Computer Architecture
02-201 / 02-601
The Conceptual Architecture of a Computer

CPU

PC

register 0
register 1
register 2
register 3
register n

CPU performs operations on data in registers (add, subtract, etc.)

registers hold small amounts of data for processing by the CPU

registers & RAM store data as binary numbers

Reading / writing to some special memory addresses may cause peripheral devices like disk, display, etc. to perform a task

RAM:

| word 0 | word 1 | word 2 | word 3 | word 4 | word 5 | word 6 | word 7 | word 8 | word 9 | word 10 | word 11 | word 12 | word 13 | word 14 | word 15 | word m |
**Binary Representation**

**Base 10 (decimal) notation:**

\[
egin{align*}
4 & \times 10^3 \\
2 & \times 10^2 \\
5 & \times 10^1 \\
6 & \times 10^0 \\
\text{+} & \quad 4 \times 10^3 \\
\hline
4 & \text{256}
\end{align*}
\]

**Base 2 (binary) notation:**

\[
\begin{align*}
1 & \times 2^{12} \\
0 & \times 2^{11} \\
0 & \times 2^{10} \\
0 & \times 2^9 \\
0 & \times 2^8 \\
1 & \times 2^7 \\
0 & \times 2^6 \\
0 & \times 2^5 \\
1 & \times 2^4 \\
0 & \times 2^3 \\
0 & \times 2^2 \\
0 & \times 2^1 \\
1 & \times 2^0 \\
\hline
100000101000000
\end{align*}
\]

Computers store the numbers in binary because it has transistors that can encode 0 and 1 efficiently. Each 0 and 1 is a *bit*.

Built-in number types each have a maximum number of bits.
Hexadecimal Representation

Decimal isn’t good for computers because they work with bits. But writing everything in binary would be tedious. Hence, we often use base 16, aka “hexadecimal”:

Base 10 (decimal) notation:

\[
\begin{align*}
4 &\times 10^3 \\
2 &\times 10^2 + \\
5 &\times 10^1 + \\
6 &\times 10^0 \\
\hline
4 &2 &5 &6
\end{align*}
\]

Base 16 (hexadecimal) notation:

\[
\begin{align*}
1 &\times 16^3 + \\
0 &\times 16^2 + \\
A &\times 16^1 + \\
0 &\times 16^0 \\
\hline
1x4096 + 10x16 = 4256
\end{align*}
\]

Need 16 different digits, so use 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
A=10, B=11, C=12, D=13, E=14, F=15
Add CPU instruction

ADD Rd, Rs, Rt

Set register Rd to Rs + Rt

An instruction is encoded as a sequence of bits:

```
0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0
```

1 = ADD  2 = R2  2 = R2  8 = R8

Written as a hexadecimal number: \(1228_{16}\)
The SUB instruction is the same format as ADD, but with a different opcode:

Instruction (aka opcode)       Rd       Rs       Rt
\[0010001000010001010100000\]

2 = SUB          2 = R2          2 = R2          8 = R8

Written as a hexadecimal number: \(2228_{16}\)
Load “Immediate” Instruction

LIMM Rd, value  
Set register Rd to value

For LIMM, the last 8 bits give the value to copy into Rd:

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 1 1 0 1 1 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 = LIMM 6 = R6 200 = value  = C8₁₆

Written as a hexadecimal number: 76C8₁₆
Load Instruction

LOAD Rd, addr

Set register Rd to ram[addr]

For LOAD, the last 8 bits give the address of the memory cell to copy into Rd:

Instruction
(aka opcode)

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

8 = LOAD  4 = R4  136 = addr

= 88_{16}

Written as a hexadecimal number: 8488_{16}
**Store Instruction**

\[ \text{STORE Rd, addr} \]

*Set register \( \text{ram}[\text{addr}] \) to \( \text{Rd} \)*

For LOAD, the last 8 bits give the address of the memory cell to copy into \( \text{Rd} \):

