Designing an Instruction Set

- Von Neumann Model
- Intro to Beta Instruction Set

Today’s handouts:
- Lecture slides
- Beta handout

Notes:
- Lab 2 due tonight!

http://xkcd.com/722/
Example: Factorial

factorial(N) = N! = N*(N-1)*...*1

C:
```c
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
}while (b != 0)
```

High-level FSM:
```
  a ← 1  a ← a * b  a ← a
  b ← N  b ← b - 1  b ← b
```
Datapath and Control FSM for Factorial

- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition logic outputs
  - Outputs: Mux select signals

\[
\begin{align*}
\text{start} & \quad 0 \\
\text{loop} & \quad 1 \\
\text{done} & \quad 2 \\
& \\
\text{a} & \leftarrow 1 \\
\text{b} & \leftarrow N \\
& \\
\text{a} & \leftarrow a \cdot b \\
\text{b} & \leftarrow b - 1
\end{align*}
\]
The von Neumann Model

- Many approaches to build a general-purpose computer. Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)

- Components:

  - Central processing unit:
    - Performs operations on values in registers & memory
  - Main memory:
    - Array of W words of N bits each
  - Input/output devices to communicate with the outside world
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

![Diagram of Central Processing Unit and Main Memory with instructions and data]

- **Central Processing Unit**
- **Main Memory**
  - Instructions
  - Data

**But, how do we know which words hold instructions and which words hold data?**

Example:

```
rc ← op(ra, rb)
```

```
0xba5eba11
```
Anatomy of a von Neumann Computer

- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath
Instructions

• Instructions are the fundamental unit of work
• Each instruction specifies:
  – An operation or opcode to be performed
  – Source operands and destination for the result

• In a von Neumann machine, instructions are executed sequentially
  – CPU logically implements this loop:
  – By default, the next PC is current PC + size of current instruction unless the instruction says otherwise
Instruction Set Architecture (ISA)

• ISA: The contract between software and hardware
  – Functional definition of operations and storage locations
  – Precise description of how software can invoke and access them

• The ISA is a new layer of abstraction:
  – ISA specifies what the hardware provides, not how it’s implemented
  – Hides the complexity of CPU implementation
  – Enables fast innovation in hardware (no need to change software!)
    • 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
    • Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
    • Quad-core Skylake K (2015): 1.75 billion transistors, 4 GHz, ~30k MIPS
  – Dark side: Commercially successful ISAs last for decades
    • Today’s x86 CPUs carry baggage of design decisions from the 70’s
Instruction Set Architecture Design

• Designing an ISA is hard:
  – How many operations?
  – What types of storage, how much?
  – How to encode instructions?
  – How to future-proof?

• How to decide? Take a quantitative approach
  – Take a set of representative benchmark programs
  – Evaluate versions of your ISA and implementation with and without feature
  – Pick what works best overall (performance, energy, area…)

• Corollary: Optimize the common case

Let’s design our own instruction set: the Beta!
Beta ISA: Storage

CPU State

Main Memory

Up to $2^{32}$ bytes (4GB of memory) organized as $2^{30}$ 4-byte words

Each memory word is 32-bits wide, but for historical reasons the $\beta$ uses byte memory addresses. Since each word contains four 8-bit bytes, addresses of consecutive words differ by 4.

Why separate registers and main memory?

Tradeoff: Size vs speed and energy

r31 hardwired to 0
Storage Conventions

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
  - Load them
  - Compute on them
  - Store the results

```c
int x, y;
y = x * 37;
```

```
R0 ← Mem[0x1008]
R0 ← R0 * 37
Mem[0x100C] ← R0
```
Beta ISA: Instructions

• Three types of instructions:
  – Arithmetic and logical: Perform operations on general registers
  – Loads and stores: Move data between general registers and main memory
  – Branches: Conditionally change the program counter

• All instructions have a fixed length: 32 bits (4 bytes)
  – Tradeoff (vs variable-length instructions):
    • Simpler decoding logic, next PC is easy to compute
    • Larger code size
Beta ALU Instructions

Format: 

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r_c</th>
<th>r_a</th>
<th>r_b</th>
<th>unused</th>
</tr>
</thead>
</table>

Example coded instruction: ADD

32-bit hex: 0x80611000

We prefer to write a **symbolic representation**: ADD(r1,r2,r3)

ADD(ra,rb,rc):

Reg[rc] ← Reg[ra] + Reg[rb]

“Add the contents of ra to the contents of rb; store the result in rc”

Similar instructions for other ALU operations:

- arithmetic: ADD, SUB, MUL, DIV
- compare: CMPEQ, CMPLT, CMPLE
- boolean: AND, OR, XOR, XNOR
- shift: SHL, SHR, SAR
Implementation Sketch #1

Now that we have our first set of instructions, we can create a more concrete implementation sketch:

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r_c</th>
<th>r_a</th>
<th>r_b</th>
<th>unused</th>
</tr>
</thead>
</table>

32 registers

ALU

operations

PC

4

+
Should We Support Constant Operands?

