Instruction Set Architectures: Talking to the Machine
The Architecture Question

• How do we build computer from contemporary silicon device technology that executes general-purpose programs quickly, efficiently, and at reasonable cost?

• i.e. How do we build the computer on your desk.
In the beginning...

The Difference Engine

- Physical configuration specifies the computation

ENIAC
The Stored Program Computer

- The program is data
  - i.e., it is a sequence of *numbers* that machine interprets
- A very elegant idea
  - The same technologies can store and manipulate programs and data
- Programs can manipulate programs.
The Stored Program Computer

• A very simple model
• Several questions
  • How are program represented?
  • How do we get algorithms out of our brains and into that representation?
  • How does the computer interpret a program?
Representing Programs

• We need some basic building blocks -- call them “instructions”

• What does “execute a program” mean?

• What instructions do we need?

• What should instructions look like?

• Is it enough to just specify the instructions?

• How complex should an instruction be?
Program Execution

- This is the algorithm for a stored-program computer
- The Program Counter (PC) is the key

1. **Instruction Fetch**
   - Read instruction from program storage (mem[PC])

2. **Instruction Decode**
   - Determine required actions and instruction size

3. **Operand Fetch**
   - Locate and obtain operand data

4. **Execute**
   - Compute result value

5. **Result Store**
   - Deposit results in storage for later use

6. **Next Instruction**
   - Determine successor instruction (i.e. compute next PC). Usually this mean PC = PC + <instruction size in bytes>
Motivating Code segments

- $a = b + c$
- $a = b + c + d$
- $a = b \& c$
- $a = b + 4$
- $a = b - (c \times (d/2) - 4)$
- if (a) $b = c$
- if (a == 4) $b = c$
- while (a != 0) $a--$
- $a = 0x{\text{DEADBEEF}}$
- $a = \text{foo}[4]$
- $\text{foo}[4] = a$
- $a = \text{foo}.bar$
- $a = a + b + c + d +... +z$
- $a = \text{foo}(b)$; -- next class
What instructions do we need?

• Basic operations are a good choice.
  • Motivated by the programs people write.
  • Math: Add, subtract, multiply, bit-wise operations
  • Control: branches, jumps, and function calls.
  • Data access: Load and store.

• The exact set of operations depends on many, many things
  • Application domain, hardware trade-offs, performance, power, complexity requirements.
  • You will see these trade-offs first hand in the ISA project and in 141L.
What should instructions look like?

• They will be numbers -- i.e., strings of bits
• It is easiest if they are all the same size, say 32 bits
  • We can break up these bits into “fields” -- like members in a class or struct.
• This sets some limits
  • On the number of different instructions we can have
  • On the range of values any field of the instruction can specify
Is specifying the instructions sufficient?

• No! We also must what the instructions operate on.
• This is called the “Architectural State” of the machine.
  • Registers -- a few named data values that instructions can operate on
  • Memory -- a much larger array of bytes that is available for storing values.
• How big is memory? 32 bits or 64 bits of addressing.
  • 64 is the standard today for desktops and larger.
  • 32 for phones and PDAs
  • Possibly fewer for embedded processors

• We also need to specify semantics of function calls
  • The “Stack Discipline,” “Calling convention,” or “Application binary interface (ABI)”.
How complex should instructions be?

• More complexity
  • More different instruction types are required.
  • Increased design and verification costs
  • More complex hardware.
  • More difficult to use -- What’s the right instruction in this context?

• Less complexity
  • Programs will require more instructions -- poor code density
  • Programs can be more difficult for humans to understand
  • In the limit, decrement-and-branch-if-negative is sufficient
    • Imagine trying to decipher programs written using just one instruction.
    • It takes many, many of these instructions to emulate simple operations.

• Today, what matters most is the compiler
  • The Machine must be able to understand program
  • A program must be able to decide which instructions to use
Big “A” Architecture

- The Architecture is a contract between the hardware and the software.
  - The hardware defines a set of operations, their semantics, and rules for their use.
  - The software agrees to follow these rules.
  - The hardware can implement those rules \textit{IN ANY WAY IT CHOOSES!}
    - Directly in hardware
    - Via a software layer
    - Via a trained monkey with a pen and paper.
- This is a classic interface -- they are everywhere in computer science.
- “Interface,” “Separation of concerns,” “API,” “Standard,”
- For your project you are designing an Architecture -- not a processor.
The Perils of a Standard

- Binary compatibility
  - Read the section on x86 assembly.
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - **Processor state:** PC, “operand stack”, “Base ptr”
  - **Push** -- Put something from memory onto the stack
  - **Pop** -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

```
+16
+12
+8
+4

Memory

X
Y
B
C
A

Base ptr (BP)

0x1000

PC

SP

Memory

X
Y
B
C
A

SP

PC

Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop

```

compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+,-,\times,...$ -- Replace top two values with the result

```
Push 8 (BP)
Push 12 (BP)
Mult
Push 0 (BP)
Push 4 (BP)
Mult
Sub
Store 16 (BP)
Pop
```
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $\pm, \times, \ldots$ -- Replace top two values with the result

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

Base ptr (BP) 0x1000

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>$+4$</td>
</tr>
<tr>
<td>Y</td>
<td>$+8$</td>
</tr>
<tr>
<td>B</td>
<td>$+12$</td>
</tr>
<tr>
<td>C</td>
<td>$+16$</td>
</tr>
<tr>
<td>A</td>
<td>$\cdot$</td>
</tr>
</tbody>
</table>

$B \times C$
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

![Diagram](image.png)
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - +, -, *,… -- Replace top two values with the result
  - Store -- Store the top of the stack

Push 8 (BP)
Push 12 (BP)
Mult
Push 0 (BP)
Push 4 (BP)
Mult
Sub
Store 16 (BP)
Pop

Base ptr (BP) 0x1000

Memory

- X
- Y
- B
- C
- A

Memory Base ptr (BP) SP

PC
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, \times, \ldots$ -- Replace top two values with the result

Push 8 (BP)
Push 12 (BP)
Mult
Push 0 (BP)
Push 4 (BP)
Mult
Sub
Store 16 (BP)
Pop

Base ptr (BP) 0x1000

Memory

- X
- Y
- B
- C
- A

PC

Stack:

- X*Y
- B*C
- $\cdot$
- $\cdot$

SP

$0x1000$
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “Base ptr”
  - Push -- Put something from memory onto the stack
  - Pop -- take something off the top of the stack
  - $+, -, *, ...$ -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 8(BP)
Push 12(BP)
Mult
Push 0(BP)
Push 4(BP)
Mult
Sub
Store 16(BP)
Pop
```

```
From Brain to Bits

Your brain

Brain/ Fingers/ SWE

Programming Language (C, C++, Java)

Compiler

Assembly language

Assembler

Machine code (i.e., .o files)

Linker

Executable (i.e., .exe files)
C Code

int i;
int sum = 0;
int j = 4;
for(i = 0; i < 10; i++) {
    sum = i * j + sum;
}
In the Compiler

```
Function

decl: i

decl: sum = 0

decl: j = 4

Loop

init: i = 0

test: i < 10

inc: i++

Body

statement: =

lhs: sum

rhs: expr

+  

sum

*  

j

i
```
In the Compiler

Control flow graph w/high-level instructions

sum = 0
j = 4
i = 0

i < 10?
false
true

t1 = i * j
sum = sum + t1
i++;

...

Control flow graph w/real instructions

addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0

addi $t0, $zero, 10
bge $s2, $t0

true
false

add $s0, $t0
addi $s2, $s2, 1

...
addi $s0, $zero, 0
addi $s1, $zero, 4
addi $s2, $zero, 0

addi $t0, $zero, 10
bge $s2, $t0, after

body:
mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
br top

after:
...

Out of the Compiler

Assembly language
Labels in the Assembler

0x00  addi $s0, $zero, 0
0x04  addi $s1, $zero, 4
0x08  addi $s2, $zero, 0

top:  
0x0C  addi $t0, $zero, 10
0x10  bge $s2, $t0, after

  mult $t0, $s1, $s2
0x14  add $s0, $t0
0x18  addi $s2, $s2, 1
0x1C  br  top

0x20  after:

‘after’ is defined at 0x20
used at 0x10
The value of the immediate for the branch
is 0x20-0x10 = 0x10

‘top’ is defined at 0x0C
used at 0x1C
The value of the immediate for the branch
is 0x0C-0x1C = 0xFFFF0 (i.e., -0x10)
Assembly Language

• “Text section”
  • Hold assembly language instructions
  • In practice, there can be many of these.

• “Data section”
  • Contain definitions for static data.
  • It can contain labels as well.

• The addresses in the data section have no relation to the addresses in the data section.

• Pseudo instructions
  • Convenient shorthand for longer instruction sequences.
.data
foo_a:
    .word 0

.text
foo:
    lda $t0, foo_a
    ld  $s0, 0($t0)
    addi $s0, $s0, 1
    st  $s0, 0($t0)

after:
    0x18...

lda $t0, foo_a
becomes these instructions (this is not assembly language!)
andi $t0, $zero, ((foo_a & 0xff00) >> 16)
sll  $t0, $t0, 16
andi $t0, $t0, (foo_a & 0xff)

If foo is address 0x0, where is after?
The assembler computes and inserts these values.