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 9 = \text{STORE} \]
\[ 4 = \text{R4} \]
\[ 136 = \text{addr} \]
\[ = 88_{16} \]

Written as a hexadecimal number:
\[ 9488_{16} \]
An Example Program

LIMM R1, 64 // R1 = 200
LIMM R2, 1E // R2 = 30
ADD R2, R1, R2 // R2 = R1 + R2
STORE R2, 46 // ram[70] = R2

What does this program do?
An Example Program

LIMM R1, 64 // R1 = 200
LIMM R2, 1E // R2 = 30
ADD R2, R1, R2 // R2 = R1 + R2
STORE R2, 46 // ram[70] = R2

What does this program do? Stores 200 + 30 into memory location 70

Similar to the following Go program:

```go
var r1 int = 200
var r2 int = 30
r2 = r1 + r2
var ram70 int = r2
```

Go manages the registers and memory locations for you.

It may keep a variable in a register, memory, or both.
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

LIMM R1, 64  // R1 = 200 ; 7164
LIMM R2, 1E  // R2 = 30 ; 721E
ADD R2, R1, R2  // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

LIMM R1, 64  // R1 = 200 ; 7164
LIMM R2, 1E   // R2 = 30  ; 721E
ADD R2, R1, R2  // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

These integers are stored in the same RAM used for variables:

<table>
<thead>
<tr>
<th>address</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>7164</td>
</tr>
<tr>
<td>11:</td>
<td>721E</td>
</tr>
<tr>
<td>12:</td>
<td>1212</td>
</tr>
<tr>
<td>13:</td>
<td>9242</td>
</tr>
</tbody>
</table>
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

