Low-Level Programming

ICOM 4036
Lecture 4

Prof. Bienvenido Velez
Practical Universal Computers
(John) Von Neumann Architecture (1945)

CPU is a universal TM
An interpreter of some programming language (PL)

This looks just like a TM Tape

John von Neumann, 1950's
Outline

• The Von Neumann Architecture
• From Voltages to Computers
• Low-level Programming
• Implementing HLL Abstractions
  – Control structures
  – Data Structures
  – Procedures and Functions
The (John) Von Neumann Architecture (late 40’s)

- I/O devices: Allow communication with outside world
- Central Processing Unit (CPU): Interprets instructions
- Memory: Stores both programs and data

After 60 years ... most processors still look like this!
The von Neumann Architecture
Central Processing (CPU)

Control Unit (FSM)

active

control

status

Data Paths
Interconnected registers, muxes, decoders, …

passive

I/O devices

Central Processing Unit (CPU)

Memory
The (John) Von Neumann Architecture
The Memory Unit

I/O devices

Central Processing Unit (CPU)

Memory

word size

address space

0
ADD A

1
SUB B

2
JUMP 1

3

A

B

...

N-1
The (John) Von Neumann Architecture

Stored Program Concept

- Programs and their data coexist in memory
- Processor, under program control, keeps track of what needs to be interpreted as instructions and what as data.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A</td>
<td></td>
</tr>
<tr>
<td>SUB B</td>
<td></td>
</tr>
<tr>
<td>JUMP 1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
</tbody>
</table>
Easy I
Memory Interface

CPU

MEMORY

address

data word

memory op
\{R, W, NOP\}
Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Format (16 bits)

I | opcode | X

I = Indirect bit
Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Set

<table>
<thead>
<tr>
<th>Name</th>
<th>Opcode</th>
<th>Action I=0</th>
<th>Action I=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp</td>
<td>00 000</td>
<td>AC ← not AC</td>
<td>AC ← not AC</td>
</tr>
<tr>
<td>ShR</td>
<td>00 001</td>
<td>AC ← AC / 2</td>
<td>AC ← AC / 2</td>
</tr>
<tr>
<td>BrN</td>
<td>00 010</td>
<td>AC &lt; 0 ⇒ PC ← X</td>
<td>AC &lt; 0 ⇒ PC ← MEM[X]</td>
</tr>
<tr>
<td>Jump</td>
<td>00 011</td>
<td>PC ← X</td>
<td>PC ← MEM[X]</td>
</tr>
<tr>
<td>Store</td>
<td>00 100</td>
<td>MEM[X] ← AC</td>
<td>MEM[MEM[X]] ← AC</td>
</tr>
<tr>
<td>Load</td>
<td>00 101</td>
<td>AC ← MEM[X]</td>
<td>AC ← MEM[MEM[X]]</td>
</tr>
<tr>
<td>And</td>
<td>00 110</td>
<td>AC ← AC and X</td>
<td>AC ← AC and MEM[X]</td>
</tr>
<tr>
<td>Add</td>
<td>00 111</td>
<td>AC ← AC + X</td>
<td>AC ← AC + MEM[X]</td>
</tr>
</tbody>
</table>

Easy all right … but universal it is!
Easy 1
Data Paths (with control points)
Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Some Immediate Observations on the Easy I ISA

• Accumulator (AC) is implicit operand to many instructions. No need to use instruction bits to specify one of the operands. More bits left for address and opcodes.

• Although simple, Easy I is universal. (given enough memory). Can you see this?

• Immediate bit specifies level of indirection for the location of the operand. I = 1: operand in X field (immediate). I=1 operand in memory location X (indirect).
Easy I - Control Unit

Control Unit
Combinational Logic

DataPaths + state

AC:15
OpCode
I bit
Current State

EDB_{sel}
AO_{le}
AO_{sel}
AC_{le}
DI_{le}
PC_{is}
PC_{sel}
MEM_{op}
ALU_{op}
Next State

clock

Spring 2006
ICOM 4036 Programming Languages
Lecture 4
What makes a CPU cycle?

