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

Two Goals

- Prepare you for 141L Project: Lab 2
  - Your own processor for Bitcoin mining!
  - Understand what an ISA is and what it must do.
  - Understand the design questions they raise
  - 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 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

ENIAC

Physical configuration specifies the computation
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?
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. **Instruction Execute**
   - Compute result value
5. **Store Result**
   - Deposit results in storage for later use
6. **Compute Next PC**
   - Determine successor instruction (i.e. compute next PC). Usually this mean PC = PC + <instruction size in bytes>
Instruction Set Architecture (ISA)

- 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 conform to the interface.
  - The hardware agrees to implement the interface.
- But, the hardware can implement its side of the contract IN ANY WAY IT CHOOSES!
  - Directly in hardware
  - Via a software layer
  - Via a trained monkey with a pen and paper.
- The ISA is an interface: they are everywhere in computer science.
  - “Interface,” “Separation of concerns,” “API,” “Standard,”
What operations (ops) do we need?

Basic ones are a good start.
- Ones *commonly found in programs*.
- Math: Add, subtract, bit-wise operations, mul/div, FP
- Control: branches, jumps, and function calls.
- Data access: Load and store.
- Miscellaneous: I/O, manage VM, protection, cache, etc.

Beyond the basics.
- Eliminate uncommon ops
  - popcount? parity? evaluate polynomial?
- Eliminate instructions that execute no faster than an equivalent sequence of instructions
- Add *secret sauce* instructions based on application domain, hardware trade-offs, performance, power.

Two philosophies:
- Minimalism, Efficiency: MIPS 3% rule; 3-day waiting period
- Monopolism: Intel SSEx instruction set
Op Hunting

Iterate over a list of typical code sequences; how inefficient is it to handle them with current op?

\[
\begin{align*}
\text{a} &= \text{b} + \text{c} \\
\text{a} &= \text{b} + \text{c} + \text{d} \\
\text{a} &= \text{b} \& \text{c} \\
\text{a} &= \text{b} | \text{c} \\
\text{if} \ (\text{a}) \ {\{} \ {\}} \\
\text{while} \ (\text{a}) \ {\{} \ {\}} \\
\text{do while} \ {\{} \ {\}} \\
\text{a} &= \text{x}[3] \\
\text{x}[3] &= \text{a} \\
\text{a} &= \text{x}[\text{i}] \\
\text{x}[\text{i}] &= (\text{a} > \text{b}) \\
\text{a} &= \text{x}[\text{i}+3] \\
\text{x}[\text{i}+3] &= \text{a} \\
\text{a} &= \text{x}[\text{i} \times 4] \\
\text{x}[\text{i} \times 4] &= \text{a}++ \\
\text{y} &= \text{f}(\text{a}, \text{b}, \text{c}) \\
\text{q} &= 0x\text{DEAD}_{-}\text{BEEF} \\
\text{q} &= \text{p}->\text{next};
\end{align*}
\]
What data will instructions operate on?

Is specifying the ops sufficient?
No! We also must what the ops operate on.

This is the “Architectural State” of the machine.
- Registers: “local memory” a few fast, named data values
- Memory: larger, slower array of bytes; unnamed data 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, now moving to 64.
  - Possibly fewer for embedded microcontrollers
How do instructions access memory?

- **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.

- **CISC (complex instr set computer):**
  - Combine memory and computation operations into a single operation.
  - Sometimes hundreds or thousands of ops:
    - e.g. `REP MOVSB` == `strcpy` on x86
  - Many machines will convert CISC instructions into RISC instructions behind your back. It’s faster.
Bytes and Words

<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>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

<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>0xFFFFC</td>
<td>.</td>
</tr>
</tbody>
</table>

Modern machines use “byte addressable” memories
How will we represent instructions?

