6.004 Computation Structures
Spring 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.
Designing an Instruction Set

Let's Build a Simple Computer

A Programmable Control System

A First Program

Quiz 2 FRIDAY

A First Program

Computing \(N(N-1)\) with this data path is a multi-step process. We can control the processing at each step with a FSM. If we allow different control sequences to be loaded into the control FSM, then we allow the machine to be programmed.
An Optimized Program

A \rightarrow 1

B \leftarrow N

S0

A \leftarrow A \cdot B

B \leftarrow B - 1

A \leftarrow A \cdot B

S1

S2

S3

Some parts of the program can be computed simultaneously:

<table>
<thead>
<tr>
<th>Sn</th>
<th>Sn+1</th>
<th>A_sel</th>
<th>A_le</th>
<th>B_sel</th>
<th>B_le</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Computing Factorial

The advantage of a programmable control system is that we can reconfigure it to compute new functions.

In order to compute N!, we will need to add some new logic and an input to our control FSM:

Control Structure for Factorial

Programmability allows us to reuse data paths to solve new problems. What we need is a general purpose data path, which can be used to efficiently solve most problems as well as an easier way to control it.

<table>
<thead>
<tr>
<th>Z</th>
<th>S0</th>
<th>S1</th>
<th>A_sel</th>
<th>A_le</th>
<th>B_sel</th>
<th>B_le</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A Programmatic Engine

We've used the same data paths for computing N*(N-1) and Factorial; there are a variety of other computations we might implement simply by re-programming the control FSM.

Although our little machine is programmable, it falls short of a practical general-purpose computer – and fails the Turing Universality test – for three primary reasons:

1. It has very limited storage: it lacks the “expandable” memory resource of a Turing Machine.
2. It has a tiny repertoire of operations.
3. The “program” is fixed. It lacks the power, e.g., to generate a new program and then execute it.
A General-Purpose Computer
The von Neumann Model

Many architectural approaches to the general purpose computer have been explored. The one on which nearly all modern, practical computers is based was proposed by John von Neumann in the late 1940s. Its major components are:

Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.

Memory: storage of \(N\) words of \(W\) bits each, where \(W\) is a fixed architectural parameter, and \(N\) can be expanded to meet needs.

Input/Output: Devices for communicating with the outside world.

The Stored Program Computer

The von Neumann architecture easily addresses the first two limitations of our simple programmable machine example:

- A richer repertoire of operations, and
- An expandable memory.

But how does it achieve programmability?

CPU fetches and executes - interprets - successive instructions of the program...

- Program is simply data for the interpreter, specifying what computation to perform
- Single expandable resource pool - main memory - constrains both data and program size.

Anatomy of a von Neumann Computer

Coding of instructions raises some interesting choices...

- Tradeoffs: performance, compactness, programmability
- Uniformity. Should different instructions...
  - Be the same size?
  - Take the same amount of time to execute?
- Complexity. How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed

Mix of engineering & Art...

Trial (by simulation) is our best technique for making choices!

Our representative example: the \(\beta\) architecture!
**Programming Model**

a representative, simple, contemporary RISC

**Fetch/Execute loop:**
- fetch Mem[PC]
- PC = PC + 4†
- execute fetched instruction (may change PC!)
- repeat!

†Even though each memory word is 32-bits wide, for historical reasons the Beta uses byte memory addresses. Since each word contains four 8-bit bytes, addresses of consecutive words differ by 4.

**Main Memory**

<table>
<thead>
<tr>
<th>0 1 2 3</th>
<th>(4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit “words”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor State</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC = 00</td>
<td>3 2 1 0</td>
</tr>
<tr>
<td>r0</td>
<td>32-bit “words”</td>
</tr>
<tr>
<td>r1</td>
<td>next-instruction</td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td>0000000...0</td>
</tr>
</tbody>
</table>

**General Registers**

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>...</th>
<th>r31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000...0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Formats**

All Beta instructions fit in a single 32-bit word, whose fields encode combinations of:
- a 6-bit OPCODE (specifying one of < 64 operations)
- several 5-bit OPERAND locations, each one of the 32 registers
- an embedded 16-bit constant (“literal”)

There are two instruction formats:
- Opcode, 3 register operands (2 sources, destination)
  
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>unused</th>
</tr>
</thead>
</table>

- Opcode, 2 register operands, 16-bit literal constant
  
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r8</th>
<th>r9</th>
<th>16-bit signed constant</th>
</tr>
</thead>
</table>

**ALU Operations**

Sample coded operation: ADD instruction

```
0100000001100001100100000100100000011001111111111111101
```

OPCODE = 110000, encoding ADDC

r3 = 3, encoding R3 as destination

```
Reg[rc] = Reg[ra] + sxt(const)
```

```
arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SARC
```

**ALU Operations with Constant**

ADDC instruction: adds constant, register contents:

```
010000000110000110010000010010000001111111111111111101
```

OPCODE = 110000, encoding ADDC

r3 = 3, encoding R3 as destination

```
Reg[rc] = Reg[ra] + sxt(const)
```

```
add the contents of ra to const; store the result in rc
```

**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”
Do We Need Built-in Constants?

Percentage of the operations that use a constant operand

One way to answer architectural questions is to evaluate the consequences of different choices using carefully chosen representative benchmarks (programs and/or code sequences). Make choices that are "best" according to some metric (cost, performance, ...).

Baby’s First Beta Program

(fragment)

Suppose we have N in r1, and want to compute N*(N-1), leaving the result in r2:

SUBC(r1,1,r2) | put N-1 into r2
MUL(r2,r1,r2) | leave N*(N-1) in r2

These two instructions do what our little ad-hoc machine did. Of course, limiting ourselves to registers for storage falls short of our ambitions.... it amounts to the finite storage limitations of an FSM!