Cycle time must accommodate signal propagation
Easy I – Timing Example
ALU Operation

DI

A

ALU

B

AC

CLK

DI_{le}

DI_{out}

ALU_{out}

AC_{le}

AC_{out}
Easy I
Control Unit
(Level 0 Flowcharts)

Fetch
Read next instruction

Decode
Determine what it does and prepare to do it. Fetch operands.

Execute
Do it!

We will ignore indirect bit (assuming I = 0) for now
Easy I
Control Unit
(Level 1 Flowcharts)

Level 1: Each box may take several CPU cycles to execute
Easy I  
Control Unit  
(Level 2 Flowcharts)

Each box may take only one CPU cycle to execute

Byte Addressable  
Can you tell why?

RESET
Invariant
At the beginning of the fetch cycle
AO holds address of instruction to be fetched and PC points to following instruction

Opcode must be an input to CU’s sequential circuit
Easy I
Control Unit
(Level 2 Flowcharts)

AOpr

DI \rightarrow \text{ABUS} \rightarrow \text{ALUA}
AC \rightarrow \text{ALUB}
\text{ALU} \rightarrow \text{AC}
PC \rightarrow \text{AO}
PC + 2 \rightarrow \text{PC}

\text{Restore fetch invariant}

\text{fetch}
# Easy I Control Unit

*(Level 2 Flowcharts)*

<table>
<thead>
<tr>
<th>sopr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC → ALUB</td>
</tr>
<tr>
<td>ALU → AC</td>
</tr>
<tr>
<td>PC → AO</td>
</tr>
<tr>
<td>PC + 2 → PC</td>
</tr>
</tbody>
</table>

```
SOpr
```

```
AC + 2
```

```
fetch
```
Easy I
Control Unit
(Level 2 Flowcharts)

load1

DI<0:9> \rightarrow ABUS \rightarrow AO

load2

AO \rightarrow EAB
EDB \rightarrow DI

load3

DI \rightarrow ABUS \rightarrow ALUA
ALU \rightarrow AC
PC \rightarrow AO
PC + 2 \rightarrow PC

fetch

Load
Easy I
Control Unit
(Level 2 Flowcharts)

- DI<0:9> → ABUS → AO
- AC → EDB
- AO → EAB
- PC → AO
- PC + 2 → PC
- Store
- Store1
- Store2
- Fetch
Easy I
Control Unit
(Level 2 Flowcharts)

PC → AO
PC + 2 → PC

Assume branch not taken. Allow AC:15 to propagate.

1 (AC<0)

DI<0:9> → PC
DI<0:9> → AO
PC + 2 → PC

0 (AC>0)

Can we accomplish all this in 1 cycle? How?

BrN

Bit 15 of AC input to the CU’s sequential circuit
Inside the Easy-I PC

PC capable of loading and incrementing simultaneously
Easy I
Control Unit
(Level 2 Flowcharts)

- DI<0:9> → PC
- DI<0:9> → AO
- PC + 2 → PC
Easy I
Data Paths (with control points)
### Easy I

**Control Unit State Transition Table (Part I)**

<table>
<thead>
<tr>
<th>Curr State</th>
<th>opcode</th>
<th>AC:15</th>
<th>Next State</th>
<th>ALU op</th>
<th>Mem OP</th>
<th>PC sel</th>
<th>PC is</th>
<th>DI le</th>
<th>AC le</th>
<th>AO sel</th>
<th>AO le</th>
<th>EDB sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset1</td>
<td>xx xxx</td>
<td>x</td>
<td>reset2</td>
<td>XXX</td>
<td>NOP</td>
<td>01</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>reset2</td>
<td>xx xxx</td>
<td>x</td>
<td>fetch</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>fetch</td>
<td>00 00x</td>
<td>x</td>
<td>sopr</td>
<td>XXX</td>
<td>NOP</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>fetch</td>
<td>00 010</td>
<td>x</td>
<td>brn1</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>fetch</td>
<td>00 011</td>
<td>x</td>
<td>jump</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>fetch</td>
<td>00 100</td>
<td>x</td>
<td>store1</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>fetch</td>
<td>00 101</td>
<td>x</td>
<td>load1</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>fetch</td>
<td>00 11x</td>
<td>x</td>
<td>aopr</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>aopr</td>
<td>00 110</td>
<td>x</td>
<td>fetch</td>
<td>AND</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>aopr</td>
<td>00 111</td>
<td>x</td>
<td>fetch</td>
<td>ADD</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>sopr</td>
<td>00 000</td>
<td>x</td>
<td>fetch</td>
<td>NOTB</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>sopr</td>
<td>00 001</td>
<td>x</td>
<td>fetch</td>
<td>SHRB</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
## Control Unit State Transition Table (Part II)

