Outline

X86/Y86

ARM

Pipelining

Memory
High-Level View of Microarchitectures

CPU

- IP
- eax
- ebx
- ecx
- ZF

registers

ALU
Float
Memory

FUs

L1, L2

memory module

memory module

I/O (USB, ...)

data address control
CPUs

- Process a sequential assembler program
- Data held in registers
- Program controls which data is given to which FU, and where the result is stored
- Program controls transfer of data between registers and memory
- Caches speed up access to frequently used memory cells
Instruction Set Architectures

- These summarise the behavior of a CPU from the point of view of the programmer

- An ISA describes “what the CPU does”

- Ideally as little as possible about “how the CPU does it”
We will study two ISAs:

1. CISC: specifically the Y86  
   (academic variant of Intel’s x86)
2. RISC: specifically the ARM 32 architecture

One of the goals of this course is to understand the difference
Visible Registers

RAM

- Contains data and the program

Data registers

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>eax</td>
<td>ecx</td>
<td>edx</td>
<td>ebx</td>
<td>esp</td>
<td>ebp</td>
<td>esi</td>
<td>edi</td>
</tr>
</tbody>
</table>

Instruction Pointer (IP)

- Points to address of current instruction

Flag registers (ZF, ...)

- Store flags for branches
Y86 Assembler

- Subset of Intel’s x86 assembler

✔ You can run a Y86 program on your x86 machine!

✖ The reverse does not work in general, as too many instructions are missing (you are welcome to mend this)
Y86 Instructions

- **add/sub**: Addition/subtraction of the values in two registers; 
  ZF is set appropriately

- **RRmov**: copies value of one register into another
- **RMmov**: copies value of a register into RAM
- **MRmov**: copies value from RAM into a register

- **jnz**: Jumps to relative address if $ZF = 0$
Y86 Loads and Stores

- Loads and stores have a Displacement:
  
  $ea = esi + \text{Displacement}$

- The displacement is included in the instruction word as immediate constant

- The register $esi$ is used as offset
# Y86 Instruction Formats

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Semantics</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$RD \leftarrow RD + RS$</td>
<td><img src="image" alt="01" /> <img src="image" alt="11 RS RD" /></td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>$RD \leftarrow RD - RS$</td>
<td><img src="image" alt="29" /> <img src="image" alt="11 RS RD" /></td>
</tr>
<tr>
<td><strong>jnz</strong></td>
<td>if($\neg ZF$)&lt;br&gt;$IP \leftarrow IP + Distance$</td>
<td><img src="image" alt="75" /> Distance</td>
</tr>
<tr>
<td><strong>RRmov</strong></td>
<td>$RD \leftarrow RS$</td>
<td><img src="image" alt="89" /> <img src="image" alt="11 RS RD" /></td>
</tr>
<tr>
<td><strong>RMmov</strong></td>
<td>$MEM[ea] \leftarrow RS$</td>
<td><img src="image" alt="89" /> <img src="image" alt="11 RS 110" /> Displacement</td>
</tr>
<tr>
<td><strong>MRmov</strong></td>
<td>$RS \leftarrow MEM[ea]$</td>
<td><img src="image" alt="8b" /> <img src="image" alt="11 RS 110" /> Displacement</td>
</tr>
<tr>
<td><strong>hlt</strong></td>
<td></td>
<td><img src="image" alt="f4" /></td>
</tr>
</tbody>
</table>
Example 1

```
add eax, edx
```

- Intel convention: the **target register** is always on the left-hand side.
- The target register is a source register, too!
- Semantics:
  
  \[
  \text{eax } \leftarrow \text{eax } + \text{edx}
  \]
Example 2

\[ \text{mov edx, } [\text{BYTE one+esi}] \]

Semantics:

\[ \text{edx } \leftarrow \text{MEM[esi+17]} \]
How do Branches Work?

```c
if (a==b) {
    T;
}
else {
    F;
}
```
How do Branches Work?

```c
if (a==b) {
    T;
} else {
    F;
}
```

```assembly
mov eax, [BYTE a+esi]
mov ebx, [BYTE b+esi]
sub eax, ebx
jnz f
```

; Code for ‘T’

```assembly
mov eax, [BYTE one+esi]
add eax, eax
jnz e
```

; Code for ‘F’

```assembly
e ; ...
```
### Assembler Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembler using Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>29 F6</td>
<td>sub esi, esi</td>
</tr>
<tr>
<td>02</td>
<td>29C0</td>
<td>sub eax, eax</td>
</tr>
<tr>
<td>04</td>
<td>29DB</td>
<td>sub ebx, ebx</td>
</tr>
<tr>
<td>06</td>
<td>8B 56 17</td>
<td>mov edx, [BYTE one+esi]</td>
</tr>
<tr>
<td>09</td>
<td>01D0</td>
<td>add eax, edx</td>
</tr>
<tr>
<td>0B</td>
<td>01C3</td>
<td>add ebx, eax</td>
</tr>
<tr>
<td>0D</td>
<td>89C1</td>
<td>mov ecx, eax</td>
</tr>
<tr>
<td>0F</td>
<td>8B 56 1B</td>
<td>mov edx, [BYTE ten+esi]</td>
</tr>
<tr>
<td>12</td>
<td>29D1</td>
<td>sub ecx, edx</td>
</tr>
<tr>
<td>14</td>
<td>75 F0</td>
<td>jnz 1</td>
</tr>
<tr>
<td>16</td>
<td>F4</td>
<td>hlt</td>
</tr>
<tr>
<td>17</td>
<td>01 00 0000</td>
<td>one dd 1</td>
</tr>
<tr>
<td>1B</td>
<td>0A 00 0000</td>
<td>ten dd 10</td>
</tr>
</tbody>
</table>