Needed: instruction-set support for reading and writing locations in main memory...

LOADS & STORES

\[
\begin{align*}
\text{LD} (ra, \text{const}, rc) & : \text{Reg}[rc] = \text{Mem}[\text{Reg}[ra] + \text{sxt}(	ext{const})] \\
\text{ST} (rc, \text{const}, ra) & : \text{Mem}[\text{Reg}[ra] + \text{sxt}(	ext{const})] = \text{Reg}[rc]
\end{align*}
\]

“Fetch into rc the contents of the memory location whose address is C plus the contents of ra”

Abbreviation: LD(C, rc) for LD(R31, C, rc)

“Store the contents of rc into the memory location whose address is C plus the contents of ra”

Abbreviation: ST(rc, C) for ST(rc, C, R31)

BYTE ADDRESSES, but only 32-bit word accesses to word-aligned addresses are supported. Low two address bits are ignored!
Common “Addressing Modes”

- **Absolute**: `constant`
  - Value = Mem[constant]
  - Use: accessing static data
- **Indirect (aka Register deferred)**: `(Rx)`
  - Value = Mem[Reg[x]]
  - Use: pointer accesses
- **Displacement**: `constant(Rx)`
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables
- **Indexed**: `(Rx + Ry)`
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: access to array elements
- **Scaled**: `constant(Rx)[Ry]`
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: access to array elements

β can do these with appropriate choices for Ra and const

Argh! Is the complexity worth the cost?
Need a cost/benefit analysis!

Capability so far: Expression Evaluation

Translation of an Expression:

```c
int x, y;
y = (x-3)*(y+123456)
x: long(0)
y: long(0)
c: long(123456)
...```

- **VARIABLES** are allocated storage in main memory
- **VARIABLE** references translate to LD or ST
- **OPERATORS** translate to ALU instructions
- **SMALL CONSTANTS** translate to ALU instructions with built-in constants
- **"LARGE" CONSTANTS** translate to initialized variables

NB: Here we assume that variable addresses fit into 16-bit constants

Can We Run Every Algorithm?

Model thus far:
- Executes instructions sequentially
- Number of operations executed = number of instructions in our program!

Good news: programs can’t "loop forever"!
- Halting problem is solvable for our current Beta subset!

Bad news: can’t compute Factorial:
- Only supports bounded-time computations
- Can’t do a loop, e.g., for Factorial!

Needed: ability to change the PC.

Memory Operands: Usage

<table>
<thead>
<tr>
<th>Mode</th>
<th>TeX</th>
<th>Spice</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoincrement</td>
<td>1%</td>
<td>3%</td>
<td>4%</td>
</tr>
<tr>
<td>Displacement deferred</td>
<td>2%</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>20%</td>
<td>41%</td>
</tr>
<tr>
<td>Register deferred</td>
<td>9%</td>
<td>4%</td>
<td>18%</td>
</tr>
<tr>
<td>Displacement</td>
<td>0%</td>
<td>56%</td>
<td>66%</td>
</tr>
</tbody>
</table>

Figure by MIT OpenCourseWare.
Beta Branch Instructions

The Beta’s branch instructions provide a way of conditionally changing the PC to point to some nearby location...

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

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

```markdown
NB: “offset” is a SIGNED CONSTANT encoded as part of the instruction!
```

BEQ (ra, label, rc): Branch if equal  
BNE (ra, label, rc): Branch if not equal

```markdown
PC = PC + 4;  
Reg[rc] = PC;  
if (Reg[ra] == 0)  
PC = PC + 4*offset;
```

```markdown
offset = (label - <addr of BNE/BEQ>) / 4 - 1  
= up to 32767 instructions before/after BNE/BEQ
```

Summary

- Programmable data paths provide some algorithmic flexibility, just by changing control structure.
- Interesting control structure optimization questions – e.g., what operations can be done simultaneously?
- von Neumann model for general-purpose computation: need
  - support for sufficiently powerful operation repertoire
  - Expandable Memory
  - Interpreter for program stored in memory
- ISA design requires tradeoffs, usually based on benchmark results: art, engineering, evaluation & incremental optimizations
- Compilation strategy
  - runtime “discipline” for software implementation of a general class of computations
  - Typically enforced by compiler, run-time library, operating system. We’ll see more of these!

Now we can do Factorial...

**Synopsis (in C):**
- Input in n, output in ans
- r1, r2 used for temporaries
- follows algorithm of our earlier data paths.

```markdown
int n, ans;  
r1 = 1;  
r2 = n;  
while (r2 != 0) {
  r1 = r1 * r2;  
  r2 = r2 - 1
}  
ans = r1;
```

**Beta code, in assembly language:**

```markdown
n: long(123)  
ans: long(0)  
...  
ADDC(r31, 1, r1) | r1 = 1  
LD(n, r2) | r2 = n  
loop: BEQ(r2, done, r31) | while (r2 != 0)  
MUL(r1, r2, r1) | r1 = r1 * r2  
SUBC(r2, 1, r2) | r2 = r2 - 1  
BEQ(r31, loop, r31) | Always branches!  
done: ST(r1, ans, r31) | ans = r1
```

Summary

- Programmable data paths provide some algorithmic flexibility, just by changing control structure.
- Interesting control structure optimization questions – e.g., what operations can be done simultaneously?
- von Neumann model for general-purpose computation: need
  - support for sufficiently powerful operation repertoire
  - Expandable Memory
  - Interpreter for program stored in memory
- ISA design requires tradeoffs, usually based on benchmark results: art, engineering, evaluation & incremental optimizations
- Compilation strategy
  - runtime “discipline” for software implementation of a general class of computations
  - Typically enforced by compiler, run-time library, operating system. We’ll see more of these!