LIMM R1, 64  // R1 = 200  ; 7164
LIMM R2, 1E  // R2 = 30   ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC (“program counter”) that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64 // R1 = 200 ; 7164
LIMM R2, 1E // R2 = 30 ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46 // ram[70] = R2 ; 9246
```

These integers are stored in the same RAM used for variables:

<table>
<thead>
<tr>
<th>Address</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>7164</td>
</tr>
<tr>
<td>11:</td>
<td>721E</td>
</tr>
<tr>
<td>12:</td>
<td>1212</td>
</tr>
<tr>
<td>13:</td>
<td>9246</td>
</tr>
</tbody>
</table>

The CPU has a special register called PC (“program counter”) that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

LIMM R1, 64  // R1 = 200 ; 7164
LIMM R2, 1E  // R2 = 30 ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.
Where is the program stored?

The program is just a sequence of integers that encode for instructions.

LIMM R1, 64  // R1 = 200   ; 7164
LIMM R2, 1E   // R2 = 30    ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.
LIMM R1, 64  // R1 = 100 ; 7164
LIMM R2, 1E  // R2 = 30 ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246
LIMM R1, 64 // R1 = 100 ; 7164
LIMM R2, 1E // R2 = 30 ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46 // ram[70] = R2 ; 9246
LIMM R1, 64  // R1 = 100 ; 7164
LIMM R2, 1E  // R2 = 30 ; 721E
ADD R2, R1, R2  // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

PC = 12
CPU
R0 = 0
R1 = 64
R2 = 1E
R3 = 0
Rn = 0

address    RAM
10:       7164
11:       721E
12:       1212
13:       9242
14:       0
70:       0
LIMM R1, 64 // R1 = 100 ; 7164
LIMM R2, 1E // R2 = 30 ; 721E
ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
STORE R2, 46 // ram[70] = R2 ; 9246

PC = 13
R0 = 0
R1 = 64
R2 = 82
R3 = 0
Rn = 0

address          RAM
10:   7164
11:   721E
12:   1212
13:   9246
14:   0
70:   0
LIMM R1, 64  // R1 = 100 ; 7164
LIMM R2, 1E  // R2 = 30 ; 721E
ADD R2, R1, R2  // R2 = R1 + R2 ; 1212
STORE R2, 46  // ram[70] = R2 ; 9246

PC = 14

CPU

R0 = 0
R1 = 64
R2 = 82
R3 = 0
Rn = 0

RAM

<table>
<thead>
<tr>
<th>address</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>7164</td>
</tr>
<tr>
<td>11:</td>
<td>721E</td>
</tr>
<tr>
<td>12:</td>
<td>1212</td>
</tr>
<tr>
<td>13:</td>
<td>9242</td>
</tr>
</tbody>
</table>

PC = 14:

70: 82
# Ending the program

The computer will keep grabbing an integer, interpreting it as an instruction, and then incrementing PC indefinitely.

To stop this process, you have to add a HALT instruction:

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>ignored</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

0 = HALT

- LIMM R1, 64 // R1 = 200 ; 7164
- LIMM R2, 1E // R2 = 30 ; 721E
- ADD R2, R1, R2 // R2 = R1 + R2 ; 1212
- STORE R2, 46 // ram[46] = R2 ; 9246
- HALT // stop program ; 0000
Input, Output, and 0

Register 0 always has value 0

Stores to memory location FF writes the value to the output

Reads from memory location FF read a value from the input

STORE R3, FF  Write the value R3 to the output
LOAD R4, FF    Read the next input value into R3

Example:

LIMM R1, 64    // R1 = 200          ; 7164
LOAD R2, FF    // R2 = input        ; 72FF
ADD R2, R1, R2 // R2 = R1 + R2      ; 1212
STORE R2, FF   // write R2 to output ; 92FF
HALT           // stop program      ; 0000
Other Arithmetic Operations: AND

**AND Rd, Rs, Rt**

Set register Rd to Rs AND Rt

AND takes two binary numbers a and b and creates a binary c number where the ith bit of c is 1 if and only if the ith bits of both a and b are 1:

| a: | 0 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 0 |
| b: | 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0 0 |
| c: | 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 |

1 because both a and b have 1 in this position.
0 because both a has 0 in this position.
Exclusive Or: XOR

XOR Rd, Rs, Rt

Set register Rd to Rs XOR Rt

XOR takes two binary numbers a and b and creates a binary c number where the ith bit of c is 1 if either a or b but not both have 1 in their ith bit:

<table>
<thead>
<tr>
<th>a:</th>
<th>0 0 1 0 0 0 1 0 0 0 1 0 1 1 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td>0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0</td>
</tr>
<tr>
<td>c:</td>
<td>0 0 0 0 1 0 1 0 0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

0 because both a and b have 1 in this position.

1 because exactly one of a, b have 1 at this position.
Go has bitwise operators:
& = AND
^ = XOR

```go
var r1 int = 200
var r2 int = 30
r2 = r1 & r2
var r3 int
r3 = r2 ^ r1
```

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000011001000</td>
<td>0000000000011110</td>
<td>0000000011000000</td>
</tr>
</tbody>
</table>
Left and Right Shift

**LSHIFT Rd, Rs, Rt**

Set register Rd to Rs shifted to the left by Rt digits.

**Rs:** 1 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 0

**Rd:** 1 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0

- Digits fall off the left and disappear.
- 0s come in at the right.
Left and Right Shift

**LSHIFT** Rd, Rs, Rt

Set register Rd to Rs shifted to the left by Rt digits

**RSHIFT** Rd, Rs, Rt

Set register Rd to Rs shifted to the right by Rt digits

- Rs: 1 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 0
- Rd: 1 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0

Digits fall off the left and disappear

0s come in at the right
### AND, XOR, LSHIFT, RSHIFT

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Rd</th>
<th>Rs</th>
<th>Rt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND Rd, Rs, Rt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Set register Rd to Rs AND Rt</td>
</tr>
<tr>
<td>XOR Rd, Rs, Rt</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Set register Rd to Rs XOR Rt</td>
</tr>
<tr>
<td>LSHIFT Rd, Rs, Rt</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Set register Rd to Rs &lt;&lt; Rt</td>
</tr>
<tr>
<td>RSHIFT Rd, Rs, Rt</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Set register Rd to Rs &gt;&gt; Rt</td>
</tr>
</tbody>
</table>

The instruction format similar to ADD, SUB:

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>Rs</th>
<th>Rt</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0010100101010000</td>
</tr>
<tr>
<td>3 = AND</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>3228&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>4 = XOR</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4228&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>5 = LSHIFT</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>5228&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
<tr>
<td>6 = RSHIFT</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>6228&lt;sub&gt;16&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
Load Indirect

Use the value in a register as an address into RAM to read from:

\[ \text{LOAD.I } Rd, Rt \]

Set register \( Rd \) to \( \text{ram}[Rt] \)

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>ignored</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 = LOAD.I  2 = R2  8 = R8  A208_{16}

What’s the analog in Go of this operation?
**Load Indirect**

Use the value in a register as an address into RAM to read from:

\[
\text{LOAD.I Rd, Rt} \quad \text{Set register Rd to ram[Rt]}
\]

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>ignored</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 0 1 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 = LOAD.I \hspace{0.5cm} 2 = R2 \hspace{0.5cm} 8 = R8 \hspace{0.5cm} A208_{16}

What's the analog in Go of this operation?

Pointer dereferencing with *:

```
var r8 *int = 70 // not legal in Go
var r2 int
r2 = *r8   // LOAD.I R2 R8
```
Store Indirect

Use the value in a register as an address into RAM to write to:

\[
\text{STORE.I Rd, Rt}
\]

Set register ram[Rt] to Rd

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>ignored</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 = LOAD.I  2 = R2  8 = R8  B208_{16}

What’s the analog in Go of this operation?
Store Indirect

Use the value in a register as an address into RAM to write to:

\[
\text{STORE.I Rd, Rt}
\]

Set register \( \text{ram[Rt]} \) to \( \text{Rd} \)

<table>
<thead>
<tr>
<th>Instruction (aka opcode)</th>
<th>Rd</th>
<th>ignored</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111 0010 0100 0000 0000 0100 0000 0000 0000 0000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 = LOAD.I  2 = R2  8 = R8  B208\_16

What’s the analog in Go of this operation?

Pointer dereferencing with * in an assignment:

```
var r8 *int = 70 // not legal in Go
var r2 int
*r8 = r2 // STORE.I R2 R8
```
Summary So Far

• Instructions are encoded as integers stored in memory.

• PC incremented after each instruction.

• Can read / write to memory using either an explicit address (immediate), or the contents of another register as the address (indirect)

• Can perform arithmetic operations on registers.

• Input / Output done via reads/writes to special memory locations.

How would we write an for loop?
**Jumps: Manipulating the PC**

JUMP Rd  
Set PC to Rd  

JMP0 Rd, addr  
If Rd == 0, set PC to addr

```
15: LIMM R1, 1 // 7101
16: LIMM R5, 18 // 7518
17: LIMM R6, 0 // 7600
18: JMP0 R3, 1C // C31C
19: ADD R6, R6, R2 // 1662
1A: SUB R3, R3, R1 // 2331
1B: JUMP R5 // E500
1C: STORE R6, FF // 96FF
```

```
func mul(r2, r3 int) {
    r1 := 1
    r6 := 0

    for r3 != 0 {
        r6 = r6 + r2
        r3 = r3 - r1
    }
    fmt.Println(r2)
}
```
If Statements

JMPP Rd, addr

If Rd > 0, set PC to addr

10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:

if r3 >= r4 {
  fmt.Println(r3)
} else {
  fmt.Println(r4)
}

How would we write a condition a == b?
If Statements

If Rd > 0, set PC to addr

JMPP Rd, addr

10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:

if r3 >= r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}

How would we write a condition a == b?
If Statements

JMPP Rd, addr

If Rd > 0, set PC to addr

10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:

if r3 >= r4 {
  fmt.Println(r3)
} else {
  fmt.Println(r4)
}

condition
then part
else part

if condition was false (r4-r3 > 0)

How would we write a condition a == b?
If Statements

JMPP Rd, addr

If Rd > 0, set PC to addr

