A Simple Computer

Nerd Chef at work.

```
move flour, bowl
add milk, bowl
add egg, bowl
move bowl, mixer
rotate mixer
...
```
Computing Models

• A simple computer model with a unified notion of “data” and “instructions”
  • “Von Neumann” architecture model
  • The first key idea is a model of “memory”

• Others
  – Computing with a table, state-machines, Turing machines with many procedures, etc.
Memory

- Memory stores bits
- Bits are grouped into larger clusters called **words**
- Each word has an address and contents
  - Address is a memory location’s “Name”
  - Contents are a memory location’s “Value”
- Memory stores “Data” and “Instructions”
- We often refer to addresses symbolically like variables in algebra

Address:   [Blank space]
An Array of Words

- Addresses are organized sequentially in an array
- Addresses are
  - Numerical
  - Symbolic (Label)
- The numerical address is fixed (governed by the hardware)
- Labels are user defined
Words = \{\text{Instructions, Data}\}

- Each word of memory can be interpreted as either binary data (number, character, a bit pattern, etc.) or as an instructions
- Not all bit patterns are valid instructions, however.
- Instructions cause the computer to perform a operation
- A program is a collection of instructions
- In general, instructions are executed sequentially
The execution of a program is governed by a simple repetitive loop.

Typically, instructions are fetched from sequential addresses.

A special register, called the program counter (PC), is used to point to the current instruction in memory.
The Stored-Program Computer

- Instructions and Data are stored together in a common memory
- Sequential semantics: To the programmer all instructions appear to be executed sequentially

Key idea: Memory holds not only data, but *coded instructions* that make up a program.

CPU fetches and executes instructions from memory …
- The CPU is a H/W interpreter
- Program IS simply data for this interpreter
- Main memory: Single expandable resource pool
  - constrains both data and program size
  - don’t need to make separate decisions of how large of a program or data memory to buy
Anatomy of an Instruction

• Instruction sets have a simple structure
• Broken into fields
  – Operation (Opcode) - Verb
  – Operands - Noun
• Recipes provide a near perfect analogy
Instruction Operands

• Operands come from three sources
  - Memory
  - As an immediate constant (part of the instruction)
  - From one of several a special “scratch-pad” locations called “registers”
• Registers hold temporary results
• Most operations are performed using the contents of registers
• Registers can be the “source” or “destination” or instructions
UNC-101

- The UNC-101 is a simple 16-bit computer
- It has
  - 65536 or $2^{16}$ memory locations
  - Each location has 16-bits
  - 15 registers, that are referred to as ($1$-$15$)
  - A special operand, $0$, that can be used anywhere that a register is allowed. It provides a value of 0, and cannot write to it
  - A simple instruction set
Instructions: Concrete Examples

addi $4, $5, 1


• All instructions are broken to parts
  - Operation codes (Opcodes), usually mnemonic
  - Operands usually stylized (e.g. “$” implies the contents of the register, whose number follows)
Arithmetic Instructions

add $D, $A, $B  \quad \text{Reg}[D] \leftarrow \text{Reg}[A] + \text{Reg}[B]

sub $D, $A, $B  \quad \text{Reg}[D] \leftarrow \text{Reg}[A] - \text{Reg}[B]

sgt $D, $A, $B  \quad \text{Reg}[D] \leftarrow 1 \text{ if } (\text{Reg}[A] > \text{Reg}[B])
\quad \text{0, otherwise}

• Where D, A, B are one of \{1,2, \ldots 15\}
• All operands come from registers
Immediate Arithmetic Instructions

\[ \text{addi } \$D, \$A, \text{imm} \quad \text{Reg}[D] \leftarrow \text{Reg}[A] + \text{imm} \]
\[ \text{subi } \$D, \$A, \text{imm} \quad \text{Reg}[D] \leftarrow \text{Reg}[A] - \text{imm} \]
\[ \text{sgti } \$D, \$A, \text{imm} \quad \text{Reg}[D] \leftarrow 1 \text{ if } (\text{Reg}[A] > \text{imm}) \]
\[ \text{0, otherwise} \]

• Where \( D, A \) are one of \( \{1,2, \ldots 15\} \)
• 2 operands come from registers
• Third, “Immediate” operand is a constant, which is encoded as part of the instructions
Multiply? Divide?

