Instruction Set
Architectures: Talking to the Machine
The Next Two Weeks

• Two Goals
• Prepare you for your 141 Project
  • Understand what an ISA is and what it must do.
  • Understand the design questions they raise
  • Begin to think about what makes a good ISA vs a bad one
  • See an example of designing an ISA.
• Learn to “see past your code” to the ISA
  • Be able to look at a piece of C (or Java) code and know what kinds of instructions it will produce.
  • Understand (or begin to) the compiler’s role
  • Be able to roughly estimate the performance of code based on this understanding (we will refine this skill throughout the quarter.)
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?
- What data will the instructions operate on?
- How complex should an instruction be?
- How do functions work?
Program Execution

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

1. Instruction Fetch
2. Instruction Decode
3. Operand Fetch
4. Instruction Execute
5. Compute Result
6. Store Result
7. Compute Next PC

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 means PC = PC + <instruction size in bytes>
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 141 project you are designing an Architecture -- not a processor.
  • (in 141L, you will design a processor)
What instructions do we need?

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

• The exact set of operations depends on many things
  • Application domain, hardware trade-offs, performance, power, complexity requirements.
  • You may need some misc instructions to handle system specific-issues - e.g. manage cache/vm/io etc.
  • You will see these trade-offs first hand in the ISA project and in 141L.

• Two philosophies:
  • Stanford MIPS-X 3% rule
  • Intel’s SSEn as N → ∞
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 data will instructions operate on?

- 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
  - What about 48?
How do instructions access memory?

- In modern ISAs, every byte (8b) has an address.
- CISC (complex instr set computer):
  - Anything you like; but the underlying hw will turn it into RISC ops!
- RISC (reduced instr set computer):
  - Arithmetic instrs just operate on registers
  - Memory instrs access memory
    - Load -- move a piece of data from memory into a register
    - Store -- move the contents of a register into memory.
## Bytes and Words

### Byte addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td>...</td>
<td>..</td>
</tr>
<tr>
<td>0xFFF</td>
<td>..</td>
</tr>
<tr>
<td>0xFFFF</td>
<td>..</td>
</tr>
</tbody>
</table>

### Word Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>

Modern machines use “byte addressable” memories.
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
  - Given the address of an instruction, it will be easy to find the “next” one.
- They will have internal structure
  - Subsets of bits represent different aspects of the instruction -- which operation to perform. Which data to operate on.
  - A regular structure will make them easier to interpret
  - Most instructions in the ISA should “look” the same.
- 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
How complex should instructions be?

- **More complexity**
  - More different instruction types are required.
  - Increased design and verification costs.
  - More complex hardware; can slow down clock frequency.
  - 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 (i.e., the compiler) must be able to decide which instructions to use

- Each instruction should do about the same amount of work.
How do functions work?

- The “Stack Discipline,” “Calling convention,” or “Application Binary Interface (ABI)”.
  - How to pass arguments
  - How to keep track of function nesting
  - How to manage “the stack”
Motivating Code segments

- $a = b + c$;
- $a = b + c + d$;
- $a = b \& c$;
- $a = b + 4$;
- $a = b - (c \ast (d/2) - 4)$;
- if (a) $b = c$;
- if (a == 4) $b = c$;
- while (a != 0) a--;
- $a = 0xDEADBEEF$;
- $a = \text{foo}[4]$;
- $\text{foo}[4] = a$;
- $a = \text{foo}.bar$;
- $a = a + b + c + d + \ldots + z$;
- $a = \text{foo}(b)$; -- next class

- What instructions do we need?
- What should instructions look like?
- What data will the instructions operate on?
- How complex should an instruction be?

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises
On Deck

• Finish up the ISA design example
  • Memory
  • Large constants
  • Functions
• Questions about project.
• x86 assembly overview.
Accessing Memory

- In your ISA, an instruction should do at most one memory op (e.g. a load or store). Doing more adds a lot of timing complexity.
- Loads in MIPS
  - lw r3, offset(r2) -> R[rt] = mem[R[rs] + imm]
- Stores in MIPS
  - sw r3, offset(r2) -> mem[R[rs] + imm] = R[rt]
- Does it makes sense that rt is an input to sw and an output of lw?
Large Constants

• Some constants are just as big as the instruction - no room for opcode!
  • example: Create 0xDEADBEEF -- 32 bit values
• MIPS -- 16 bit immediate
  • add $2, zero, 0xDEAD
  • sll $2, $2, 16
  • ori $2, $2, 0xBEEF
  • Alternative:
    • Assembly: LoadConst r1, 0xDEADBEEF
    • RTL: \[ R[r1] = \text{mem}[\text{PC}+4]; \text{PC} = \text{PC} + 8. \]
    • Good idea? See next slide.
Uniformity and Compiler Friendliness in MIPS