<table>
<thead>
<tr>
<th>Current State</th>
<th>opcode</th>
<th>AC:15</th>
<th>Next State</th>
<th>ALU op</th>
<th>Mem OP</th>
<th>PC sel</th>
<th>PC is</th>
<th>DI le</th>
<th>AC le</th>
<th>AO sel</th>
<th>AO le</th>
<th>EDB sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>store1</td>
<td>xx xxx</td>
<td>x</td>
<td>store2</td>
<td>XXX</td>
<td>NOP</td>
<td>11</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>store2</td>
<td>xx xxx</td>
<td>x</td>
<td>store3</td>
<td>XXX</td>
<td>WR</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>load1</td>
<td>xx xxx</td>
<td>x</td>
<td>load2</td>
<td>XXX</td>
<td>NOP</td>
<td>11</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>load2</td>
<td>xx xxx</td>
<td>x</td>
<td>load3</td>
<td>XXX</td>
<td>RD</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>load3</td>
<td>xx xxx</td>
<td>x</td>
<td>fetch</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>brn1</td>
<td>xx xxx</td>
<td>0</td>
<td>fetch</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>brn1</td>
<td>xx xxx</td>
<td>1</td>
<td>brn2</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>brn2</td>
<td>xx xxx</td>
<td>x</td>
<td>fetch</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>jump</td>
<td>xx xxx</td>
<td>x</td>
<td>fetch</td>
<td>XXX</td>
<td>NOP</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

CU with 14 states => 4 bits of state

This is a (micro)program that interprets machine code
Easy-I Control Unit – Some missing details

4-bit Encodings for States

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset1</td>
<td>0000</td>
</tr>
<tr>
<td>reset2</td>
<td>0001</td>
</tr>
<tr>
<td>fetch</td>
<td>0010</td>
</tr>
<tr>
<td>aopr</td>
<td>0011</td>
</tr>
<tr>
<td>sopr</td>
<td>0100</td>
</tr>
<tr>
<td>store1</td>
<td>0101</td>
</tr>
<tr>
<td>store2</td>
<td>0110</td>
</tr>
<tr>
<td>store3</td>
<td>0111</td>
</tr>
<tr>
<td>load1</td>
<td>1000</td>
</tr>
<tr>
<td>load2</td>
<td>1001</td>
</tr>
<tr>
<td>load3</td>
<td>1010</td>
</tr>
<tr>
<td>brn1</td>
<td>1011</td>
</tr>
<tr>
<td>brn2</td>
<td>1100</td>
</tr>
<tr>
<td>jump</td>
<td>1101</td>
</tr>
</tbody>
</table>

ALU Operation Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>000</td>
<td>A</td>
</tr>
<tr>
<td>NOTB</td>
<td>001</td>
<td>not B</td>
</tr>
<tr>
<td>AND</td>
<td>010</td>
<td>A and B</td>
</tr>
<tr>
<td>ADDR</td>
<td>011</td>
<td>A + B</td>
</tr>
<tr>
<td>SHRBO</td>
<td>100</td>
<td>B / 2</td>
</tr>
</tbody>
</table>

We know how to implement this ALU!