• You may have noticed that some math functions are missing, such as multiply and divide.
• Often, more complicated operations are implemented using a series of instructions called a routine.
• Simple operations lead to faster computers, because it is often the case the speed of a computer is limited by the most complex task it has to perform. Thus, simple instructions permit fast computer (KISS principle).
KISS == RISC?

• In the later 20 years of the 1900’s computer architectures focused on developing simple computers that were able to execute as fast as possible
• Led to minimalist, and simple, instruction sets
  - Do a few things fast
  - Compose more complicated operations from a series of simple ones
• Collectively, these computers were called Reduced Instruction Set Computers (RISC)
Load/Store

• Certain instructions are reserved for accessing the contents of memory
• The *only* instructions that access memory
• Move data to registers, operate on it, save it

\[
\begin{align*}
st \ D, & \ A & \text{memory[Reg}[A]\text{]} & \leftarrow & \text{Reg}[D] \\
ld \ D, & \ A & \text{Reg}[D] & \leftarrow & \text{memory[Reg}[A]\text{]} \\
stx \ D, & \ A, \text{imm} & \text{memory[Reg}[A]+\text{imm}] & \leftarrow & \text{Reg}[D] \\
ldx \ D, & \ A, \text{imm} & \text{Reg}[D] & \leftarrow & \text{memory[Reg}[A]+\text{imm}] \\
\end{align*}
\]
Bitwise Logic Instructions

and \( D, A, B \)
\[
\text{Reg}[D] \leftarrow \text{Reg}[A] \& \text{Reg}[B]
\]
or \( D, A, B \)
\[
\text{Reg}[D] \leftarrow \text{Reg}[A] \mid \text{Reg}[B]
\]
xor \( D, A, B \)
\[
\text{Reg}[D] \leftarrow \text{Reg}[A] \uparrow \text{Reg}[B]
\]

• Where \( D, A, B \) are one of \{1,2, \ldots 15\}
• All operands come from registers
• Performs a bitwise 2-input Boolean operation on the bits of the \( A \) and \( B \) operands and saves the result in \( D \)

• Assuming \( \text{Reg}[1] = 12 \) (0x000c) and \( \text{Reg}[2] = 10 \) (0x000a) and \( 3, 1, 2 \) # gives \( \text{Reg}[3] = 8 \) (0x0008)
or \( 3, 1, 2 \) # gives \( \text{Reg}[3] = 14 \) (0x000e)
xor \( 3, 1, 2 \) # gives \( \text{Reg}[3] = 6 \) (0x0006)
Closing the Gap

- A computer language closer to one we’d speak
  - High-Level construct:
  
  - Assembly language:
    - ld $1,$0,item1
    - ld $2,$0,item2
    - add $1,$1,$2
    - st $1,$0,total
  
  - Binary (machine language):
    - 0xf10f, 0x0008, 0xf20f, 0x0009,
    - 0x0112, 0xf10e, 0x0007
An Assembler

• A symbolic machine language
• One-to-one correspondence between computer instruction = line of assembly
• Translates symbolic code to binary machine code
• Manages tedious details
  - Locating the program in memory
  - Figures out addresses
    (e.g. item1 rather than 0x0008)
• Generates a list of numbers
Assembly Code

main:   add $1,$0,$0
       add $2,$0,$0
loop:   ldx  $3,$2,item
        sgei $4,$2,10
        bne $0,$4,$0,done
       addi $2,$2,1
       beq $0,$0,$0,loop
done:   stx $1,$0,total
end:    beq $0,$0,$0,end

# $1 = total
# $2 = index
# $3 = item[index]
# total = total + $3
# if (index >= 10)
# we're done
# next index
# loop back
# save total
# the end

total: .data 0
item:   .data 1,3,5,7,9,11,13,15,17,19
Assembly Errors

• Generally, the assembler will generate a useful error message to help correcting your program

add $1,$1,1
beq $0,$0,loop
mul $1,$2,$3
ldx array,$0,$1
Labels

• Declaration
  - At the beginning of a line
  - Ends with a colon

• Reference
  - Anywhere that an immediate operand is allowed

Example:

```
loop:    addi $1,$1,1
         beq $0,$0,$0,loop
```
Closing the Gap...

