9. Programmable Machines

6.004x Computation Structures
Part 2 – Computer Architecture

Copyright © 2015 MIT EECS
Example: Factorial

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

C:

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

initially: \[ a = 1, \ b = 5 \]
after iter 1: \[ a = 5, \ b = 4 \]
after iter 2: \[ a = 20, \ b = 3 \]
after iter 3: \[ a = 60, \ b = 2 \]
after iter 4: \[ a = 120, \ b = 1 \]
after iter 5: \[ a = 120, \ b = 0 \]
Done!
Example: Factorial

\[ \text{factorial}(N) = N! = N \times (N-1) \times \cdots \times 1 \]

C:

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

High-level FSM:

- Helpful to translate into hardware
- D-registers (a, b)
- 2-bits of state (start, loop, done)
- Boolean transitions (b'==0, b'!=0)
- Register assignments in states (e.g., a ← a * b)
Datapath for Factorial

- Draw registers
- Draw combinational circuit for each assignment
- Connect to input muxes

\[
\begin{align*}
\text{start} & \rightarrow \text{loop} & \text{done} & \rightarrow \text{loop} \\
\text{loop} & \rightarrow \text{done} & b & != 0 \\
& & b & == 0 \\
\end{align*}
\]

- \( a \leftarrow 1 \)
- \( a \leftarrow a \times b \)
- \( b \leftarrow N \)
- \( b \leftarrow b - 1 \)
- \( b \left\{ \begin{array}{ll}
\text{if } b \neq 0 & \text{then } a \leftarrow a \\
\text{else } & b \leftarrow b
\end{array} \right. \)
Control FSM for Factorial

- Draw combinational logic for transition conditions
- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition logic outputs
  - Outputs: Mux select signals

\[
\begin{array}{c|c}
0 & a \leftarrow 1 \\
1 & a \leftarrow a \times b \\
2 & a \leftarrow a \\
\end{array}\quad \begin{array}{c|c}
0 & b \leftarrow N \\
1 & b \leftarrow b - 1 \\
2 & b \leftarrow b \\
\end{array}
\]

Control FSM

\[
\begin{array}{c|c|c|c|c}
S & Z & wa_{SEL} & wb_{SEL} & S' \\
00 & 0 & 10 & 00 & 01 \\
00 & 1 & 10 & 00 & 01 \\
01 & 0 & 01 & 01 & 01 \\
01 & 1 & 01 & 01 & 10 \\
10 & 0 & 00 & 10 & 10 \\
10 & 1 & 00 & 10 & 10 \\
\end{array}
\]
Control FSM Hardware

ROM
8 locs x 6 bits

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100001</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>100001</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>010101</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>010110</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>001010</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>001010</td>
<td></td>
</tr>
</tbody>
</table>

Current state

Next state

2

2
So Far: Single-Purpose Hardware

• Problem → Procedure (High-level FSM) → Implementation

• Systematic way to implement high-level FSM as a datapath + control FSM
  – Is this implementation an FSM itself?
  – If so, can you draw the truth table?

• How should we generalize our approach so we can solve many problems with one set of hardware?
  – More storage for operands and results
  – A larger repertoire of operations
  – General-purpose datapath
A Simple Programmable Datapath

- Each cycle, this datapath:
  - Reads two operands \( (a, b) \) from 4 registers \( (R0-R3) \)
  - Performs one operation of \( +, -, *, \text{NAND} \) on operands
  - Optionally writes result to a register

- Control FSM:
A Control FSM for Factorial

• Assume initial register contents:
  - R0 value = 1
  - R1 value = N
  - R2 value = -1
  - R3 value = 0

• Control FSM:

```
-- Control FSM

asel = 0   asel = 1   asel = 1   asel = 1
bSEL = 1    bSEL = 2   bSEL = 3  bSEL = 3
opsel = 2 (*) opsel = 0 (+) opsel = X opsel = X
wen = 1     wen = 1    wen = 0    wen = 0
wSEL = 0    wSEL = 1   wSEL = X   wSEL = X

z == 0

loop mul  loop sub  loop beq  done

z == 1

R0 ← R0 * R1  R1 ← R1 + R2

N! in R0
```
New Problem → New Control FSM

• You can solve many more problems with this datapath!
  – Exponentiation, division, square root, ...
  – But nothing that requires more than four registers

• By designing a control FSM, we are programming the datapath

• Early digital computers were programmed this way!
  – ENIAC (1943):
    • First general-purpose digital computer
    • Programmed by setting huge array of dials and switches
    • Reprogramming it took about 3 weeks
"Eniac" by Unknown - U.S. Army Photo.
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

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

```
rc ← op(ra, rb)
```
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