Control Bus Operation Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>00</td>
</tr>
<tr>
<td>ReaD</td>
<td>01</td>
</tr>
<tr>
<td>WRite</td>
<td>10</td>
</tr>
</tbody>
</table>
## Easy I

**Control Unit State Transition Table (Part I)**

<table>
<thead>
<tr>
<th>Curr State</th>
<th>opcode</th>
<th>AC:15</th>
<th>Next State</th>
<th>ALU op</th>
<th>Mem OP</th>
<th>PC sel</th>
<th>PC is</th>
<th>DI le</th>
<th>AC le</th>
<th>AO sel</th>
<th>AO le</th>
<th>EDB sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>xx xxx</td>
<td>x</td>
<td>0001</td>
<td>XXX</td>
<td>00</td>
<td>01</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>xx xxx</td>
<td>x</td>
<td>0010</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 00x</td>
<td>x</td>
<td>0100</td>
<td>XXX</td>
<td>00</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 010</td>
<td>x</td>
<td>1011</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 011</td>
<td>x</td>
<td>1101</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 100</td>
<td>x</td>
<td>0101</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 101</td>
<td>x</td>
<td>1000</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0010</td>
<td>00 11x</td>
<td>x</td>
<td>0011</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0011</td>
<td>00 110</td>
<td>x</td>
<td>0010</td>
<td>010</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>00 111</td>
<td>x</td>
<td>0010</td>
<td>011</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>00 000</td>
<td>x</td>
<td>0010</td>
<td>001</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>00 001</td>
<td>x</td>
<td>0010</td>
<td>100</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## Control Unit State Transition Table (Part II)

<table>
<thead>
<tr>
<th>Current State</th>
<th>opcode</th>
<th>AC:15</th>
<th>Next State</th>
<th>ALU op</th>
<th>Mem OP</th>
<th>PC sel</th>
<th>PC is</th>
<th>DI le</th>
<th>AC le</th>
<th>AO sel</th>
<th>AO le</th>
<th>EDB sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>xx xxxx</td>
<td>x</td>
<td>0110</td>
<td>XXX</td>
<td>00</td>
<td>11</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0110</td>
<td>xx xxxx</td>
<td>x</td>
<td>0111</td>
<td>XXX</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>xx xxxx</td>
<td>x</td>
<td>1001</td>
<td>XXX</td>
<td>00</td>
<td>11</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1001</td>
<td>xx xxxx</td>
<td>x</td>
<td>1010</td>
<td>XXX</td>
<td>01</td>
<td>11</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1010</td>
<td>xx xxxx</td>
<td>x</td>
<td>0010</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1011</td>
<td>xx xxxx</td>
<td>0</td>
<td>0010</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1011</td>
<td>xx xxxx</td>
<td>1</td>
<td>1100</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1100</td>
<td>xx xxxx</td>
<td>x</td>
<td>0010</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1101</td>
<td>xx xxxx</td>
<td>x</td>
<td>0010</td>
<td>XXX</td>
<td>00</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
Building the Easy-I C-Unit

2 Approaches

• **Harwired**
  – Apply well known sequential circuit techniques

• **Micro-programmed**
  – Treat state transition table as a program
  – Build a new abstraction layer

The Microprogramming abstraction level
Building the Easy-I C-Unit
Hardwired Approach

Control
Unit

ROM

next
state

control
point
signals

state

Memory
Unit

Data
Paths

10

AC:15

DI<10:14>

5

4

11

2

control
bus

bus

address

bus

data

next
state

control
point
signals
Computing Integer Division
Iterative C++ Version

int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}

We ignore procedures and I/O for now
Definition
Instruction Set Architecture

• What it is:
  – The programmers view of the processor
  – Visible registers, instruction set, execution model, memory model, I/O model

• What it is not:
  – How the processors if build
  – The processor’s internal structure
Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Format (16 bits)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>opcode</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I = Indirect bit
# Easy I