- Understand how to program computers at a “high-level”, much closer to a spoken language
- Computers require precise, unambiguous, instructions
- Computers have no context... like people do
- However, we can imagine “higher-level” instructions and “systematic” methods for converting them into “low-level” assembly instructions
Accessing Array Variables

• What we want:
  - The vector of related variables referenced via numeric subscripts rather than distinct names

• Examples:

```
"High-Level Language"

int a[5];
main() {
    int i = 3;
    a[i] = 2;
}

"MIPS Assembly"

main:   ldx   $2,$0,i
        addi  $3,$0,2
        stx   $3,$2,a
halt:   beq   $0,$0,$0,halt

a:       .space 5
i:       .data 3
```

Allocates space for a 5 uninitialized integers
Accessing a “Data Structure”

• What we want:
  - Data structures are another aggregate variable type, where elements have “names” rather than indices

• Examples:

```
“High-Level Language”

struct point {
    int x, y;
} p;

main() {
    p.x = 3;
    p.y = 2;
}
```

```
“Assembly”

main:   addi   $1,$0,p
addi   $2,$0,3
stx    $2,$1,0
addi   $2,$0,2
stx    $2,$1,1
halt:   beq    $0,$0,$0,halt

Allocate space for 2 uninitialized integers (8-bytes)
```

p: .space 2
Conditionals

High-Level

if (expr) {
    STUFF
}

Assembly:

(compute $expr$ in $rx$
beq $0,$rx,$0,Lendif
(compute STUFF)
Lendif:

High-Level

if (expr) {
    STUFF1
} else {
    STUFF2
}

Assembly:

(compute $expr$ in $rx$
beq $0,$rx,$0,Lelse
(compute STUFF1)
beq $0,$0,$0,Lendif
Lelse:

(compute STUFF2)
Lendif:

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```plaintext
if (y < 32) {
    x = x + 1;
}
```

becomes:

```plaintext
ldx $2,$0,y
sgei $1,$2,32
bne $0,$1,$0,Lendif
ldx $2,$0,x
addi $2,$2,1
stx $2,$0,x
Lendif:
```
Loops

Computers spend a lot of time executing loops. Generally loops come in 3 flavors:
- do something “while” a statement is true
- do something “until” a statement becomes true
- repeat something a prescribed number of times
FOR Loops

- Most high-level languages provide loop constructs that establish and update an iteration variable that controls the loop's behavior.

**High-Level code:**

```c
int sum = 0;
int a[10] = {1,2,3,4,5,6,7,8,9,10};

int i;
for (i=0; i<10; i=i+1) {
    sum = sum + a[i];
}
```

**Assembly:**

```
add $3,$0,$0       # i=0
Lfor:  ldx $2,$0,sum
       ldx $1,$3,a
       add $2,$2,$1
       stx $2,$0,sum
       add $3,$3,1       # i=i+1
       sgei $2,$3,10
       beq $0,$2,$0,Lfor
Lendfor:

sum:      .data 0x0
a:         .data 1,2,3,4,5
            .data 6,7,8,9,10
```
Procedures

• Procedures or “subroutines” are reusable code fragments, that are “called”, executed, and then return back from where they were called from.

```assembly
beq  $15,$0,$0,routine
add  $1,$1,$3
```
Procedure Body

• The “Callee” executes its instructions and then “returns” back to the “Caller”
• Uses the jump register (jr) instruction

routine:
  add   $2,$0,$0
  addi  $3,$0,1

loop:
  sge    $4,$1,$3
  beq    $0,$4,$0,return
  sub    $1,$1,$3
  addi   $2,$2,1
  addi   $3,$3,2
  beq    $0,$0,$0,loop

return:
  jr     $0,$15
Parameters

- Most interesting functions have parameters that are “passed” to them by the caller
- Examples \text{Mult}(x, y), \text{Sqrt}(x)
- Caller and Callee must agree on a way to pass parameters and return results. Usually this is done by a convention
- For example, we could pass parameters in sequential registers (\$1, \$2, \$3, etc.) and a single returned value in the next available register.

\begin{verbatim}
ldx \$1,\$0,a
beq \$15,\$0,\$0,routine
ldx \$1,\$0,x
addi \$1,\$1,\$2
\end{verbatim}