The result is in ebx
The NASM Assembler

- **Windows:**
  
nasm -f win32 my_test.asm
  
link /subsystem:console /entry:start my_test.obj

- **Linux:**
  
nasm -f elf my_test.asm
  
ld -s -o my_test my_test.o

- **MacOS:**
  
nasm -f macho my_test.asm
  
ld -arch i386 -o my_test my_test.o
Inline Assembler with Visual Studio

```c
int one=1, ten=10, result;

int main() {
    __asm {
      sub esi, esi
      sub eax, eax
      sub ebx, ebx
      l: mov edx, [one+esi]
      add eax, edx
      add ebx, eax
      mov ecx, eax
      mov edx, [ten+esi]
      sub ecx, edx
      jnz l
      mov [result+esi], ebx
    }

    printf("Result: %d\n", result);
    return 0;
}
```
Debugging with GDB (Part 1)

- **run**
  Start execution

- **x/[size] Label**
  Dump a region of the memory

- **x/[sizei] Label**
  Disassemble some memory region, e.g. `x/5i $pc`

- **info registers**
  Show the value of the registers

- **step**
  Execute one instruction
Debugging with GDB (Part 2)

- `break label`
  set breakpoint at `label`

- `info break`
  show the breakpoints

- `delete breakpoints number`
  well, delete a breakpoint

- `continue`
  resume the execution after a breakpoint
Debugging with Visual Studio
Debugging with XCode
Extensions: Comparisons

We would love to have Y86 commands for

\[
\text{\textbf{if} (a < b) \{ \ldots \}}
\]

These obviously depend on the number representation:

\[
\begin{array}{c|c}
\hline
\text{with sign} & \text{without sign} \\
\hline
0 > -7 & 0 < 9 \\
\text{twoc}(0000) > \text{twoc}(1001) & \text{bin}(0000) < \text{bin}(1001) \\
\hline
\end{array}
\]

Reminder: Number Interpretation

Binary representation:

\[ bin() : \{0, 1\}^n \rightarrow \{0, \ldots, 2^n - 1\} \]

\[ bin(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i \]

Two’s complement:

\[ twoc() : \{0, 1\}^n \rightarrow \{-2^{n-1}, \ldots, 2^{n-1} - 1\} \]

\[ twoc(x) = -2^{n-1} \cdot x_{n-1} + bin(x_{n-2}, \ldots, x_0) \]
Comparing Unsigned Integers

Unsigned integers:

\[ \text{bin}(a) < \text{bin}(b) \iff \text{bin}(a) - \text{bin}(b) < 0 \]

Recall: \(-b = (\neg b) + 1\)
We get the “+1” for free by setting the carry-in of the adder.
Comparing Unsigned Integers

Unsigned integers:

\[ \text{bin}(a) < \text{bin}(b) \iff \text{bin}(a) - \text{bin}(b) < 0 \]

Recall: \(-b = (\neg b) + 1\)

We get the “+1” for free by setting the carry-in of the adder.

Let’s pretend we compute with one more bit (“zero extension”):

\[
\begin{array}{cccccc}
0 & a_{n-1} & \ldots & a_1 & a_0 \\
+ & 1 & \neg b_{n-1} & \ldots & \neg b_1 & \neg b_0 \\
= & c_n & c_{n-1} & \ldots & c_1 & 1 \\
\end{array}
\]

(carry bits)

\[
\begin{array}{cccccc}
 & s_n & s_{n-1} & \ldots & s_1 & s_0 \\
\end{array}
\]

(sum)
Comparing Unsigned Integers

Unsigned integers:

\[ \text{bin}(a) < \text{bin}(b) \iff \text{bin}(a) - \text{bin}(b) < 0 \]

Recall: \(-b = (\neg b) + 1\)
We get the “+1” for free by setting the carry-in of the adder.

Let’s pretend we compute with one more bit (“zero extension”):

\[
\begin{array}{ccccccc}
0 & a_{n-1} & \ldots & a_1 & a_0 \\
+ & 1 & \neg b_{n-1} & \ldots & \neg b_1 & \neg b_0 \\
\hline \\
& c_n & c_{n-1} & \ldots & c_1 & 1 \\
\hline \\
\end{array}
\]

(carry bits)

(\text{sum})

Thus:

\[ \text{bin}(a) - \text{bin}(b) < 0 \iff s_n \]
Comparing Unsigned Integers

Unsigned integers:

\[ \text{bin}(a) < \text{bin}(b) \iff \text{bin}(a) - \text{bin}(b) < 0 \]

Recall: \(-b = (\neg b) + 1\)

We get the “+1” for free by setting the carry-in of the adder.

Let’s pretend we compute with one more bit (“zero extension”):

\[
\begin{array}{cccccc}
0 & a_{n-1} & \ldots & a_1 & a_0 \\
+ & 1 & \neg b_{n-1} & \ldots & \neg b_1 & \neg b_0 \\
\hline
& c_n & c_{n-1} & \ldots & c_1 & 1 \\
= & s_n & s_{n-1} & \ldots & s_1 & s_0 \\
\end{array}
\]

(carry bits)

(sum)

Thus: \( \text{bin}(a) - \text{bin}(b) < 0 \iff s_n \iff \neg c_n \)
Comparing Signed Integers

Two’s complement:

$$twoc(a) < twoc(b) \iff twoc(a) - twoc(b) < 0$$

Again, let’s pretend we have an extra bit (“sign extension”):

$$\begin{array}{ccccccc}
  a_{n-1} & a_{n-1} & \ldots & a_1 & a_0 \\
+ & -b_{n-1} & -b_{n-1} & \ldots & -b_1 & -b_0 \\
= & c_n & c_{n-1} & \ldots & c_1 & 1 \\
\end{array}$$

(carry bits)

$$\begin{array}{ccccccc}
  s_n & s_{n-1} & \ldots & s_1 & s_0 \\
\end{array}$$

(sum)
Comparing Signed Integers

Two’s complement:

\[ \text{twoc}(a) < \text{twoc}(b) \iff \text{twoc}(a) - \text{twoc}(b) < 0 \]

Again, let’s pretend we have an extra bit (“sign extension”):

\[
\begin{array}{cccccc}
  & a_{n-1} & a_{n-1} & \ldots & a_1 & a_0 \\
+ & \neg b_{n-1} & \neg b_{n-1} & \ldots & \neg b_1 & \neg b_0 \\
\Rightarrow & c_n & c_{n-1} & \ldots & c_1 & 1 \\
= & s_n & s_{n-1} & \ldots & s_1 & s_0
\end{array}
\]

