High-Level View of Microarchitectures

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
  - Index: 0 1 2 3 4 5 6 7
  - Name: eax ecx edx ebx esp ebp esi edi
  - 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: eax = esi + 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>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>RD→RD+RS</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>RD→RD+RS</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jnz</td>
<td>if(ZF)IP+Distance</td>
<td>75</td>
<td>Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRmov</td>
<td>RD→RS</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMmov</td>
<td>MEM[ea]→RS</td>
<td>85</td>
<td>Displacement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRmov</td>
<td>RS→MEM[ea]</td>
<td>8b</td>
<td>Displacement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hlt</td>
<td></td>
<td>f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 1

```plaintext
add eax, edx
```

- Intel convention: the target register is always on the left-hand side
- The target register is a source register, too!
- Semantics: 
  ```plaintext
eax ← eax + edx
  ```
Example 2

```assembly
mov edx, [BYTE one+esi]
8B 56 17
```

Opcode (MRmov)

```
Displacement
01 010 edx
```

Semantics:

```
edx ← MEM[esi+17]
```

D. Kroening: AIMS Embedded Systems Programming MT 2015 13

How do Branches Work?

```assembly
if (a==b) {
T ;
}
else {
5 F ;
}
→
mov eax, [BYTE a+esi]
mov ebx, [BYTE b+esi]
sub eax, ebx
jnz f
```

```assembly
else {
5 F ;
}
```

```assembly
test eax, eax
jnz e
```

```
; Code for 'F'
```

D. Kroening: AIMS Embedded Systems Programming MT 2015 14

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>29F6</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>8B56 17</td>
<td>mov edx, [BYTE one+esi]</td>
</tr>
<tr>
<td>08</td>
<td>01D0</td>
<td>add eax, edx</td>
</tr>
<tr>
<td>08</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>8B56 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>75F0</td>
<td>jnz l</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

D. Kroening: AIMS Embedded Systems Programming MT 2015 15

Inline Assembler with Visual Studio

```c
int one=1, ten =10, result;
int main() {
    asm {
        "sub esi, esi"
        "sub eax, eax"
        "sub ebx, ebx"
        "mov edx, [BYTE one+esi]"
        "add eax, edx"
        "add ebx, eax"
        jnz l
        "mov edx, [BYTE ten+esi]"
        "add ecx, edx"
        l:
        "mov [result+esi], ebx"
    }
    printf("Result :%d\n", result);
    return 0;
}
```

D. Kroening: AIMS Embedded Systems Programming MT 2015 17

Debugging with GDB (Part 1)

- run
  Start execution
- x/size Label
  Dump a region of the memory
- x/size Label
  Disassemble some memory region, e.g. x/5i $pc
- info registers
  Show the value of the registers
- step
  Execute one instruction

D. Kroening: AIMS Embedded Systems Programming MT 2015 18
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

Extensions: Comparisons

We would love to have Y86 commands for

```plaintext
if( a<b) { ... }
```

These obviously depend on the number representation:

<table>
<thead>
<tr>
<th>with sign</th>
<th>without sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &gt; −7</td>
<td>0 &lt; 9</td>
</tr>
<tr>
<td>twoc(0000) &gt; twoc(1001)</td>
<td>bin(0000) &lt; bin(1001)</td>
</tr>
</tbody>
</table>

Reminder: Number Interpretation

Binary representation:

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

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

Two’s complement:

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

```plaintext
twoc(x) = -2^{n-1} \cdot x_{n-1} + bin(x_{n-2}, \ldots, x_0)
```

Comparing Unsigned Integers

Unsigned integers:

```plaintext
bin(a) < bin(b) \iff bin(a) − bin(b) < 0
```

Recall: 

```plaintext
−b = (−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”):

```plaintext
\begin{align*}
0 & a_{n-1} \ldots a_1 a_0 \\
+ 1 & -b_{n-1} \ldots -b_1 -b_0 \\
\rightarrow & c_n c_{n-1} \ldots c_1 1 \quad (carry \ bits) \\
\text{sum} & = s_n s_{n-1} \ldots s_1 s_0
\end{align*}
```

Thus:

```plaintext
bin(a) − bin(b) < 0 \iff s_n \iff −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"):

\[
\begin{align*}
    a_{n-1} & \quad a_{n-2} \quad \ldots \quad a_{1} \quad a_{0} \\
    + & \quad \sim b_{n-1} \quad \sim b_{n-2} \quad \ldots \quad \sim b_{1} \quad \sim b_{0} \\
    = & \quad s_{n} \quad s_{n-1} \quad \ldots \quad s_{1} \quad 1 \quad \text{(carry bits)} \\
        & \quad s_{n-1} \quad \ldots \quad s_{1} \quad 0 \quad \text{(sum)}
\end{align*}
\]

Thus: \( \text{twoc}(a) - \text{twoc}(b) < 0 \iff s_{n} \iff a_{n-1} \oplus \sim b_{n-1} \oplus c_{n} \iff s_{n-1} \oplus c_{n-1} \oplus c_{n} \)

New Flags: CF, SF, OF

We\(^1\) introduce three new flags for arithmetic operations:

- **CF**: The carry flag (\(c_{n}\) in case of additions, \(\neg c_{n}\) in case of subtraction)
- **SF**: The sign flag (\(s_{n-1}\))
- **OF**: The overflow flag (\(c_{n} \oplus c_{n-1}\))

\(^1\)meaning Intel did so

Examples (Part 1)

\[
\begin{align*}
    000 \ldots 000 & \quad = 0 \\
    + & \quad 000 \ldots 001 = 1 \\
    \text{ZF} = 0, \text{CF} = 0, \text{SF} = 0, \text{OF} = 0 \\
    000 \ldots 000 & \quad = 1 \\
    000 \ldots 001 & \quad = 1 \\
    000 \ldots 000 & \quad = 0 \\
    111 \ldots 111 & \quad = -1 \\
    + & \quad 000 \ldots 010 = 2 \\
    111 \ldots 110 & \quad = 2^{n-1} - 1
\end{align*}
\]

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>jns, jne</td>
<td>\neg ZF</td>
</tr>
<tr>
<td>jnae, jb</td>
<td>CF</td>
</tr>
<tr>
<td>jae, jab</td>
<td>\neg CF</td>
</tr>
<tr>
<td>jna, jbe</td>
<td>CF \lor ZF</td>
</tr>
<tr>
<td>ja, jabe</td>
<td>\neg (CF \lor ZF)</td>
</tr>
<tr>
<td>jnge, jl</td>
<td>SF \lor OF</td>
</tr>
<tr>
<td>jge, jnl</td>
<td>\neg (SF \lor OF)</td>
</tr>
<tr>
<td>jng, jle</td>
<td>\neg ((SF \lor OF) \lor 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. jabe = "jump if not (below or equal)"

Examples (Part 2)

\[
\begin{align*}
    011 \ldots 111 & \quad = 2^{n-1} - 1 \\
    + & \quad 000 \ldots 001 = 1 \\
    \text{ZF} = 0, \text{CF} = 0, \text{SF} = 0, \text{OF} = 1 \\
    011 \ldots 110 & \quad = 2^{n-1} \\
    100 \ldots 000 & \quad = -2^{n-1} \\
    - & \quad 000 \ldots 001 = 1 \\
    100 \ldots 000 & \quad = 2^{n-1} - 1
\end{align*}
\]

Branching Instructions for Comparisons

```
sub ax, bx
Jxxx target
...
branch if with sign without sign
ax = bx je je
ax \neq bx jne jne
ax > bx jg ja
ax \geq bx jge jae
ax < bx jl jb
ax \leq bx jle jbe
```
Example Branching Instructions

```
start sub esi, esi ; array index
mov edx, [BYTE Intmax+esi] ; Minimum
mov ecx, [BYTE Top+esi] ; top index
sub ebx, ebx
L mov eax, ebx
sub eax, ecx
jge skip

mov esi, ebx
mov edi, [BYTE Array+esi] ; edi := array[esi]
mov eax, edi
sub eax, edx
jge skip

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

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

ARM Today

- 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