## A Simple Accumulator Processor

### Instruction Set Architecture (ISA)

## Instruction Set

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Opcode</th>
<th>Action $i=0$</th>
<th>Symbolic Name</th>
<th>Action $i=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp</td>
<td>00 000</td>
<td>AC ← not AC</td>
<td>Comp</td>
<td>AC ← not AC</td>
</tr>
<tr>
<td>ShR</td>
<td>00 001</td>
<td>AC ← AC / 2</td>
<td>ShR</td>
<td>AC ← AC / 2</td>
</tr>
<tr>
<td>BrNi</td>
<td>00 010</td>
<td>AC &lt; 0 ⇒ PC ← X</td>
<td>BrN</td>
<td>AC &lt; 0 ⇒ PC ← MEM[X]</td>
</tr>
<tr>
<td>Jumpi</td>
<td>00 011</td>
<td>PC ← X</td>
<td>Jump</td>
<td>PC ← MEM[X]</td>
</tr>
<tr>
<td>Storei</td>
<td>00 100</td>
<td>MEM[X] ← AC</td>
<td>Store</td>
<td>MEM[MEM[X]] ← AC</td>
</tr>
<tr>
<td>Loadi</td>
<td>00 101</td>
<td>AC ← MEM[X]</td>
<td>Load</td>
<td>AC ← MEM[MEM[X]]</td>
</tr>
<tr>
<td>Andi</td>
<td>00 110</td>
<td>AC ← AC and X</td>
<td>And</td>
<td>AC ← AC and MEM[X]</td>
</tr>
<tr>
<td>Addi</td>
<td>00 111</td>
<td>AC ← AC + X</td>
<td>Add</td>
<td>AC ← AC + MEM[X]</td>
</tr>
</tbody>
</table>
# Easy I Memory Model

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD A</td>
</tr>
<tr>
<td>4</td>
<td>SUB B</td>
</tr>
<tr>
<td>6</td>
<td>JUMP 1</td>
</tr>
<tr>
<td>512</td>
<td></td>
</tr>
</tbody>
</table>

- **8 bits**
- **8 bits**

---

Spring 2006

ICOM 4036 Programming Languages

Lecture 4
Computing Integer Division
Iterative C++ Version

```c++
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```
Computing Integer Division
Iterative C++ Version

```c++
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Data: Global Layout

```assembly
0:   andi  0   # AC = 0
     addi 12
     storei 1000 # a = 12 (a stored @ 1000)
     andi  0   # AC = 0
     addi  4
     storei 1004 # b = 4 (b stored @ 1004)
     andi  0   # AC = 0
     storei 1008 # result = 0 (result @ 1008)
```

Issues
- Memory allocation
- Data Alignment
- Data Sizing
Computing Integer Division
Iterative C++ Version

```c
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Code: Conditionals
If-Then

```
0: andi 0    # AC = 0
addi 12
storei 1000 # a = 12 (a stored @ 1000)
andi 0     # AC = 0
addi 4
storei 1004 # b = 4 (b stored @ 1004)
andi 0     # AC = 0
storei 1008 # result = 0 (result @ 1008)
main: loadi 1004 # compute a – b in AC
comp # using 2’s complement add
addi 1
add 1000
brni exit # exit if AC negative
```

Issues
- Must translate HLL boolean expression into ISA-level branching condition
Computing Integer Division
Iterative C++ Version

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

Translate Code: Iteration (loops)

```
0: andi 0 # AC = 0
addi 12
storei 1000 # a = 12 (a stored @ 1000)
andi 0 # AC = 0
addi 4
storei 1004 # b = 4 (b stored @ 1004)
andi 0 # AC = 0
storei 1008 # result = 0 (result @ 1008)
main: loadi 1004 # compute a – b in AC
comp # using 2’s complement add
addi 1
add 1000
brni exit # exit if AC negative
loop: loadi 1000
brni endloop

jump loop
endloop:
exit:
```
Computing Integer Division
Iterative C++ Version

```c++
int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}
```

Translate Code: Arithmetic Ops

