Building the Computer

- Retrieval of instructions from memory (**fetch**),
- Interpretation of the instructions (**decode**),
- Execution of the instructions (**execute**),
- Reading and writing data to/from memory (**read/write**).
Review (Notes #1)

*The instruction pipeline*

- **Fetch:**
  - Read an instruction from the memory

- **Decode:**
  - Interpret the instruction in terms of logical and arithmetic operations (micro-code)

- **Execute:**
  - Carry out the instruction (the micro-code)

- **Write:**
  - Write data back to memory or registers if necessary

---

**Our Simple Computer**

We did not include the instruction fetch stage
Our Simple Computer

Op-code / Operand Selection

<table>
<thead>
<tr>
<th>OP-CODE</th>
<th>OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>000</td>
</tr>
<tr>
<td></td>
<td>001</td>
</tr>
<tr>
<td>Load/Read:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Write:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>110</td>
</tr>
</tbody>
</table>

Q: What kind of instructions are these? 0-address, 1-address, 2-address, or 3-address instructions?

Fetching an Instruction

Remember the Program Counter (PC):
A register that contains the address of the next instruction

Components

RAM or ROM

Instruction Storage (program storage)

Program Counter

Adder

Why?

Components

RAM or ROM

Instruction Storage (program storage)

Program Counter

Adder

Why?
Fetching an Instruction

Decoding the Instruction
Decoding the Instructions

<table>
<thead>
<tr>
<th></th>
<th>ABC</th>
<th>WR0</th>
<th>RD0</th>
<th>WR1</th>
<th>RD1</th>
<th>WR2</th>
<th>RD2</th>
<th>WR3</th>
<th>RD3</th>
<th>S0</th>
<th>S1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>(R1) + (R2)</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R1</td>
<td>(R1) + (M)</td>
<td>001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td>(M)</td>
<td>010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Not assigned</td>
<td>011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>(M)</td>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>M</td>
<td>(R1)</td>
<td>101</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>M</td>
<td>(R2)</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Not assigned</td>
<td>111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

WRO = ABC + ABC = A(B ∨ C)

RDO = ABC + ABC + ABC = A((B ∨ C) + ABC)

And so on ......

WR1 = A ∨ BC = A(0 ∨ 1)
RD1 = A ∨ BC = 0(0 ∨ 0)
WR2 = A ∨ BC
RD2 = A ∨ BC + ABC = 1(1 ∨ 1)
WR3 = 0
RD3 = A ∨ BC
s0 = BC
s1 = ABC + BC

should be 1 for the MAR to be filled from IR
Instruction Set Architecture

Let's make up some assembly mnemonics!

- **MOV** for the load and write commands
- **ADD** for the addition commands

Example program:

1. MOV R1, [12h]
2. MOV R2, [13h]
3. ADD R1, R2
4. MOV [14h], R1

Assembler

```
0101 0010
1001 0011
0010 0000
1011 0100
```

To program memory

```
00h 0101 0010
01h 1001 0011
02h 0010 0000
03h 1011 0100
```

Designing the assembler is a different class (CS)

Programming our simple computer

On startup

/Initially (PC) = 00h
Execution of Instructions

Execution of Instructions

Note:
Both Program RAM and Data RAM latch data to their outputs at the clock's rising edge.

All registers latch data on the clock's falling edge.
Execution of Instructions

First Clock Cycle
“Rising Edge”
Execution of Instructions

Second Clock Cycle
“Rising Edge”

Program RAM or ROM

Instruction RAM (32x8)

Out

Address

IR

Decoder Circuitry (Control Unit)

RD0

Data Bus

Data

WR0

RD2

8-bit Adder

RD1

WR1

RD2

R1

R2

s0

s1

0

1

0

1

Execution of Instructions

Second Clock Cycle
“Falling Edge”

Program RAM or ROM

Instruction RAM (32x8)

Out

Address

IR

Decoder Circuitry (Control Unit)

RD0

Data Bus

Data

WR0

RD2

8-bit Adder

RD1

WR1

RD2

R1

R2

s0

s1

0

1

0

1
Execution of Instructions

Third Clock Cycle
“Rising Edge”

Data Bus

Execution of Instructions

Third Clock Cycle
“Falling Edge”
Execution of Instructions

Fourth Clock Cycle
“Rising Edge”

Fourth Clock Cycle
“Falling Edge”
Execution of Instructions

Fifth Clock Cycle
“Rising Edge”

Datapath Design

Barrel Shifters
Logic Unit
Arithmetic Unit
Register File
Function Unit

Implementation of Functions

Function Unit

Select (FS)  

Status Flags

V
C
N
Z

F

Function Unit Examples

F = x
F = x + 1
F = x + y
F = x + y + 1
F = x + \bar{y}
F = s\bar{x}

FS determines what function is being executed!
**Registers**

*Arithmetic & Logical Shift, Rotate*

The combinational shifter can be designed using 4 multiplexers

The advantage of the combinational shifter over the register is that it only takes one clock cycle to perform any shift operation.