(carry bits) (sum)

Thus: \( \text{twoc}(a) - \text{twoc}(b) < 0 \iff s_n \)
Comparing Signed Integers

Two’s complement:

\[ \text{twoc}(a) < \text{twoc}(b) \iff \text{twoc}(a) - \text{twoc}(b) < 0 \]

Again, let’s pretend we have an extra bit (“sign extension”):

\[
\begin{array}{cccccc}
\quad & a_{n-1} & a_{n-1} & \ldots & a_1 & a_0 \\
+ & -b_{n-1} & -b_{n-1} & \ldots & -b_1 & -b_0 \\
= & c_n & c_{n-1} & \ldots & c_1 & 1 \\
\quad & s_n & s_{n-1} & \ldots & s_1 & s_0
\end{array}
\]

(carry bits) (sum)

Thus:

\[ \text{twoc}(a) - \text{twoc}(b) < 0 \iff s_n \iff a_{n-1} \oplus -b_{n-1} \oplus c_n \]
Comparing Signed Integers

Two’s complement:

\[ \text{twoc}(a) < \text{twoc}(b) \iff \text{twoc}(a) - \text{twoc}(b) < 0 \]

Again, let’s pretend we have an extra bit (“sign extension”):

\[
\begin{array}{cccccc}
  a_{n-1} & a_{n-1} & \ldots & a_1 & a_0 \\
  \neg b_{n-1} & \neg b_{n-1} & \ldots & \neg b_1 & \neg b_0 \\

  \hline
  c_n & c_{n-1} & \ldots & c_1 & 1 \\
  \hline
  s_n & s_{n-1} & \ldots & s_1 & s_0
\end{array}
\]

(carry bits)

(sum)

Thus:

\[
\begin{align*}
  \text{twoc}(a) - \text{twoc}(b) < 0 & \iff s_n \\
  a_{n-1} \oplus \neg b_{n-1} \oplus c_n & \iff s_{n-1} \oplus c_{n-1} \oplus c_n
\end{align*}
\]
New Flags: CF, SF, OF

We\textsuperscript{1} introduce three new flags for arithmetic operations:

- **CF**: The carry flag 
  \((c_n \text{ in case of additions, } \neg c_n \text{ in case of subtraction})\)

- **SF**: The sign flag \((s_{n-1})\)

- **OF**: The overflow flag \((c_n \oplus c_{n-1})\)

\textsuperscript{1}meaning Intel did so
Examples (Part 1)

\[
\begin{align*}
000 \ldots 000 & = 0 \\
+ & 000 \ldots 001 = 1 \\
0000 \ldots 000 & = 000 \ldots 001 = 1
\end{align*}
\]

\[ZF = 0, CF = 0, SF = 0, OF = 0\]
Examples (Part 1)

\[
\begin{array}{c}
000\ldots000 + 000\ldots001 = 0000\ldots000 = 0 \\
\hline
000\ldots000 = 1 \\
\hline
ZF = 0, CF = 0, SF = 0, OF = 0
\end{array}
\]

\[
\begin{array}{c}
000\ldots001 - 000\ldots001 = 1111\ldots111 = -1 \\
\hline
000\ldots000 = 0 \\
\hline
ZF = 1, CF = 0, SF = 0, OF = 0
\end{array}
\]
Examples (Part 1)

\[
\begin{align*}
000\ldots000 & = 0 \\
+ 000\ldots001 & = 1 \\
0000\ldots000 & = 000\ldots001 = 1 \\
\text{ZF} = 0, CF = 0, SF = 0, OF = 0 \\
\end{align*}
\]

\[
\begin{align*}
000\ldots001 & = 1 \\
- 000\ldots001 & = 1 \\
1111\ldots111 & = 000\ldots000 = 0 \\
\text{ZF} = 1, CF = 0, SF = 0, OF = 0 \\
\end{align*}
\]

\[
\begin{align*}
111\ldots111 & = -1 \\
+ 000\ldots010 & = 2 \\
1111\ldots110 & = 000\ldots001 = 1 \\
\text{ZF} = 0, CF = 1, SF = 0, OF = 0 \\
\end{align*}
\]
Examples (Part 2)

\[
\begin{align*}
011\ldots111 & = 2^{n-1} - 1 \\
000\ldots001 & = 1 \\
0111\ldots110 & = 2^{n-1}
\end{align*}
\]

\[\text{ZF} = 0, \text{CF} = 0, \text{SF} = 1, \text{OF} = 1\]
Examples (Part 2)

\[
\begin{align*}
011\ldots111 & \quad = 2^{n-1} - 1 \\
+ & \quad 000\ldots001 \\
0111\ldots110 & \quad = 1 \\
\hline
100\ldots000 & \quad = 2^{n-1}
\end{align*}
\]

\[\text{ZF} = 0, \text{CF} = 0, \text{SF} = 1, \text{OF} = 1\]

\[
\begin{align*}
100\ldots000 & \quad = -2^{n-1} \\
- & \quad 000\ldots001 \\
1000\ldots001 & \quad = 1 \\
\hline
011\ldots111 & \quad = 2^{n-1} - 1
\end{align*}
\]

\[\text{ZF} = 0, \text{CF} = 0, \text{SF} = 0, \text{OF} = 1\]
Branching Instructions for Comparisons

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>jz, je</td>
<td>ZF</td>
</tr>
<tr>
<td>jnz, jne</td>
<td>¬ZF</td>
</tr>
<tr>
<td>jnae, jb</td>
<td>CF</td>
</tr>
<tr>
<td>jae, jnb</td>
<td>¬CF</td>
</tr>
<tr>
<td>jna, jbe</td>
<td>CF ∨ ZF</td>
</tr>
<tr>
<td>ja, jnbe</td>
<td>¬(CF ∨ ZF)</td>
</tr>
<tr>
<td>jnge, jl</td>
<td>SF ⊕ OF</td>
</tr>
<tr>
<td>jge, jnl</td>
<td>¬(SF ⊕ OF)</td>
</tr>
<tr>
<td>jng, jle</td>
<td>((SF ⊕ OF) ∨ ZF)</td>
</tr>
<tr>
<td>jg, jnle</td>
<td>¬((SF ⊕ OF) ∨ ZF)</td>
</tr>
<tr>
<td>jmp near</td>
<td>unconditional</td>
</tr>
</tbody>
</table>

n = not, z = zero, e = equal,
g = greater, l = less, a = above, b = below

i.e. jnbe = “jump if not (below or equal)”
### Branching Instructions for Comparisons

```
sub    ax, bx
Jxxx target
...
```