```
0:  andi  0     # AC = 0
    addi 12
storei 1000  # a = 12 (a stored @ 1000)
and 0  # AC = 0
addi 4
storei 1004  # b = 4 (b stored @ 1004)
and 0  # AC = 0
storei 1008  # result = 0 (result @ 1008)
main:  loadi 1004  # compute a – b in AC
        comp  # using 2’s complement add
        addi 1
        add 1000
        brni exit  # exit if AC negative
loop:  loadi 1000
        brni endloop
        loadi 1004  # compute a – b in AC
        comp  # using 2’s complement add
        addi 1
        add 1000  # Uses indirect bit I = 1
        jumpi loop
endloop: exit:
```
Computing Integer Division
Iterative C++ Version

C++

HLL

Easy-I
Assembly Language

Translate Code: Assignments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

```
0: andi 0       # AC = 0
    addi 12
    storei 1000   # a = 12 (a stored @ 1000)
    andi 0        # AC = 0
    addi 4
    storei 1004   # b = 4  (b stored @ 1004)
    andi 0        # AC = 0
    storei 1008   # result = 0 (result @ 1008)
main: loadi 1004 # compute a - b in AC
    comp          # using 2’s complement add
    addi 1
    add 1000
    brni exit     # exit if AC negative
loop: loadi 1000
    brni endloop
    loadi 1004    # compute a - b in AC
    comp          # using 2’s complement add
    addi 1
    add 1000     # Uses indirect bit I = 1
    storei 1000