**Datapath**
- Internal storage
- Address
- Data
- Control
- Status

**Control Unit**
- Address
- Instructions

**Main Memory**
- Address

**Registers**
- dest
- asel
- bsel

**Operations**
- ALU
- status

**Main Memory**
- 1101000111011
- \( R1 \leftarrow R2 + R3 \)
**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

```
Fetch instruction
Decode instruction
Read src operands
Execute
Write dst operand
Compute next PC
```
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
    - Both implement x86 ISA
  - 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

PC

r0

r1

r2

...

r31

General Registers

32-bit “words”

000000...0

r31 hardwired to 0

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
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;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>n</td>
</tr>
<tr>
<td>0x1004</td>
<td>r</td>
</tr>
<tr>
<td>0x1008</td>
<td>x</td>
</tr>
<tr>
<td>0x100C</td>
<td>y</td>
</tr>
<tr>
<td>0x1010</td>
<td></td>
</tr>
</tbody>
</table>

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\textsubscript{c}</th>
<th>r\textsubscript{a}</th>
<th>r\textsubscript{b}</th>
<th>unused</th>
</tr>
</thead>
</table>

Example coded instruction: ADD

32-bit hex: \texttt{0x80611000}  
We prefer to write a \textit{symbolic representation}: \texttt{ADD(r1,r2,r3)}

\texttt{ADD(ra,rb,rc)}:

\begin{align*}
\text{Reg}[rc] & \leftarrow \text{Reg}[ra] + \text{Reg}[rb] \\
\text{“Add the contents of ra to the contents of rb; store the result in rc”}
\end{align*}

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

operations

PC

4
Should We Support Constant Operands?

Many programs use small constants frequently
e.g., our factorial example: 0, 1, -1

**Tradeoff:**
- When used, they save registers and instructions
- More opcodes → 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 indicies*
- >80% of executed compare instructions
  - *Loop termination condition*
- >25% of executed load instructions
  - *Offsets into data structures*
Beta ALU Instructions with Constant

Format:

| OPCODE | $r_c$ | $r_a$ | 16-bit signed constant |

Example instruction: ADDC adds register contents and constant:

OPCODE = 110000, encoding ADDC

$r_c$ = 3, encoding R3 as destination

$r_a$ = 1, encoding R1 as first operand

16-bit two’s complement constant, encoding -3 as second operand (will be sign-extended to become 32-bit two’s complement operand)

Symbolic version: ADDC($r_1$, -3, $r_3$)

ADDC($r_a$, const, $r_c$):

Reg[$r_c$] ← Reg[$r_a$] + 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, CMPELEC

boolean: ANDC, ORC, XORC, XNORC

shift: SHLC, SHRC, SARC
Next we add the datapath hardware to support small constants as the second ALU operand:

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

![Diagram of ALU with added hardware for small constants]
Beta Load and Store Instructions

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

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

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

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

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

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.
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?

• No! Recall high-level FSM:

  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).

### Beta Branch Instructions Table

<table>
<thead>
<tr>
<th>BEQ or BNE</th>
<th>(r_c)</th>
<th>(r_a)</th>
<th>16-bit signed constant</th>
</tr>
</thead>
</table>

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

- \(\text{NPC } \leftarrow \text{PC } + 4\)
- \(\text{Reg}[rc] \leftarrow \text{NPC}\)
- if (\(\text{Reg}[ra] == 0\))
  - \(\text{PC } \leftarrow \text{NPC } + 4*\text{offset}\)
- else
  - \(\text{PC } \leftarrow \text{NPC}\)

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

- \(\text{NPC } \leftarrow \text{PC } + 4\)
- \(\text{Reg}[rc] \leftarrow \text{NPC}\)
- if (\(\text{Reg}[ra] != 0\))
  - \(\text{PC } \leftarrow \text{NPC } + 4*\text{offset}\)
- else
  - \(\text{PC } \leftarrow \text{NPC}\)

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

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

- Remember control FSM for our simple programmable datapath?
  - Control FSM states \(\rightarrow\) instructions!
    - Not the case in general
    - Happens here because datapath is similar to basic von Neumann datapath
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 | r_c | r_a | unused |

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

Useful for procedure call return...

\[
\begin{align*}
[0x100] & \quad \text{BEQ(R31,sqrt,R28)} \\
[0x678] & \quad \text{BEQ(R31, sqrt, R28)} \\
\end{align*}
\]

sqrt:

\[
\begin{align*}
\text{R28} & = 0x104 \\
\text{R28} & = 0x67C \\
\text{JMP(R28, R31)} \\
1^{\text{st}} \text{ time: PC} & \leftarrow 0x104 \\
2^{\text{nd}} \text{ time: PC} & \leftarrow 0x67C
\end{align*}
\]
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:

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

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

32 bits

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