- 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
Encodings: MIPS

addu $3, $2, $5
sll $3, $2, 5
addu $3, $2, imm16
beq $3, $4, imm16
Encoding MIPS: Operation

addu $3, $2, $5
sll $3, $2, 5
addu $3, $2, imm16
beq $3, $4, imm16
Encoding MIPS: Src 1

addu $3, $2, $5
sll $3, $2, 5
addu $3, $2, imm16
beq $3, $4, imm16
Encoding MIPS: Src 2

- **addu $3, $2, $5**
- **sll $3, $2, 5**
- **addu $3, $2, imm16**
- **beq $3, $4, imm16**
encoding mips: branch target

addu $3, $2, $5
sll $3, $2, 5

addu $3, $2, imm16

beq $3, $4, imm16
Encoding MIPS: Dest

\[
\begin{align*}
\text{addu} &\quad \text{$3$, $2$, $5$} \\
\text{sll} &\quad \text{$3$, $2$, $5$} \\
\text{addiu} &\quad \text{$rt$, $rs$, imm16} \\
\text{beq} &\quad \text{$3$, $4$, imm16}
\end{align*}
\]
## Encodings: Intel

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four prefixes of 1 byte each (optional)</td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

Instructions are variable length
Description of encoding continues for 20 pages...
Encoding Vanilla ISA (141L)

See manual!

- ALU
  - op: 5 bits
  - rd: 5 bits
  - rs: 6 bits

- Branch
  - op: 5 bits
  - rd: 5 bits
  - offset: 6 bits

Destinations: 32 registers
Sources: 32 registers + 32 constants
### Vanilla: Opcode Map

- Filled with basics that are tough to get by without
- Lots of space to extend (10 opc = 20480/65,536)
- (Bitcoin mining, here we come!)

<table>
<thead>
<tr>
<th>13..11</th>
<th>000</th>
<th>010</th>
<th>011</th>
<th>111</th>
<th>110</th>
<th>100</th>
<th>101</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ADDU</td>
<td>SLLV</td>
<td>SRAV</td>
<td>NOR</td>
<td>OR</td>
<td>SRLV</td>
<td>AND</td>
<td>SUBU</td>
</tr>
<tr>
<td>10</td>
<td>BEQZ</td>
<td>BGTZ</td>
<td>BLTZ</td>
<td>JALR</td>
<td></td>
<td></td>
<td>BNEQZ</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LW</td>
<td>SW</td>
<td></td>
<td>SB</td>
<td></td>
<td>LBU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>SLT</td>
<td>MOV</td>
<td>SPCL</td>
<td></td>
<td></td>
<td>SLTU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vanilla’s Two-Address Instructions

Most instructions have a *destructive* write

To keep the first source, you must use `mov`

But not loads (lw, lb) and stores (sw,sb)

E.g., :

\[
\text{mem}[x] = \text{mem}[x+1] \quad \# x \text{ in } $4
\]

```
mov  $3, $4  \# \$3 = \$4 \; ; \text{avoid squashing } \$4
add $3, %one  \# \$3 = \$3 + 1
lw  $5, $3  \# \$5 = \text{mem}[$3]
sw  $4, $5  \# \text{mem}[$4] = \$5
```
Extending Vanilla

Could we extend Vanilla to optimize this case?

\[ \text{mem}[x] = \text{mem}[x+1] \]

- `mov  $3, $4`
- `addu $3, %one`
- `lw   $5, $3`
- `sw   $4, $5`
Could we extend Vanilla to optimize this case?

Maybe an *implicit register, plus indexed addressing*?

lw2 rd, rs

\[ [2] = \text{mem}[\text{rd} + \text{rs}] \]

\[
\begin{array}{c|c|c}
\text{lw2} & \text{rd} & \text{rs} \\
\end{array}
\]

\[
\begin{array}{cccc}
10101 & 5 & 5 & 6 \\
15 & 11 & 10 & 0 \\
\end{array}
\]

\[
\begin{array}{l}
\text{mem}[x] = \text{mem}[x+1] \\
\text{mov} & $3, $4 \\
\text{addu} & $3, \%one \\
\text{lw} & $5, $3 \\
\text{sw} & $4, $5 \\
\end{array}
\]
Could we extend Vanilla to optimize this case? Maybe an *implicit register, plus indexed addressing?*

\[ \text{lw2 } \text{rd,rs} \]
\[ [2] = \text{mem}[ \text{rd} + \text{rs}] \]

\[ \text{mem}[x] = \text{mem}[x+1] \]

mov \ $3, $4
addu \ $3, %one
lw \ $5, $3
sw \ $4, $5

\[ \text{lw2 } \text{$4, %one} \]
sw \ $4, $2
Extending Vanilla

Could we extend Vanilla to optimize this case?
Maybe an *implicit register, plus indexed addressing?*

```
addu 10101 rd rs

15 11 10 0

mem[x] = mem[x+1]

mov $3, $4
addu $3, %one
lw $5, $3
sw $4, $5
```

```
addu2 rd,rs
[2] = [rd] + [rs]
```

```
addu2 $4, %one
lw $3, $2
sw $4, $3
```
Extending Vanilla

Could we extend Vanilla to optimize this case? Maybe an *implicit register, plus indexed addressing?*

lw2 rd,rs

\[ [2] = \text{mem}[rd + rs] \]

*Encoding space: 2048*
Extending Vanilla

Could we extend Vanilla to optimize this case?
Maybe an *implicit* register, plus *indexed* addressing?