• 3 instruction formats: I, R, and J.
  • R-type: Register-register Arithmetic
  • I-type: immediate arithmetic; loads/stores; cond branches
  • J-type: Jumps - Non-conditional, non-relative branches
  • opcodes are always in the same place
  • rs and rt are always in the same place; as is RD if it exists
  • The immediate is always in the same place

• Similar amounts of work per instruction
  • 1 read from instruction memory
  • <= 1 arithmetic operations
  • <= 2 register reads
  • <= 1 register write
  • <= 1 data store/load

• Fixed instruction length
• Relatively large register file: 32
• Reasonably large immediate field: 16 bits

• Wise use of opcode space
  • 6 bits of opcode
  • R-type gets another 6 bits of “function”
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

```c
int Factorial(int x) {
    if (x == 0)
        return 1;
    else
        return x * Factorial(x - 1);
}
```
Register Discipline

- All registers > 0 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>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temp</td>
<td>no</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>$s0-$s7</td>
<td>16-23</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”

```c
int Foo(int a, int b, int c, int d, int e) {
    ...
}
```
Storage for Local Variables

- Local variables go on the stack too.
- $fp$ -- frame pointer (points to base of this frame)
- $sp$ -- stack pointer

```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)
0xBAD0: jal Foo
```

**Callee**

```
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.
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

```
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

sum = 0
j = 4
i = 0

i < 10?
false
true

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

...
Out of the Compiler

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

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

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

...
Labels in the Assembler

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

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

addi $s2, $zero, 1
br top

‘after’ is defined at 0x20
used at 0x10
The value of the immediate for the branch is $0x20 - (0x10 + 0x04) = 0x0C

‘top’ is defined at 0x0C
used at 0x1C
The value of the immediate for the branch is $0x0C-(0x1C + 0x04) = 0xFFEC (i.e., -0x14)
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 text section have no relation to the addresses in the data section.
- Pseudo instructions
  - Convenient shorthand for longer instruction sequences.
.data and pseudo instructions

```c
int a = 0;

void foo() {
    a++;
    ...
}
```

```asm
.lw $3, a
.addiu $3, $3, 1
.sw $3, a
.after:
.addiu $2, $2, 1
... 
.bne $2, after
```

.becomes these instructions:

```asm
.lui $at, %hi(a)
.addu $at, $gp, $at
.lw $3, %lo(a)($at)
```

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

- MIPS is a 3-address, RISC ISA
  - add rd, rs, rt -- 3 operands
  - RISC -- reduced instruction set. Relatively small number of operation. Very regular encoding. RISC is the “right” way to build ISAs if instr storage is cheap and you have a lot of encoding space (unlike for your 141 ISA!)

- 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 mem → acc = acc + mem

- Implicit Registers
  - e.g.; different instruction types assume particular output regs
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

```plaintext
Push 12(BP)
Push 8(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
compute $A = X \times Y - B \times C$

- **Stack-based ISA**
  - Processor state: PC, “operand stack”, “global 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 12($gp)
Push 8($gp)
Mult
Push 0($gp)
Push 4($gp)
Mult
Sub
Store 16($gp)
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
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 12(BP)
PUSH 8(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

- **Example Instructions**
  - Push 12(BP)
  - Push 8(BP)
  - Mult
  - Push 0(BP)
  - Push 4(BP)
  - Mult
  - Sub
  - Store 16(BP)
  - Pop

- **Diagram**
  - Memory
    - X
    - Y
    - B
    - C
    - A
  - Base ptr (BP) 0x1000
  - 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
  - $+, -, *, ...$ -- Replace top two values with the result

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

```
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>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+4</td>
<td>X</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+8</td>
<td>Y</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>+16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Base ptr (BP) 0x1000

PC

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
  - +, -, *,... -- Replace top two values with the result
  - Store -- Store the top of the stack

```
Push 12(BP)
Push 8(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 12(BP)
Push 8(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
  - Store -- Store the top of the stack

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

Memory

```
X
Y
B
C
A
```

Base ptr (BP) 0x1000
Time-based Addressing

- Named registers are sooo last century.
- Instead, store the register file as a shift register
- Refer to temporaries by how many instructions ago they were created
- Assume there’s a special stack pointer, sp.

\[
\text{sum} = a + b + c;
\]

```
read sp
lw  0(v0)
lw  4(v1)
lw  8(v2)
add v0, v1
add v0, v3
```

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>b</td>
</tr>
<tr>
<td>v2</td>
<td>c</td>
</tr>
<tr>
<td>v1</td>
<td>b+c</td>
</tr>
<tr>
<td>v0</td>
<td>a+b+c</td>
</tr>
</tbody>
</table>

From One Core to Multi-Core to Many-core

Intel P4  
1 core

Intel Core 2 Duo  
2 cores

Intel Nehalem  
4 cores

SPARC T1  
8 cores

Cell BE  
8 + 1 cores

MIT Raw  
16 cores

Tilera  
64 cores