**target:**

<table>
<thead>
<tr>
<th>branch if</th>
<th>with sign</th>
<th>without sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax = bx</td>
<td>je</td>
<td>je</td>
</tr>
<tr>
<td>ax ≠ bx</td>
<td>jne</td>
<td>jne</td>
</tr>
<tr>
<td>ax &gt; bx</td>
<td>jg</td>
<td>ja</td>
</tr>
<tr>
<td>ax ≥ bx</td>
<td>jge</td>
<td>jae</td>
</tr>
<tr>
<td>ax &lt; bx</td>
<td>jl</td>
<td>jb</td>
</tr>
<tr>
<td>ax ≤ bx</td>
<td>jle</td>
<td>jbe</td>
</tr>
</tbody>
</table>

Example Branching Instructions

```asm
start  sub esi, esi  ; array index
mov edx, [BYTE Intmax+esi]  ; Minimum
mov ecx, [BYTE Top+esi]  ; top index
sub ebx, ebx  ; counter

L  mov eax, ebx
sub eax, ecx
jae end  ; counter ≥ Top?

mov esi, ebx
mov edi, [BYTE Array+esi]  ; edi := array[ebx]

mov eax, edi
sub eax, edx
jge skip  ; array[ebx] ≥ Minimum?

mov edx, edi  ; Minimum := array[ebx]

skip  sub esi, esi
mov eax, [BYTE Four+esi]  ; counter += 4
add ebx, eax
```

5
10
15
20
Example Branching Instructions (Part 2)

Four    dd  4
Top     dd  40
Array   dd  1, 2, 3, 4, 5, 6, -7, 8, 9, 10
Intmax  dd  0x7fffffff
History ARM

- 1980s: Acorn Computers
- 1982: BBC Micro (8 bit)
- 1986: ARM development kit
- 1990: ARM, “Advanced RISC Machines”, founded;
  owners: Acorn Computers, Apple and VLSI Technology
Now primarily licensed as IP, with focus on low-end embedded systems and phones (>95% market share)

Built by Apple, Nvidia, Qualcomm, Samsung, TI

2013: 37 billion ARM processors produced

Early 64-bit prototypes for application in low-power servers
Visible Data

- RAM, organised in 32-bit words

- Registers
  - R0 to R15
  - R15 is a special case: this is the PC
  - R13 is the stack pointer (SP)
  - R14 is used for the return address for function calls (LR)
  - CPSR for various flags
  - (There is another register file for floating-point numbers)
## Basic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD $R_d, R_n, R_m$</td>
<td>$R_d \leftarrow R_n + R_m$</td>
</tr>
<tr>
<td>SUB $R_d, R_n, R_m$</td>
<td>$R_d \leftarrow R_n - R_m$</td>
</tr>
<tr>
<td>MUL $R_d, R_m, R_s$</td>
<td>$R_d \leftarrow (R_m \cdot R_s)[31:0]$</td>
</tr>
<tr>
<td>SMUL $R_{dL}, R_{dH}, R_m, R_s$</td>
<td>$R_{dH}, R_{dL} \leftarrow R_m \cdot R_s$</td>
</tr>
<tr>
<td>UMUL $R_{dL}, R_{dH}, R_m, R_s$</td>
<td>$R_{dH}, R_{dL} \leftarrow R_m \cdot R_s$</td>
</tr>
<tr>
<td>SDIV $R_d, R_m, R_s$</td>
<td>$R_d \leftarrow R_m / R_s$</td>
</tr>
<tr>
<td>UDIV $R_d, R_m, R_s$</td>
<td>$R_d \leftarrow R_m / R_s$</td>
</tr>
<tr>
<td>AND $R_d, R_n, R_m$</td>
<td>$R_d \leftarrow R_n &amp; R_m$</td>
</tr>
<tr>
<td>B label</td>
<td>PC $\leftarrow$ label</td>
</tr>
<tr>
<td>BL label</td>
<td>LR $\leftarrow PC + 4$; PC $\leftarrow$ label</td>
</tr>
<tr>
<td>BX $R_m$</td>
<td>BX $\leftarrow R_m$</td>
</tr>
</tbody>
</table>

Many variants!
Setting Condition Flags

- Most instructions can be given a suffix $s$.

- In addition to the usual behaviour, the condition flags (in CPSR) are updated.

\[
\begin{array}{cccc}
31 & 30 & 29 & 28 \\
N & Z & C & V \\
\end{array}
\]

N = negative, Z = zero, C = carry, V = overflow
Using Condition Flags

Most instructions can be given condition suffixes:

<table>
<thead>
<tr>
<th>EQ</th>
<th>NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS/HS</td>
<td>carry clear</td>
</tr>
<tr>
<td>MI</td>
<td>negative</td>
</tr>
<tr>
<td>VS</td>
<td>overflow</td>
</tr>
<tr>
<td>HI</td>
<td>higher</td>
</tr>
<tr>
<td>GE</td>
<td>greater or equal</td>
</tr>
<tr>
<td>GT</td>
<td>greater than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CC/LO</th>
<th>not equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>positive (or zero)</td>
</tr>
<tr>
<td>VC</td>
<td>no overflow</td>
</tr>
<tr>
<td>LS</td>
<td>lower or same</td>
</tr>
<tr>
<td>LT</td>
<td>less than</td>
</tr>
<tr>
<td>LE</td>
<td>less than or equal</td>
</tr>
</tbody>
</table>

These use 4 bits in the instruction word.
ARM Instruction Formats

ARM uses a fixed-size instruction word:

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>Opcode</td>
<td>S</td>
<td>( R_n )</td>
<td>( R_d )</td>
<td>( R_m )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data processing

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>27</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td></td>
<td>offset</td>
</tr>
</tbody>
</table>

Branch and branch&link
There is a compressed version called “Thumb-2 Instruction Set”.