```
if r3 >= r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:
```

How would we write a condition a == b?
Example If Statement #2

```go
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```assembly
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMP0 r2, 15
13: STORE r4, FF
14: JUMP r5
15: STORE r3, FF
16: 
```
Example If Statement #2

```go
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMP0 r2, 15
13: STORE r4, FF
14: JUMP r5
15: STORE r3, FF
16:
```
Example If Statement #2

```go
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>LIMM r5, 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>SUB r2, r4, r3</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>JMP0</code> r2, 15</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>STORE r4, FF</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>JUMP r5</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>STORE r3, FF</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Condition:**
- `r3 == r4`

**Then Part:**
- `fmt.Println(r3)`

**Else Part:**
- `fmt.Println(r4)`

**If condition was true:**
- `(r4-r3 == 0)`
Example If Statement #2

```go
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMP0 r2, 15
13: STORE r4, FF
14: JUMP r5
15: STORE r3, FF

Condition: $(r4 - r3 == 0)$

If condition was true, we did the else part.

Skip the then part if we did the else part.
Function Calls

Many processors (including Intel) have explicit “call” and “return” instructions.

X-TOY doesn’t: it has an instruction that lets you write your own RET (RETURN) and CALL functions:

```
JAL Rd, addr
```

Set Rd to PC+1
Set PC to addr

This is “jump and link”: it jumps to an address, and saves where you were in a register.

```
CALL addr ⇔ JAL R15, addr
RETURN ⇔ JUMP R15
```
Function Call Parameters

Note: a function is just a block of instructions that we plan to jump into from elsewhere in the program.

How can we pass parameters into a “function”?
Function Call Parameters

Note: a function is just a block of instructions that we plan to jump into from elsewhere in the program.

How can we pass parameters into a “function”?

Option 1: The caller and the function just agree about which registers to store the parameters in:

```plaintext
// R2 and R3 should contain the // numbers to multiply; R15 should // contain the address to return to
15: LIMM R1, 1    // 7101
16: LIMM R5, 18   // 7518
17: LIMM R6, 0    // 7600
18: JMP0 R3, 1C   // C31C
19: ADD R6, R6, R2 // 1662
1A: SUB R3, R3, R1 // 2331
1B: JUMP R5      // E500
1C: STORE R6, FF  // 96FF
1D: JUMP RF      // EF00
```

```go
func mul(r2, r3 int) {
    r1 := 1
    r6 := 0
    for r3 != 0 {
        r6 = r6 + r2
        r3 = r3 - r1
    }
    fmt.Println(r2)
}
```
Example Call in X-TOY

Program Mul

// Input: None
// Output: 8 * 2 = 16 = 0x10
//
12: FF15 R[F] <- pc+1; goto 15
13: 0000 halt

Function mul

// Input: R2 and R3
// Return address: R15
// Output: to screen
// Temporary variables: R5, R6
15: 7101 R[1] <- 0001
17: 7600 R[6] <- 0000
18: C31C if (R[3] == 0) goto 1C
1B: E500 goto R[5]
1C: 96FF write R[6]
1D: EF00 goto R[F]

Notes:

Program starts at address 0x10

You must say the address of every line of code by prefixing it with addr:
Option 2: Push Parameters onto the Stack

Agree that the stack grows from memory address FE downward towards 0

Agree that R14 always holds a pointer to the top of the stack