<table>
<thead>
<tr>
<th>Shift Count</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE*</td>
<td>s₁</td>
<td>s₀</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Barrel Shifter - 4-bit

Rotate the input data bits by a number specified by the binary value of the on a set of selection lines.

<table>
<thead>
<tr>
<th>OE</th>
<th>s_1</th>
<th>s_0</th>
<th>y_3</th>
<th>y_2</th>
<th>y_1</th>
<th>y_0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Outputs high impedance</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>i_3</td>
<td>i_2</td>
<td>i_1</td>
<td>i_0</td>
<td>No rotation (pass)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i_2</td>
<td>i_1</td>
<td>i_0</td>
<td>i_3</td>
<td>Rotate left once</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i_2</td>
<td>i_0</td>
<td>i_1</td>
<td>i_3</td>
<td>Rotate left twice</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i_0</td>
<td>i_3</td>
<td>i_2</td>
<td>i_1</td>
<td>Rotate left three times</td>
</tr>
</tbody>
</table>

Logic Unit

**AND, OR, XOR, INV - 1-bit Element**

Q: How do we obtain a 4-bit or 8-bit Logic Unit?
**Arithmetic Unit**  
*Basic Element - 1-bit*

![Full Adder Diagram]

\[ S = x \oplus y \oplus C = \overline{x}y\overline{c} + \overline{x}y\overline{c} + \overline{x}y\overline{c} + \overline{x}y\overline{c} \]

\[ C_0 = (x \oplus y)c + xy = \gamma c + x + xy \]

**Function Unit**  
*Implementation of Functions*

![Function Unit Diagram]

\[ x \]
\[ y \]
\[ \text{Function Unit} \]
\[ \{ V, C, N, Z \} \]
Function Unit

Implementation

![Function Unit Implementation Diagram]

Function Unit

Example

<table>
<thead>
<tr>
<th>FS</th>
<th>RS</th>
<th>AU</th>
<th>LU</th>
<th>US</th>
<th>Micro-operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>F ← A ∧ B</td>
</tr>
<tr>
<td>0000 0100</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>F ← A + B</td>
</tr>
<tr>
<td>0000 1000</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>F ← A ◦ B</td>
</tr>
<tr>
<td>0000 1100</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>F ← ¬A</td>
</tr>
<tr>
<td>0000 0001</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>F ← A + B</td>
</tr>
<tr>
<td>0001 0001</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>F ← A - B</td>
</tr>
<tr>
<td>0010 0001</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>01</td>
<td>F ← A + 1</td>
</tr>
<tr>
<td>0011 0001</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td>F ← A</td>
</tr>
<tr>
<td>0000 0010</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>F ← sr1 A</td>
</tr>
<tr>
<td>0100 0010</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>F ← sr2 A</td>
</tr>
<tr>
<td>1000 0010</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>F ← sr3 A</td>
</tr>
<tr>
<td>1100 0010</td>
<td>11</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>F ← sr4 A</td>
</tr>
</tbody>
</table>

Q: Would you be able to make the number of control lines smaller?
Answer: Yes, there are only 12 operations; would require 4 bits; Requires some logic to decode these bits
Register File

Instead of having just two registers, a typical datapath has more than four registers. Computer with 32 or even more registers are not uncommon (PowerPC)! A set of registers having common micro-operations may be organized into a register file: a fast memory that permits one or more memory locations to be read or one or more memory locations to be written to.

Datapath

A Data
B Data

Constant in

Function
Unit

Bus A
MB

Bus B

FS
V
C
N
Z

WR
D
Address
A
Address

2\textsuperscript{m\times n}
Register File

D Data

265

266
Our Simple Computer

Data memory

RAM (32x8)

Address

8

Data

8

We did not include the instruction fetch stage

Datapath

Interface to Data Memory and I/O

Function Unit

D Data

2^m \times n

Register File

A Data

B Data

Address

0

1

Bus A

Bus B

Constant in

FS

V

C

N

Z

Data In

Data Out

MB

Address Out
to RAM or I/O

Data Bus

8

s1

s0

R1

R2

8-bit Adder

WR1 RD1

WR2 RD2

WR3 RD3

MAR

0x0

R1

R2

2mxn Register File

A Data

B Data

8-bit Adder

We did not include the instruction fetch stage