The instructions have 16 bit.

Fewer options, conditions are a separate instruction.

Aimed at better I-Cache efficiency.
Sequential Processors with Pipeline

- We will start with an implementation that
  - has the form and shape of a pipeline, but
  - processes one instruction at a time
  - processes the instructions in a fixed order of phases

- These aren’t built, but only exist for illustrative purposes.

✔ But: The step to a proper pipeline is minimal (will show!)
The 5 Instruction Phases (Stages)

1. **Instruction Fetch (IF)**
The instruction is copied from the RAM into a register (IR)
The 5 Instruction Phases (Stages)

1. **Instruction Fetch (IF)**
   The instruction is copied from the RAM into a register (IR)

2. **Instruction Decode (ID)**
   Loads the values of the operands from the register file into registers A and B;
   also increments the program counter
The 5 Instruction Phases (Stages)

1. **Instruction Fetch (IF)**
   The instruction is copied from the RAM into a register (IR)

2. **Instruction Decode (ID)**
   Loads the values of the operands from the register file into registers A and B;
   also increments the program counter

3. **Execute (EX)**
   Perform any ALU operation (say add/sub),
   address arithmetic for load/store
The 5 Instruction Phases (Stages)

1. **Instruction Fetch (IF)**
   The instruction is copied from the RAM into a register (IR)

2. **Instruction Decode (ID)**
   Loads the values of the operands from the register file into registers A and B;
   also increments the program counter

3. **Execute (EX)**
   Perform any ALU operation (say \texttt{add/sub}),
   address arithmetic for load/store

4. **Memory (M)**
   RAM access for load/store
The 5 Instruction Phases (Stages)

1. **Instruction Fetch (IF)**
The instruction is copied from the RAM into a register (IR)

2. **Instruction Decode (ID)**
Loads the values of the operands from the register file into registers A and B;
also increments the program counter

3. **Execute (EX)**
Perform any ALU operation (say \texttt{add/sub}),
address arithmetic for load/store

4. **Memory (M)**
RAM access for load/store

5. **Write-Back (WB)**
Store any result in the register file
An Implementation: High-level View
Sequential Execution

- We first implement a sequential machine: The stages are processed one after the other in the order IF – ID – EX – M – WB

- We execute exactly one instruction at a time

- In contrast to multi-cycle designs: We stick to this even if an instruction doesn’t actually use a particular stage
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let \( I_1, I_2, \ldots \) be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( I_1 )</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_1$</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_2$</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_2$</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Execution

Let $I_1, I_2, \ldots$ be the sequence of instructions in program order.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$I_2$</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Processing \texttt{add}

cycle: 0

program:

\begin{verbatim}
add edx, ebx
mov [100+esi], edx
\end{verbatim}
Example: Processing add (1)

cycle: \(0\)

program:

add edx, ebx
mov [100+esi], edx
Example: Processing `add (2)`

cycle: 1

program:

`add edx, ebx`

`mov [100+esi], edx`
Example: Processing \texttt{add} (3)

cycle: \( \begin{array}{c} \text{2} \end{array} \)

program:
\[
\begin{align*}
\text{add edx, ebx} \\
\text{mov [100+esi], edx}
\end{align*}
\]
Example: Processing **add** (4)

cycle: 3

program:

`add edx, ebx`
`mov [100+esi], edx`
Example: Processing **add (5)**

cycle: 4

program:

```plaintext
add edx, ebx
mov [100+esi], edx
```

Example: Processing \texttt{RMMov}

\begin{itemize}
  \item \textbf{cycle: 5}
  \item \textbf{program:}
    \begin{itemize}
      \item \texttt{add edx, ebx}
      \item \texttt{mov [100+esi], edx}
    \end{itemize}
\end{itemize}
Example: Processing \textsc{RMMov} (1)

cycle: 5

program:
add edx, ebx
mov [100+esi], edx
Example: Processing \texttt{RMMov (2)}

cycle: \( \text{6} \)

program:
add edx, ebx
\texttt{mov} [100+esi], edx
Example: Processing **RMmov (3)**

**Cycle:** 7

**Program:**
- add edx, ebx
- mov [100+esi], edx
Example: Processing $\text{RMmov}$ (4)

cycle: 8

program:

```
add edx, ebx
mov [100+esi], edx
```

Example: Processing **RMmov (5)**

**cycle:** 9

**program:**

```
add edx, ebx
mov [100+esi], edx
```

D. Kroening: **AIMS Embedded Systems Programming MT 2016**
Example: Processing *jnz*

cycle: 0

program:

*jnz l*

the distance is 10
Example: Processing \texttt{jnz} (1)

- **Cycle:** 0
- **Program:** \texttt{jnz 1}
- **The distance is 10**
Example: Processing jnz (2)

cycle: ①

program:

jnz l

the distance is 10
Example: Processing `jnz` (3)

cycle: ②

program:

`jnz l`

the distance is 10
Example: Processing $jnz$ (4)

cycle: 3

program:

$\text{jnz } l$

the distance is 10
Example: Processing jnz (5)

cycle: ④

program:

jnz l

the distance is 10
Pipelining

- Increases the performance using the assembly-line idea

\[
\text{performance} = \underbrace{\text{instructions per cycle}}_{\text{IPC}} \cdot \underbrace{\text{clock frequency}}_{1/\tau}
\]

- Standard technique in virtually all modern circuitry (not just CPUs, but also GPUs, video, networking, wireless, ...)

## Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
### Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>$I_1$</td>
<td>$I_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td></td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EX</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
### Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>(I_1)</td>
<td>(I_2)</td>
<td>(I_3)</td>
<td>(I_4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>(I_1)</td>
<td>(I_2)</td>
<td>(I_3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>(I_1)</td>
<td>(I_2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>(I_1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
## Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td>$I_5$</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>$I_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
Pipelining

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td>$I_5$</td>
<td>$I_6$</td>
</tr>
<tr>
<td>ID</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td>$I_5$</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td>$I_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td>$I_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>$I_1$</td>
<td>$I_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best case: one instruction per cycle!
Pipelining Performance

Performance:

\[ IPC \cdot \frac{1}{\tau} \]

\[ IPC \approx 1 \]

\[ \tau \approx D_{FF} + \frac{D}{n} \]

where:
- \( IPC \): instructions per cycle
- \( \tau \): cycle time
- \( n \): # stages
- \( D \): combinational delay without the flip flops
Implementing the Pipeline: Roadmap

1. Resolving *resource conflicts*

2. Modifying the control

3. Dealing with data and control hazards
Let’s look at our sequential machine again:

Consider the C register of an ALU instruction followed by another ALU instruction!

IR once the 2nd instruction is fetched?
Register Lifetime

Problem: IR and C need to be remembered for multiple stages!

Register Lifetime

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>W</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>A, B</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAR</td>
<td></td>
<td></td>
<td>W</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>MDRw</td>
<td></td>
<td></td>
<td>W</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>W</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDRr</td>
<td></td>
<td></td>
<td></td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>eax, ..., esi, edi</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td>W</td>
</tr>
</tbody>
</table>

⚠️ Problem: IR and C need to be remembered for multiple stages!
Register Lifetime

We resolve by replication!
Register Lifetime

We resolve by replication!
Q: Which other resources are shared by stages?
Resource Conflicts

Q: Which other resources are shared by stages?
A: The system bus (shared by IF and MEM)!
Resource Conflicts

Q: Which other resources are shared by stages?
A: The system bus (shared by IF and MEM)!

Q: What do we do?
Q: Which other resources are shared by stages?
A: The system bus (shared by IF and MEM)!

Q: What do we do?
A: Most CPUs have an L1-cache that permits two (read-)accesses simultaneously.
(Really two L1 caches: an I- and a D-cache)
Example Pipeline

cycle: 0

program (modified):

add edx, ebx
mov [100+esi], ecx

Example Pipeline (1)

cycle: 0

program (modified):

```
add edx, ebx
mov [100+esi], ecx
```
Example Pipeline (2)

cycle: ①

program (modified):
add edx, ebx
mov [100+esi], ecx
Example Pipeline (3)

cycle: 2

program (modified):
add edx, ebx
mov [100+esi], ecx
Example Pipeline (4)

cycle: 3

program (modified):
add edx, ebx
mov [100+esi], ecx
Example Pipeline (5)

cycle: ④

program (modified):

- add edx, ebx
- mov [100+esi], ecx
Example Pipeline (6)

cycle: $5$

program (modified):

- `add edx, ebx`
- `mov [100+esi], ecx`
Data and Control Dependencies

Example program with data dependency:

```
add   edx,   ebx
mov   [100+esi], edx
```
Data and Control Dependencies

Example program with data dependency:

\[
\begin{align*}
\text{add} & \quad \text{edx, ebx} \\
\text{mov} [100 + \text{esi}], & \quad \text{edx}
\end{align*}
\]

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```plaintext
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>ID</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>EX</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```assembly
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>…</td>
</tr>
<tr>
<td>ID</td>
<td>…</td>
</tr>
<tr>
<td>EX</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>MEM</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>…</td>
</tr>
<tr>
<td>ID</td>
<td>…</td>
</tr>
<tr>
<td>EX</td>
<td>…</td>
</tr>
<tr>
<td>MEM</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>WB</td>
<td>add edx, ebx</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```plaintext
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>…</td>
</tr>
<tr>
<td>ID</td>
<td>…</td>
</tr>
<tr>
<td>EX</td>
<td>…</td>
</tr>
<tr>
<td>MEM</td>
<td>…</td>
</tr>
<tr>
<td>WB</td>
<td>mov [100+esi], edx</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:
Like that?

<table>
<thead>
<tr>
<th>time</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>...</td>
</tr>
<tr>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>MEM</td>
<td>...</td>
</tr>
<tr>
<td>WB</td>
<td>...</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+ esi], edx
```
Data and Control Dependencies

Example program with data dependency:

\[
\text{add edx, ebx} \\
\text{mov [100 + esi], edx}
\]

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

\[
\begin{align*}
&\text{add } edx, \text{ ebx} \\
&\text{mov } [100+esi], edx
\end{align*}
\]

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>ID</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>EX</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

DATA DEPENDENCY!

✗ We would now read the wrong (old) value of edx!
Data and Control Dependencies

Example program with data dependency:

\[
\text{add } edx, ebx \\
\text{mov [100+esi], edx}
\]

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>EX</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>MEM</td>
<td>add edx, ebx</td>
</tr>
<tr>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

DATA DEPENDENCY!

✗ We would now read the wrong (old) value of edx!
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td><code>mov [100+esi], edx</code></td>
</tr>
<tr>
<td>EX</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>MEM</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>WB</td>
<td><code>add edx, ebx</code></td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```assembly
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>EX</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>MEM</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>WB</td>
<td>BUBBLE</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```plaintext
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>...</td>
</tr>
<tr>
<td>EX</td>
<td>mov [100+esi], edx</td>
</tr>
<tr>
<td>MEM</td>
<td>BUBBLE</td>
</tr>
<tr>
<td>WB</td>
<td>BUBBLE</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

\[
\text{add edx, ebx} \\
\text{mov \([100+esi]\), edx}
\]

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>...</td>
</tr>
<tr>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>MEM</td>
<td>\text{mov ([100+esi]), edx}</td>
</tr>
<tr>
<td>WB</td>
<td>\text{BUBBLE}</td>
</tr>
</tbody>
</table>
Data and Control Dependencies

Example program with data dependency:

```
add edx, ebx
mov [100+esi], edx
```

Execution in the pipeline:

<table>
<thead>
<tr>
<th>time</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>...</td>
</tr>
<tr>
<td>ID</td>
<td>...</td>
</tr>
<tr>
<td>EX</td>
<td>...</td>
</tr>
<tr>
<td>MEM</td>
<td>...</td>
</tr>
<tr>
<td>WB</td>
<td>mov [100+esi], edx</td>
</tr>
</tbody>
</table>
Memory

- ROM: *read-only memory*

- RAM: *random-access memory*  
  (but usually means *random-access read and write memory*)

- SRAM: *static RAM*  
  stores state as long as power is supplied

- DRAM: *dynamic RAM*  
  implemented using capacitors;  
  the state is lost without periodic refresh
RAM in PCs

- 30 pin SIMM
- 72 pin SIMM
- MicroDIMM
- 184 pin RAMBus RIMM
- 100 pin DIMM
- 72 pin SODIMM
- 144 pin SDRAM SODIMM
- 200 pin DDR SODIMM
- 168 pin SDRAM DIMM
- 184 pin DDR DIMM
- 240 pin DDR-2 DIMM
Addresses

- RAM/ROM-Chips store many (billions of) bits
- Distinguish using an address
- The address is given in binary
- Plus $WE$: read/write
- The data pins are used for reading as well as writing
RAM/ROM chips are a 2D matrix

The address is split into a row and column

The binary encoding is turned into unary using a decoder
SRAM Cell with Two Inverters

- Reading and writing
- Address line selects the cell
- State is held using the inverters (latch)
- Read by comparing \( Data \) and \( \overline{Data} \)
SRAM Cell in CMOS

Address Line
VDD
GND

Data

Data
DRAM

- DRAM uses capacitors
- more simplistic and easier to build than SRAM
- high density, low cost
- But: slower!

→ fast but expensive SRAM for caches (more on that later)
→ slow but inexpensive DRAM for the main memory
Reminder: Capacitors

Store an electric charge – but only for limited time
DRAM Cell

A bit is stored as a capacity and has to be refreshed periodically.
Data Buses

Connecting multiple memory chips:

❌ No! I/O pins are expensive!
Data Buses

- Goal: effective use of the pricey wires
- Idea: share wires for data and addresses among RAM modules
Interface RAM Chips

- Control signals:
  - $CS$ (Chip Select) – activates a particular chip
  - $WE$ (Write Enable)
  - $OE$ (Output Enable)

- Inactive chips have high-impedance outputs ($Z$)

- Write by setting $\overline{WE}$, read by setting $\overline{OE}$

- Interface constraint: $\overline{OE}$ and $\overline{WE}$ are never both active
Write Cycle

\[
\begin{align*}
CS & \quad \text{valid} \\
WE & \\
OE & \\
Address & \\
Data & \text{valid}
\end{align*}
\]
Read Cycle

\( CS \)

\( WE \)

\( OE \)

Address

Data  valid
Row- und Column-Address-Strobes

- Idea: save even more wires by sending the address in two (or more) steps

- Typical: row and column are sent separately

- $\overline{RAS}$: Row Address Strobe,
  $\overline{CAS}$: Column Address Strobe
RAS/CAS Write Cycle

Address

Row

Col

\[ \overline{RAS} \]

\[ \overline{CAS} \]

\[ \overline{WE} \]

Data

valid
RAS/CAS Read Cycle

Address

Row  Col

\overline{RAS}

\overline{CAS}

\overline{WE}

Data  valid
Bus-Bursts

- RAM has long latency
  - RAM is often accessed sequentially

- Caches therefore are arranged in **lines**: a sequence of consecutive addresses (e.g. 256 bytes)

- **Bus-bursts**: efficient transmission of an entire cache line
Bus-Bursts

Address Row Col

\(\overline{RAS}\)

\(\overline{CAS}\)

\(\overline{OE}\)

\(DATA\) D0 D1 D2 D3

\(CLK\)

CAS Latency (CL)
Double Data Rate (DDR) RAM

Address

Row  Col

RAS

CAS

OE

DATA

D0 D1 D2 D3

CLK

CAS Latency (CL)
Timings

<table>
<thead>
<tr>
<th>Description</th>
<th>Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4GB 800MHz DDR2 Non-ECC CL5 (5-5-5-15) DIMM (Kit of 2)</td>
<td>KHX6400D2K2/4G</td>
</tr>
<tr>
<td>4GB 800MHz DDR2 Non-ECC Low-Latency CL4 (4-4-4-12) DIMM (Kit of 2)</td>
<td>KHX6400D2LLK2/4G</td>
</tr>
<tr>
<td>4GB 1066MHz DDR2 Non-ECC CL5 (5-5-5-15) DIMM (Kit of 2)</td>
<td>KHX8500D2K2/4G</td>
</tr>
<tr>
<td>4GB 1066MHz DDR2 Non-ECC Low-Latency CL5 (Kit of 2) Tall HS</td>
<td>KHX8500D2T1K2/4G</td>
</tr>
<tr>
<td>4GB 1066MHz DDR2 Non-ECC CL5 (5-5-5-15) DIMM (Kit of 4)</td>
<td>KHX8500D2K4/4G</td>
</tr>
<tr>
<td>8GB 800MHz DDR2 Non-ECC Low-Latency CL4 (4-4-4-12) DIMM (Kit of 4)</td>
<td>KHX6400D2LLK4/8G</td>
</tr>
</tbody>
</table>

HyperX DDR3 1375MHz, 1600MHz, 1625MHz, 1800MHz, 1866MHz and 2000MHz

<table>
<thead>
<tr>
<th>Description</th>
<th>Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1GB 1375MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX11000D3LL/1G</td>
</tr>
<tr>
<td>1GB 1600MHz DDR3 Non-ECC CL9 (9-9-9-27) DIMM</td>
<td>KHX12800D3/1G</td>
</tr>
<tr>
<td>1GB 1625MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX13000D3LL/1G</td>
</tr>
<tr>
<td>1GB 1625MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX13000AD3LL/1G</td>
</tr>
<tr>
<td>1GB 1800MHz DDR3 Non-ECC CL8 (8-8-8-24) DIMM</td>
<td>KHX14400D3/1G</td>
</tr>
<tr>
<td>1GB 1800MHz DDR3 Non-ECC CL8 (8-8-8-24) DIMM</td>
<td>KHX14400AD3/1G</td>
</tr>
<tr>
<td>2GB 1375MHz DDR3 Non-ECC CL9 (9-9-9) DIMM (Kit of 2)</td>
<td>KHX11000D3K2/2G</td>
</tr>
<tr>
<td>2GB 1375MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX11000D3LL/2G</td>
</tr>
<tr>
<td>2GB 1375MHz DDR3 Non-ECC CL7 (7-7-7-20) DIMM (Kit of 2) Intel XMP</td>
<td>KHX11000D3LLK2/2G</td>
</tr>
<tr>
<td>2GB 1375MHz DDR3 Non-ECC CL7 (7-7-7-20) DIMM (Kit of 2) Intel XMP</td>