Many programs use small constants frequently

**Tradeoff:**

- When used, they save registers and instructions
- More opcodes $\rightarrow$ more complex control logic and datapath

Analyzing operands when running SPEC CPU benchmarks, we find that constant operands appear in

- $>50\%$ of executed arithmetic instructions
  - *Loop increments, scaling indices*
- $>80\%$ of executed compare instructions
  - *Loop termination condition*
- $>25\%$ of executed load instructions
  - *Offsets into data structures*
Beta ALU Instructions with Constant

Format:

\[
\begin{array}{cccc}
\text{OPCODE} & r_c & r_a & 16\text{-bit signed constant} \\
\end{array}
\]

Example instruction: ADDC adds register contents and constant:

Symbolic version: ADDC(r1,-3,r3)

ADDCC(r_a, const, r_c):

\[\text{Reg}[r_c] \leftarrow \text{Reg}[r_a] + \text{sext(const)}\]

“Add the contents of \(r_a\) to \(const\); store the result in \(r_c\)”

Similar instructions for other ALU operations:

- arithmetic: ADDC, SUBC, MULC, DIVC
- compare: CMPEQC, CMPLTC, CMPLEC
- boolean: ANDC, ORC, XORC, XNORC
- shift: SHLC, SHRC, SARC
Next we add the datapath hardware to support small constants as the second ALU operand:

Next we add the datapath hardware to support small constants as the second ALU operand:
Beta Load and Store Instructions

Loads and stores move data between the internal registers and main memory

\[
\text{LD}(ra, \text{const}, rc) \quad \text{Reg}[rc] \leftarrow \text{Mem}[\text{Reg}[ra] + \text{sext}(\text{const})]
\]

Load \(rc\) with the contents of the memory location

\[
\text{ST}(rc, \text{const}, ra) \quad \text{Mem}[\text{Reg}[ra] + \text{sext}(\text{const})] \leftarrow \text{Reg}[rc]
\]

Store the contents of \(rc\) into the memory location

To access memory the CPU has to generate an address. LD and ST compute the address by adding the sign-extended constant to the contents of register \(ra\).

- To access a constant address, specify R31 as \(ra\).
- To use only a register value as the address, specify a constant of 0.

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>(rc)</th>
<th>(ra)</th>
<th>16-bit signed constant</th>
</tr>
</thead>
</table>

Address calculation is just like ADDC instruction!
Using LD and ST

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
  - Load them
  - Compute on them
  - Store the results

```c
int x, y;
y = x * 37;
```

```
R0 ← Mem[0x1008]
R0 ← R0 * 37
Mem[0x100C] ← R0
```

```
LD(R31,0x1008,R0)
MULC(R0,37,R0)
ST(R0,0x100C,R31)
```
Can We Solve Factorial With ALU Instructions?

- \( \text{factorial}(N) = N! = N \times (N-1) \times \ldots \times 1 \)

- No! Factorial needs to loop

- So far we can only encode sequences of operations on registers

- Need a way to change the PC based on data values!
  - Called “branching”. If the branch is taken, the PC is changed. If the branch is not taken, keep executing sequentially.
Beta Branch Instructions

The Beta’s *branch instructions* provide a way to conditionally change the PC to point to a nearby location...

... and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

---

**BEQ or BNE**

<table>
<thead>
<tr>
<th>r_c</th>
<th>r_a</th>
<th>16-bit signed constant</th>
</tr>
</thead>
</table>

**BEQ(ra,offset,rc):** Branch if equal

NPC ← PC + 4
Reg[rc] ← NPC
if (Reg[ra] == 0)
    PC ← NPC + 4*offset
else
    PC ← NPC

**BNE(ra,offset,rc):** Branch if not equal

NPC ← PC + 4
Reg[rc] ← NPC
if (Reg[ra] != 0)
    PC ← NPC + 4*offset
else
    PC ← NPC

“offset” is a SIGNED CONSTANT encoded as part of the instruction!

offset = distance in words to branch target, counting from the instruction following the BEQ/BNE. Range: -32768 to +32767.
Can We Solve Factorial Now?

- Yes!

```c
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)
```

// Assume r1 = N

```
ADD(r31, 1, r0)  // r0 = 1
L:MUL(r0, r1, r0)  // r0 = r0 * r1
SUBC(r1, 1, r1)  // r1 = r1 - 1
BNE(r1, L, r31)  // if r1 != 0, run MUL next
// at this point, r0 = N!
```

![Diagram](6.004 Computation Structures)
Beta JMP Instruction

Branches transfer control to some predetermined destination specified by a constant in the instruction. It will be useful to be able to transfer control to a computed address.

\[ 011011 \quad r_c \quad r_a \quad unused \]

\[ \text{JMP}(Ra,Rc): \quad \text{Reg}[Rc] \leftarrow \text{PC} + 4 \]
\[ \text{PC} \leftarrow \text{Reg}[Ra] \]

Useful for procedure call return...

\[ \text{[0x100]} \quad \text{BEQ}(R31, \text{sqrt}, R28) \]
\[ \text{[0x678]} \quad \text{BEQ}(R31, \text{sqrt}, R28) \]

1\text{st} time: PC\leftarrow0x104
2\text{nd} time: PC\leftarrow0x67C
Beta ISA Summary

- **Storage:**
  - Processor: 32 registers (r31 hardwired to 0) and PC
  - Main memory: 32-bit byte addresses; each memory access involves a 32-bit word. Since there are 4 bytes/word, all addresses will be a multiple of 4.

- **Instruction formats:**
  - ALU: Two input registers, or register and constant
  - Loads and stores
  - Branches, Jumps

- **Instruction types:**
  - ALU: Two input registers, or register and constant
  - Loads and stores
  - Branches, Jumps

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r_c</th>
<th>r_a</th>
<th>r_b</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td>r_c</td>
<td>r_a</td>
<td>16-bit signed constant</td>
<td></td>
</tr>
</tbody>
</table>

32 bits