jump loop
endloop:
exit:
```
Computing Integer Division
Iterative C++ Version

int a = 12;
int b = 4;
int result = 0;
main () {
    if (a >= b) {
        while (a > 0) {
            a = a - b;
            result ++;
        }
    }
}

Translate Code: Increments

0: andi 0   # AC = 0
    addi 12
    storei 1000  # a = 12 (a stored @ 1000)
    andi 0      # AC = 0
    addi 4
    storei 1004  # b = 4  (b stored @ 1004)
    andi 0      # AC = 0
    storei 1008  # result = 0 (result @ 1008)

main:  loadi 1004  # compute a – b in AC
       comp        # using 2’s complement add
addi 1
add 1000
brni exit   # exit if AC negative

loop:  loadi 1000
       brni endloop
loadi 1004  # compute a – b in AC
       comp        # using 2’s complement add
addi 1
add 1000   # Uses indirect bit I = 1
storei 1000
loadi 1008  # result = result + 1
addi 1
storei 1008
jumpi loop

endloop: exit:
### Computing Integer Division

#### Easy I Machine Code

#### Challenge
Make this program as small and fast as possible.

**Data**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>a</td>
</tr>
<tr>
<td>1004</td>
<td>b</td>
</tr>
<tr>
<td>1008</td>
<td>result</td>
</tr>
</tbody>
</table>

**Program**

<table>
<thead>
<tr>
<th>Address</th>
<th>I Bit</th>
<th>Opcode (binary)</th>
<th>X (base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>00 111</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>00 100</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>00 111</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00 100</td>
<td>1004</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>00 110</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>00 100</td>
<td>1008</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>00 101</td>
<td>1004</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>00 000</td>
<td>unused</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>00 111</td>
<td>1000</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>00 010</td>
<td>46</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>00 101</td>
<td>1000</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>00 010</td>
<td>46</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>00 101</td>
<td>1004</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>00 000</td>
<td>unused</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>00 100</td>
<td>1000</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>00 101</td>
<td>1008</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>00 111</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>00 100</td>
<td>1008</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td>00 011</td>
<td>26</td>
</tr>
</tbody>
</table>
The MIPS Architecture
ISA at a Glance

• Reduced Instruction Set Computer (RISC)
• 32 general purpose 32-bit registers
• Load-store architecture: Operands in registers
• Byte Addressable
• 32-bit address space
The MIPS Architecture

32 Register Set (32-bit registers)

<table>
<thead>
<tr>
<th>Register #</th>
<th>Reg Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r0</td>
<td>Zero constant</td>
</tr>
<tr>
<td>r4-r7</td>
<td>a0-a3</td>
<td>Function arguments</td>
</tr>
<tr>
<td>r1</td>
<td>at</td>
<td>Reserved for Operating Systems</td>
</tr>
<tr>
<td>r30</td>
<td>fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>r28</td>
<td>gp</td>
<td>Global memory pointer</td>
</tr>
<tr>
<td>r26-r27</td>
<td>k0-k1</td>
<td>Reserved for OS Kernel</td>
</tr>
<tr>
<td>r31</td>
<td>ra</td>
<td>Function return address</td>
</tr>
<tr>
<td>r16-r23</td>
<td>s0-s7</td>
<td>Callee saved registers</td>
</tr>
<tr>
<td>r29</td>
<td>sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>r8-r15</td>
<td>t0-t7</td>
<td>Temporary variables</td>
</tr>
<tr>
<td>r24-r25</td>
<td>t8-t9</td>
<td>Temporary variables</td>
</tr>
<tr>
<td>r2-r3</td>
<td>v0-v1</td>
<td>Function return values</td>
</tr>
</tbody>
</table>
The MIPS Architecture
Main Instruction Formats

Simple and uniform 32-bit 3-operand instruction formats

- **R Format**: Arithmetic/Logic operations on registers

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **I Format**: Branches, loads and stores

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Address/Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **J Format**: Jump Instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>Address/Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
MIPS Data Paths

FIGURE 5.48 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EFC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers.
Mips Packaging
The MIPS Architecture

Examples of Native Instruction Set

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic/Logic</td>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>addi $s1,$s2,K</td>
<td>$s1 = $s2 + K</td>
</tr>
<tr>
<td>Load/Store</td>
<td>lw $s1,K($s2)</td>
<td>$s1 = MEM[$s2+K]</td>
</tr>
<tr>
<td></td>
<td>sw $s1,K($s2)</td>
<td>MEM[$s2+K] = $s1</td>
</tr>
<tr>
<td>Jumps and Conditional</td>
<td>beq $s1,$s2,K</td>
<td>if ($s1=$s2) goto PC + 4 + K</td>
</tr>
<tr>
<td>Branches</td>
<td>slt $s1,$s2,$s3</td>
<td>if ($s2&lt;$s3) $s1=1 else $s1=0</td>
</tr>
<tr>
<td></td>
<td>j K</td>
<td>goto K</td>
</tr>
<tr>
<td>Procedures</td>
<td>jal K</td>
<td>$ra = PC + 4; goto K</td>
</tr>
<tr>
<td></td>
<td>jr $ra</td>
<td>goto $ra</td>
</tr>
</tbody>
</table>
# The SPIM Assembler

## Examples of Pseudo-Instruction Set

<table>
<thead>
<tr>
<th>Instruction Group</th>
<th>Syntax</th>
<th>Translates to:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic/Logic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg $s1, $s2</td>
<td>sub $s1, $r0, $s2</td>
<td></td>
</tr>
<tr>
<td>not $s1, $s2</td>
<td>nor $17, $18, $0</td>
<td></td>
</tr>
<tr>
<td><strong>Load/Store</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>li $s1, K</td>
<td>ori $s1, $0, K</td>
<td></td>
</tr>
<tr>
<td>la $s1, K</td>
<td>lui $at, 152</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ori $s1, $at, -27008</td>
<td></td>
</tr>
<tr>
<td>move $s1, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Jumps and Conditional Branches</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bgt $s1, $s2, K</td>
<td>slt $at, $s1, $s2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne $at, $0, K</td>
<td></td>
</tr>
<tr>
<td>sge $s1, $s2, $s3</td>
<td>bne $s3, $s2, foo</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ori $s1, $0, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>beq $0, $0, bar</td>
<td></td>
</tr>
<tr>
<td>foo:</td>
<td>slt $s1, $s3, $s2</td>
<td></td>
</tr>
<tr>
<td>bar:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pseudo Instructions:** translated to native instructions by Assembler
### The SPIM Assembler

#### Examples of Assembler Directives

<table>
<thead>
<tr>
<th>Group</th>
<th>Directive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Segmentation</td>
<td>.data &lt;addr&gt;</td>
<td>Data Segment starting at</td>
</tr>
<tr>
<td></td>
<td>.text &lt;addr&gt;</td>
<td>Text (program) Segment</td>
</tr>
<tr>
<td></td>
<td>.stack &lt;addr&gt;</td>
<td>Stack Segment</td>
</tr>
<tr>
<td></td>
<td>.ktext &lt;addr&gt;</td>
<td>Kernel Text Segment</td>
</tr>
<tr>
<td></td>
<td>.kdata &lt;addr&gt;</td>
<td>Kernel Data Segment</td>
</tr>
<tr>
<td>Data Allocation</td>
<td>x: .word &lt;value&gt;</td>
<td>Allocates 32-bit variable</td>
</tr>
<tr>
<td></td>
<td>x: .byte &lt;value&gt;</td>
<td>Allocates 8-bit variable</td>
</tr>
<tr>
<td></td>
<td>x: .ascii “hello”</td>
<td>Allocates 8-bit cell array</td>
</tr>
<tr>
<td>Other</td>
<td>.globl x</td>
<td>x is external symbol</td>
</tr>
</tbody>
</table>

**Assembler Directives:** Provide assembler additional info to generate machine code
Handy MIPS ISA References

- Appendix A: Patterson & Hennessy
- SPIM ISA Summary on class website
- Patterson & Hennessy Back Cover
The MIPS Architecture
Memory Model

32-bit byte addressable address space
Computing Integer Division
Iterative C++ Version