Encoding space: 2048

\[
\begin{array}{c|c|c}
\text{lw2} & \text{rd} & \text{rs} \\
10101 & 5 & 6 \\
15 & 10 & 11 \\
\end{array}
\]

\[ [2] = \text{mem}[ \text{[rd]+[rs]} ] \]

Do we now want \text{sw2 rd,rs}?
\[
\text{mem}[ \text{[rd]+[rs]} ] = [2]
\]

Now we can do \[
\text{mem } [[[\$w] + x]] = \text{mem } [[[\$y]+z]]
\]

Encoding space was 1/4 of free space though
Extending Vanilla

Or add special instruction
- Encoding space use is modest .. as is usefulness!

\[ \text{cp\_bk} \quad \$4 \]

<table>
<thead>
<tr>
<th>spcl</th>
<th>cp_bk</th>
<th>rs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011</td>
<td>00001</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Encoding space:} \quad 64 \]

\[ \text{mem}[\text{rs}] = \text{mem}[\text{rs} + 1] \]
Extending Vanilla

*Implicit register* saves even more encoding space
But makes compiler’s register allocation harder
But it’s cheap at least

\[
\text{copy\_back}
\]

\[
\begin{array}{c|c}
\text{spcl} & \text{copy\_back} \\
10011 & 000_0000_0001 \\
\end{array}
\]

\[
\begin{array}{cccc}
15 & 11 & 10 & 0 \\
\end{array}
\]

\[
\text{mem}[$4$] = \text{mem}[$4+1$]
\]
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, for general purpose machines, compiler is king
- People used to design instruction sets for humans. “Semantic Gap”
- Today the compiler uses the instruction set, not the user.
- Instruction is not useful if compiler can’t figure out when to use it.
- Sometimes, a library interfaces to weird instructions (AES) works
- Or assembly macros in gcc:
  ```
  asm("leal (\%1, \%1, 4), \%0":
       "=r" (five_times_x): "r" (x));
  ```
- 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”
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?
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

\[
\begin{align*}
lui & \$2, 0x\text{DEAD} \\
\text{ori} & \$2, \$2, 0x\text{BEEF}
\end{align*}
\]
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 --> makes pipelining easy
- 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”
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);
}
```
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>
MIPS: 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];
    ...
}
```
MIPS: Returning Control

**Caller**

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

**Callee**

```plaintext
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
```
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.
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

```c
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
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

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

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

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

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

    ‘top’ is defined at 0x0C
    used at 0x1C
    The value of the immediate for the branch is
    0x0C-(0x1C + 0x04) = 0xFFFEC (i.e., -0x14)

0x20  after:
    ...
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++;
    ...
}
```

becomes these instructions:

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

```assembly
.data
a:
    .word 0

.text
foo:
0x00  lw    $3, a
0x0c  addui $3, $3, 1
0x10  sw    $3, a
0x1c  after:
      addui $2, $2, 1
0x20  ...
      bne  $2, after
```

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 instructions 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

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

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

Base ptr (BP) 0x1000
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
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

```
PC       Memory
Push 12(BP)       X
Push 8(BP)       +4
Mult
Push 0(BP)       +8
Push 4(BP)       B
Mult
Sub
Store 16(BP)     +12
Pop
SP

B
C

0x1000
Base ptr (BP)

A
C
+16

+12

+8

+4

X
Y
B
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
  - +, -, *,... -- 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

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

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

```
B\times C
```

```
PC
```

```
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 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
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

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

```
Memory

<table>
<thead>
<tr>
<th>0x1000</th>
<th>Base ptr (BP)</th>
</tr>
</thead>
</table>

| +4  | X |
| +8  | Y |
| +12 | B |
| +16 | C |
|     | A |
```

```
X*Y-B*C
```

```
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
```
History-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>

63
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