<td>KHX112800D3/2G</td>
</tr>
<tr>
<td>2GB 1600MHz DDR3 Non-ECC CL9 (9-9-9-27) DIMM</td>
<td>KHX12800D3K2/2G</td>
</tr>
<tr>
<td>2GB 1600MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX13000D3LL/2G</td>
</tr>
<tr>
<td>2GB 1625MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX13000D3LLK2/2G</td>
</tr>
<tr>
<td>2GB 1625MHz DDR3 Non-ECC Low-Latency CL7 (7-7-7-20) DIMM</td>
<td>KHX13000D3LLK2/2GN</td>
</tr>
<tr>
<td>2GB 1625MHz DDR3 Low Latency CL8 (8-7-7-20) DIMM (Kit of 2) NVIDIA SLI</td>
<td></td>
</tr>
</tbody>
</table>
Timings

Example: 2-2-2-5

Current standard:

1. CAS Latency
2. RAS-to-CAS Delay
3. RAS Precharge
4. Act-to-Precharge Delay
Caches

- Recall: DRAM slow/cheap, SRAM fast/pricey

- Idea: use SRAM as fast cache for lots of DRAM

- “Hides” the latency of the slow DRAM

- Usually good hit rates $>90\%$
Caches: Overview

- Caches are used to store data temporarily to reduce access time to memory.
- Each cache line can store a block of data.
- The cache line index and offset determine the location of data in cache.
- If the cache line is not in the cache, a memory access is required.

Diagram:
- A cache line contains multiple cache blocks.
- Each cache block can store multiple data elements.
- The cache line index and offset determine which block is accessed.
- If a cache block is not present, the memory location is accessed.

Main memory:
- Stores data permanently.
- Accesses are required when the cache line is not present.

Cache:
- Stores data temporarily.
- Accesses are faster than main memory.
- Data is evicted from the cache when space is needed.

Tag:
- Identifies the cache block.
- Used to determine if the requested data is in the cache.

Index:
- Determines the cache block.
- Used to calculate the offset in the cache block.

Offset:
- Determines the location of the data within the cache block.
- Used to access the desired data element.
Caches: Hashing

Q: How to map the addresses?

Easiest answer: use least-significant bits

\[
\text{address} = \begin{array}{c|c|c}
\text{tag} & \text{index} & \text{offset}
\end{array}
\]

- tag: distinguishes lines with same index
- index: address in cache
- offset: distinguishes words in cache line
Collisions
Overview of Design Options for Caches

- size
- line size – number of bytes stored together
- allocation policy – when is a new entry created?
- associativity – length of list in hash table
- replacement policy – which entries to purge
- (sectoring)
- write policy – write through or write back
- split I/D cache or unified I/D cache

We will have more options once hierarchy is added.
Cache Size

- Bigger cache $\rightarrow$ better hit rate
- Bigger caches are also more expensive and have longer paths

- Partially addressed by hierarchy
  (more on that later)
Observation: memory accesses are *clustered*

- I.e., the subsequent accesses are often next to each other
- Cache entries have overhead: address bits plus flag bits
- Also remember the latency of memory!

- Reduce overhead by making cache entry bigger

- Typical size: 64 bytes (512 bits)
Associativity

- Also called “ways”

- An $n$-way cache can store $n$ entries with the same address hash

- Think of the length of the list in a hash table

- This reduces the number of collisions
Associativity

2-way cache

Cache Hierarchies

- Recall that fast SRAM is expensive, and bigger caches have long paths.

- Thus: build a cache for the cache.

- L1: closest to CPU
- L2, L3, L4: cache the next level

- Caches get bigger the closer they get to the memory.
<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>L4 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>80486DX</td>
<td>1989</td>
<td>8 KB joint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>8 KB+8 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>8 KB+8 KB</td>
<td></td>
<td>0.25 MB</td>
<td></td>
</tr>
<tr>
<td>Pentium MMX</td>
<td>1997</td>
<td>16 KB+16 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>16 KB+16 KB</td>
<td></td>
<td>0.5 MB</td>
<td></td>
</tr>
<tr>
<td>Xeon</td>
<td>1998</td>
<td>8 KB+8 KB</td>
<td></td>
<td>0.25–1 MB</td>
<td></td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>16 KB+16 KB</td>
<td></td>
<td>0.5 MB</td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>16 KB+16 KB</td>
<td></td>
<td>0.25–0.5 MB</td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>16 KB+16 KB</td>
<td></td>
<td>1.5–9 MB</td>
<td>2 or 4 MB</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2003</td>
<td>32 KB+32 KB</td>
<td></td>
<td>0.25 MB</td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2006</td>
<td>32 KB+32 KB</td>
<td></td>
<td>2 MB</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>32 KB+32 KB</td>
<td></td>
<td>0.25 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Core i5</td>
<td>2009</td>
<td>32 KB+32 KB</td>
<td></td>
<td>0.25 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Core i3</td>
<td>2010</td>
<td>32 KB+32 KB</td>
<td></td>
<td>0.25 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td>Atom SoC</td>
<td>2012</td>
<td>32 KB+24 KB</td>
<td></td>
<td>0.25 MB</td>
<td></td>
</tr>
<tr>
<td>Core M</td>
<td>2014</td>
<td></td>
<td>0.25 MB</td>
<td>3 MB</td>
<td>128 MB</td>
</tr>
</tbody>
</table>

Numbers are *per core* unless shared.