```

"PUSH R7"
LIMM R1, 1
ADD RE, RE, R1
STORE.I R7 RE

"POP R9"
LIMM R1, 1
SUB RE, RE, R1
```

0:

RAM

Stack

FE:

FF:
Option 2: Push Parameters onto the Stack

// The top of the stack should contain the two numbers to multiply; R15 should contain the address to return to

10: LOAD.I R2, RE // A20E  \( \text{Grab the number at the top of the stack} \)
11: LIMM R1, 1 // 7101  \( \text{“pop”: move the top of the stack down by 1} \)
12: SUB RE, RE, R1 // 2EE1
13: LOAD.I R3, RE // A30E  \( \text{Grab the number at the top of the stack} \)
14: SUB RE, RE, R1 // 2EE1  \( \text{move the top of the stack down by 1} \)

15: LIMM R1, 1 // 7101
16: LIMM R5, 18 // 7518
17: LIMM R6, 0 // 7600
18: JMP0 R3, 1C // C31C
19: ADD R6, R6, R2 // 1662
1A: SUB R3, R3, R1 // 2331
1B: JUMP R5 // E500
1C: STORE R6, FF // 96FF
1D: JUMP RF // EF00
How many registers are there?

16 in X-TOY
This is a typical number (6-32)

Intel processors have 6 general purpose registers in 32-bit mode, plus some others. They have 16 general purpose registers in 64-bit mode.

What if you “run out”?

Yep, that’s a problem: you may have to shuffle variables between RAM and registers if you need to use the registers for something.
Summary of X-TOY Computer

INSTRUCTION FORMATS

| Format 1: | op | d | s | t |
| Format 2: | op | d | imm |

ARITHMETIC and LOGICAL operations

1: add \( \text{R}[d] \leftarrow \text{R}[s] + \text{R}[t] \)
2: subtract \( \text{R}[d] \leftarrow \text{R}[s] - \text{R}[t] \)
3: and \( \text{R}[d] \leftarrow \text{R}[s] \& \text{R}[t] \)
4: xor \( \text{R}[d] \leftarrow \text{R}[s] ^ \text{R}[t] \)
5: shift left \( \text{R}[d] \leftarrow \text{R}[s] \ll \text{R}[t] \)
6: shift right \( \text{R}[d] \leftarrow \text{R}[s] \gg \text{R}[t] \)

TRANSFER between registers and memory

7: load immediate \( \text{R}[d] \leftarrow \text{imm} \)
8: load \( \text{R}[d] \leftarrow \text{mem}[\text{imm}] \)
9: store \( \text{mem}[\text{imm}] \leftarrow \text{R}[d] \)
A: load indirect \( \text{R}[d] \leftarrow \text{mem}[\text{R}[t]] \)
B: store indirect \( \text{mem}[\text{R}[t]] \leftarrow \text{R}[d] \)

CONTROL

0: halt \( \text{halt} \)
C: branch zero if \( (\text{R}[d] == 0) \) \( \text{pc} \leftarrow \text{imm} \)
D: branch pos. if \( (\text{R}[d] > 0) \) \( \text{pc} \leftarrow \text{imm} \)
E: jump register \( \text{pc} \leftarrow \text{R}[d] \)
F: jump and link \( \text{R}[d] \leftarrow \text{pc}; \text{pc} \leftarrow \text{imm} \)

\( \text{R}[0] \) always reads \( 0 \).
 Loads from \( \text{mem}[\text{FF}] \) come from \text{stdin}.
 Stores to \( \text{mem}[\text{FF}] \) go to \text{stdout}.
X-TOY Environment
Intel 8088 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction</td>
</tr>
<tr>
<td>INC</td>
<td>Increment by 1</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement by 1</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>SHL</td>
<td>Shift left (unsigned shift left)</td>
</tr>
<tr>
<td>SHR</td>
<td>Shift right (unsigned shift right)</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JCXZ</td>
<td>Jump if CX is zero</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not negative</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push data onto stack</td>
</tr>
<tr>
<td>POP</td>
<td>Pop data from stack</td>
</tr>
</tbody>
</table>

Another motivation for the `++` and `--` statements in Go (and C, c++, Java..): They correspond directly to a machine instruction. Has several instructions to push and pop data onto THE stack.

and about 80 others…

http://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086.2F8088_instructions
MacPaint