```cpp
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b) {
        a = a - b;
        result ++;
    }
}
```

MIPS/SPIM Version

```mips
.data  # Use HLL program as a comment
    x: .word 12  # int x = 12;
    y: .word 4  # int y = 4;
    res: .word 0  # int res = 0;
.globl main
.text
main: la $s0, x  # Allocate registers for globals
    lw $s1, 0($s0)  # x in $s1
    lw $s2, 4($s0)  # y in $s2
    lw $s3, 8($s0)  # res in $s3
while: bgt $s2, $s1, endwhile  # while (x >= y) {
    sub $s1, $s1, $s2  # x = x - y;
    addi $s3, $s3, 1  # res ++;
    j while  # }
endwhile:
    la $s0, x  # Update variables in memory
    sw $s1, 0($s0)
    sw $s2, 4($s0)
    sw $s3, 8($s0)
```
Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b) {
        a = a - b;
        result ++;
    }
}
printf("Result = %d 
");
```

MIPS/SPIM Version
Input/Output in SPIM

```
.data # Use HLL program as a comment
    x: .word 12 # int x = 12;
    y: .word 4 # int y = 4;
    res: .word 0 # int res = 0;
    pf1: .asciiz "Result = 
.text
.globl main
.text
main: la $s0, x # Allocate registers for globals
    lw $s1, 0($s0) #   x in $s1
    lw $s2, 4($s0) #   y in $s2
    lw $s3, 8($s0) #   res in $s3
while: bgt $s2, $s1, endwhile # while (x >= y) {
    sub $s1, $s1, $s2 #   x = x - y;
    addi $s3, $s3, 1 #   res ++;
    j while # }
endwhile:
    la $a0, pf1 # printf("Result = %d \n");
    li $v0, 4 # //system call to print_str
    syscall
    move $a0, $s3
    li $v0, 1 # //system call to print_int
    syscall
    la $s0, x # Update variables in memory
    sw $s1, 0($s0)
    sw $s2, 4($s0)
    sw $s3, 8($s0)
```

Spring 2006
Lecture 4
SPIM Assembler Abstractions

• Symbolic Labels
  – Instruction addresses and memory locations

• Assembler Directives
  – Memory allocation
  – Memory segments

• Pseudo-Instructions
  – Extend native instruction set without complicating architecture

• Macros