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...

- Physical configuration specifies the computation

The Difference Engine

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.
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

Read instruction from program storage (mem[PC])

Determine required actions and instruction size

Locate and obtain operand data

Compute result value

Deposit results in storage for later use

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 $(a != 0)$ $a--$
- $a = 0xDEADBEEF$
- $a = \text{foo}[4]$
- $\text{foo}[4] = a$
- $a = \text{foo}.\text{bar}$
- $a = a + b + c + d +...+z$
- $a = \text{foo}(b)$; -- 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 *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.
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)
int i;
int sum = 0;
int j = 4;
for(i = 0; i < 10; i++) {
    sum = i * j + sum;
}
In the Compiler

```plaintext
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

Control flow graph w/real instructions

```csharp
sum = 0
j = 4
i = 0

// High-level instructions

i < 10?
false
true

// Real instructions

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

mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1
```

...
Out of the Compiler

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

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

true
false

mult $t0, $s1, $s2
add $s0, $t0
addi $s2, $s2, 1

...
Labels in the Assembler

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

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

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

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 = 0xFFFFF0 (i.e., -0x10)
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

  after:
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 and pseudo instructions

void foo() {
    static int a = 0;
    a++;
    ... 
}

.data
foo_a:
    .word 0

.text
foo:
    lda $t0, foo_a
    ld $s0, 0($t0)
    addi $s0, $s0, 1
    st $s0, 0($t0)
    after:
    ...
.data and pseudo instructions

void foo() {
    static int a = 0;
    a++;
    ...
}

. data
foo_a:
    .word 0

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

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)
.data and pseudo instructions

```c
void foo() {
    static int a = 0;
    a++;
    ...
}
```

The assembler computes and inserts these values.
.data and pseudo instructions

```c
void foo() {
    static int a = 0;
    a++;
    ...
}
```

The assembler computes and inserts these values.

```
lda $t0, foo_a
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`?

ld $s0, 0($t0)
addi $s0, $s0, 1
st $s0, 0($t0)

...
.data and pseudo instructions

```c
void foo() {
    static int a = 0;
    a++;
    ...
}
```

```
.data
foo_a:
    .word 0

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

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

If `foo` is address 0x0, where is `after`?

The assembler computes and inserts these values.
ISA Alternatives

• MIPS is a 3-address, RISC ISA
  • add rs, rt, rd -- 3 operands
  • RISC -- reduced instruction set. Relatively small number of operation. Very regular encoding. RISC is the “right” way to build ISAs.

• 2-address
  • add r1, r2 --> r1 = r1 + r2
  • + few operands, so more bits for each.
  • - lots of extra copy instructions

• 1-address
  • Accumulator architectures
  • add r1 --> acc = acc + r1
Stack-based ISA

• A push-down stack holds arguments
• Some instruction manipulate the stack
  • push, pop, swap, etc.
• Most instructions operate on the contents of the stack
  • Zero-operand instructions
  • add --> t1 = pop; t2 = pop; push t1 + t2;
• Elegant in theory.
• Clumsy in hardware.
  • How big is the stack?
• Java byte code is a stack-based ISA
• So is the x86 floating point ISA
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

```
PC
```

```
Base ptr (BP)
0x1000
```

```
Memory

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

SP
```
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

Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>B</td>
</tr>
<tr>
<td>+4</td>
<td>+8</td>
<td>+12</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>+12</td>
<td>+16</td>
<td></td>
</tr>
</tbody>
</table>

Base ptr (BP)

0x1000
compute \( A = X \ast Y - B \ast C \)

- **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
  - \(+, -, *,\ldots\) -- Replace top two values with the result
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
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

---

**Base ptr (BP)**

```
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
  - $+, -, \times, \ldots$ -- 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
```

```
Memory
+8
B
+12
C
+16
A
```

```
Base ptr (BP)
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
  - $+, -, \ast, \ldots$ -- Replace top two values with the result

```
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
  - $+, -, \times, \ldots$ -- Replace top two values with the result

![Diagram of stack operations]

- 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
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

![Stack-based ISA Diagram]

Memory

Base ptr (BP)

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

```
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

![Diagram showing stack operations and memory access](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

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

```plaintext
0x1000
```

```
Memory

X
Y
B
C
A

SP
+4
+8
+12
+16

Base ptr (BP)
```
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
  - Store -- Store the top of the stack

```
X\times Y
B\times C
```

```
Memory
+4  X
+8  Y
+12 B
+16 C
```

- Base ptr (BP) 0x1000

- 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
  - +, -, *, ... -- 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
```
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
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

```
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**

---

**SP**
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
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
Supporting Function Calls

• Functions are an essential feature of modern languages

• What does a function need?
  • Arguments.
  • Storage for local variables.
  • To return control to the caller.
  • To execute regardless of who called it.
  • To call other functions (that call other functions...that call other functions)

• There are not *instructions* for this
  • It is a contract about how the function behaves
  • In particular, how it treats the resources that are shared between functions -- the registers and memory
Register Discipline

- All registers are the same, but we assign them different uses.

<table>
<thead>
<tr>
<th>Name</th>
<th>number</th>
<th>use</th>
<th>saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$7</td>
<td>26-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>26</td>
<td>global ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Arguments

• How many arguments can function have?
  • unbounded.
  • But most functions have just a few.

• Make the common case fast
  • Put the first 4 argument in registers ($a0$-$a3$).
  • Put the rest on the “stack”

int Foo(int a, int b, int c, int d, int e) {
  ...
}

...
Storage for Local Variables

- Local variables go on the stack too.

```c
int Foo(int a, int b, int c, int d, int e) {
    int bar[4];
    ...
}
```
Returning Control

**Caller**

...  
move $a0, $t1  
move $a1, $s4  
move $a2, $s3  
move $a3, $s3  
sw $t2, 0($sp)  
subi $sp, $sp, 4  
0xBAD0: jal Foo

**Callee**

```c
int Foo(int a, ...) {
    int bar[4];
    ...
    return bar[0];
}
```

```
subi $sp, $sp, 16 // Allocate bar
...
lw $v0, 0($sp)
addi $sp, $sp, 16 // deallocate bar
jr $ra // return
```
Saving Registers

• Some registers are preserved across function calls
  • If a function needs a value after the call, it uses one of these
  • But it must also preserve the previous contents (so it can honor its obligation to its caller)
  • Push these registers onto the stack.
  • See figure 2.12 in the text.