into binary opcodes and their parameters
- However, there are other things an assembler can do for us
- Versions of three essential **directives** are part of any nontrivial assembler
- They are used in conjunction with the assembler's crucial ability to **label** memory locations
- Let's see how this all fits together

.begin	Tell the assembler where the program starts
.data	Reserve a (usually labeled) memory location
.end	Tell the assembler where the program ends; usually given the start location as a parameter

Example Assembly Program

Here we see a complete program that actually does something:

- Reads two numbers *a* and *b* from device 5
- Calculates *a* - 2 + *b*
- Outputs the result to device 9
- Ends

Several notes:

- "x" at the front of a literal number indicates it is hexadecimal, like "0x" in C
- The data field reservations look like variable and constant declarations because that's what they are
- By convention, constants are in all caps, with variables in lower case
- Nothing enforces constants versus variables

Label	Opcode	Address
start	.begin	
	in	x005
	store	a
	in	x005
	store	b
	load	a
	sub	TWO
	add	b
	out	x009
	halt	
a	.data	0
b	.data	0
TWO	.data	2
	.end	start

Extending the Tiny Computer: Load/Store Architectures

Load/Store Architectures

A **load/store architecture** has multiple registers and uses multiple instruction formats. Here's a possible set of formats, with an example for each:

- Single-address
- Register-address
- Three-register

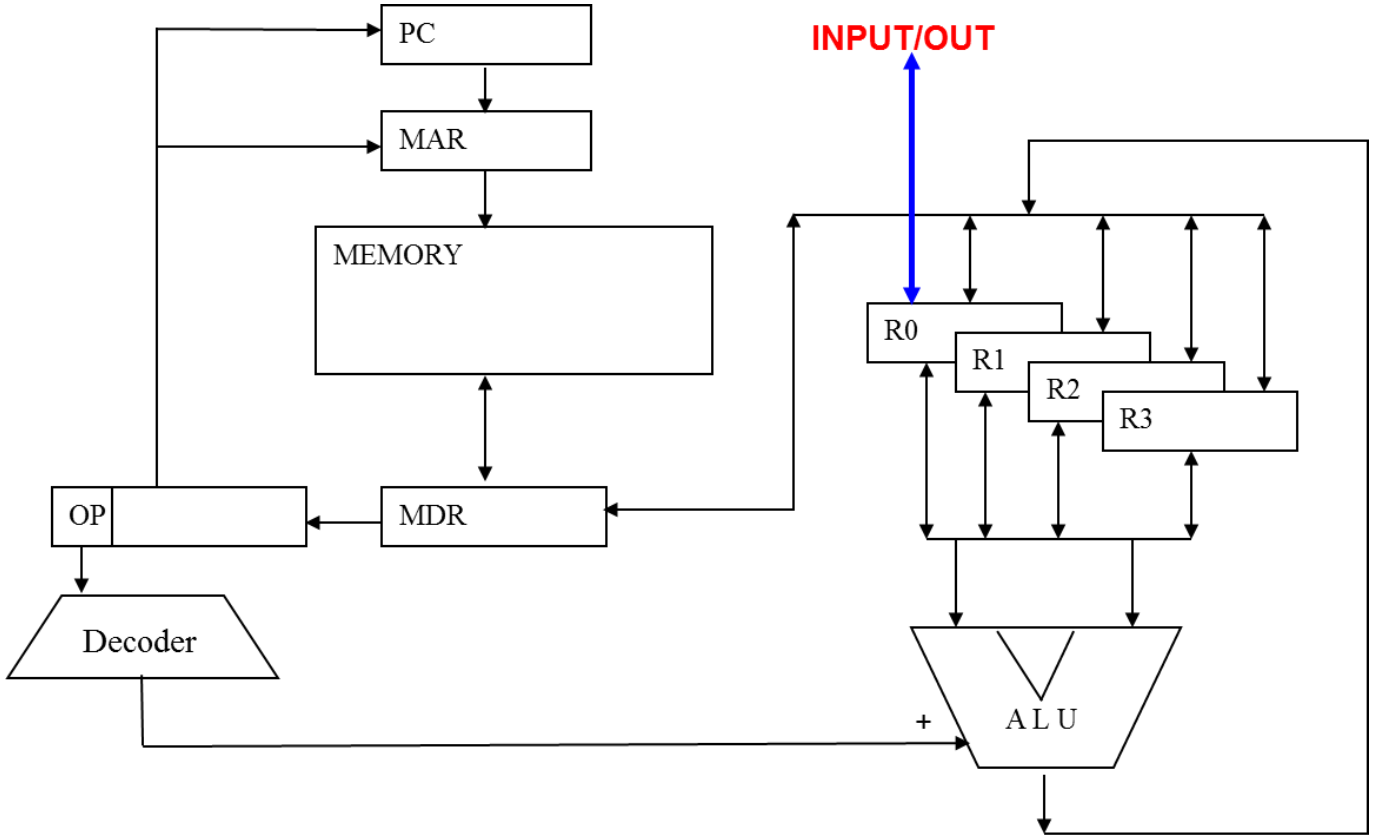
OP	ADDRESS		
JMP <address>			
OP	R_i	ADDRESS	
LOAD R3, <address>			
OP	R_i	R_j	R_k
ADD R3, R2, R1			

Load/Store Architecture Diagram

Here's what the data paths might look like in a load/store architecture.

Here all the registers are general-purpose and able to use, and be used by, the ALU.

We require I/O to happen from R0 just to keep things interesting for the next slide.



One-Address Vs. Load/Store: Multiplication

```
start   .begin
        in      x005
        store   a
        in      x005
        store   b

again   load    result
        add     a
        store   result
        load    b
        sub     ONE
        store   b
        skipz
        jmp     again

        load    result
        out     x009
        halt
a       .data   0
b       .data   1
ONE     .data   1
result  .data   0
        .end    start
```

```
start   .begin
        in      x005
        store   R0, a
        in      x005
        store   R0, b
        load    R2, result
        load    R3, a
        load    R0, b
        load    R1, ONE

again   add     R2, R2, R3
        sub     R0, R0, R1
        skipz
        jmp     again

        store   R2, result
        load    R0, result
        out     x009
        halt
a       .data   0
b       .data   1
ONE     .data   1
result  .data   0
        .end    start
```

A Different Sort of Tiny Machine

A Thought Experiment

We've been organizing our thoughts so far around register architectures.

- That's the only general-purpose computing architecture in common use…
- …but it's not the only *conceptual* architecture

Anything you can do with a set of registers and RAM, you can also do with a stack.

- Recall your postfix/RPN arithmetic
- You did do postfix/RPN arithmetic, right?
- You remember it well, right?
- You remember it very, very well, right?

This means that along with one-address accumulator machines and register-file load/store machines, we can hypothesize *stack machines*.

- You'd probably never actually *build* one of these…
- …but that doesn't mean you can't virtualize it

We do this sort of thing all the time…

- …and we've been doing it a lot longer than you probably think we have

Next Time: Virtual Machines and the P-Machine