```
ADD Rd, Rn, Rm Rd ← Rn + Rm
SUB Rd, Rn, Rm Rd ← Rn − Rm
MUL Rd, Rm, Rs Rd ← (Rm · Rs)[31 : 0]
SMUL Rd, Rm, Rs Rd ← (Rm · Rs)[31 : 0]
```

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>CS/HS</th>
<th>NE</th>
<th>CC/LO</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>carry set</td>
<td>not equal</td>
<td>carry clear</td>
</tr>
<tr>
<td>N</td>
<td>negative</td>
<td>NE</td>
<td>positive (or zero)</td>
</tr>
<tr>
<td>Z</td>
<td>overflow</td>
<td>N</td>
<td>no overflow</td>
</tr>
<tr>
<td>C</td>
<td>higher</td>
<td>N</td>
<td>lower or same</td>
</tr>
<tr>
<td>V</td>
<td>greater or equal</td>
<td>N</td>
<td>less than</td>
</tr>
<tr>
<td>GT</td>
<td>greater than</td>
<td>N</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:

31 28 27 26 25 24 23 22 21 20 19 15 12 11 0

Cond Opcode S Rn Rd Rdn data processing

31 28 27 25 24 23 0

Cond 1 0 1 L offset branch and branch&link

ARM Instruction Formats

▶ 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)

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

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

5. **Write-Back (WB)**
   Store any result in the register file
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

Example: Processing add

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

<table>
<thead>
<tr>
<th>time</th>
<th>0 1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_1$ $I_2$</td>
</tr>
<tr>
<td>ID</td>
<td>$I_1$ $I_2$</td>
</tr>
<tr>
<td>EX</td>
<td>$I_1$ $I_2$</td>
</tr>
<tr>
<td>MEM</td>
<td>$I_1$ $I_2$</td>
</tr>
<tr>
<td>WB</td>
<td>$I_1$</td>
</tr>
</tbody>
</table>

Example: Processing add (1)

program:
- add edx, ebx
- mov [100+esi], edx

Example: Processing add (2)

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

- **Cycle:** 2
- **Program:**
  - `add edx, ebx`
  - `mov [100+esi], edx`

Example: Processing \texttt{add (4)}

- **Cycle:** 3
- **Program:**
  - `add edx, ebx`
  - `mov [100+esi], edx`

Example: Processing \texttt{add (5)}

- **Cycle:** 4
- **Program:**
  - `add edx, ebx`
  - `mov [100+esi], edx`

Example: Processing \texttt{RMmov (1)}

- **Cycle:** 5
- **Program:**
  - `add edx, ebx`
  - `mov [100+esi], edx`

Example: Processing \texttt{RMmov (2)}

- **Cycle:** 6
- **Program:**
  - `add edx, ebx`
  - `mov [100+esi], edx`
Example: Processing RMmov (3)

cycle: 7
program:
add edx, ebx
mov [100+esi], edx

Example: Processing 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

Example: Processing jnz

cycle: 7
program:
jnz l
the distance is 10

Example: Processing jnz (1)

cycle: 8
program:
jnz l
the distance is 10

Example: Processing jnz (2)

cycle: 9
program:
jnz l
the distance is 10
Example: Processing jnz (3)

Example: Processing jnz (4)

Example: Processing jnz (5)

Pipelining

Pipelining Performance

\[
IPC \cdot \frac{1}{\tau} \\
\tau \approx \frac{1}{D_{FF}} + \frac{D}{n}
\]

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

Resource Conflicts

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!

Resource Conflicts

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

Example Pipeline

cycle:

program (modified):

mov [100+esi], ecx

A, B

MAR C3 Flg MDRw
MDRr

We resolve by replication!
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
```

Execution in the pipeline:
```
time 3
IF ... 
ID ... 
EX mov [100+esi], edx
MEM add edx, ebx
WB
```

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

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

Structure

- 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 Data

SRAM Cell in CMOS

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:

CPU

memory module

memory module

memory module

× 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

![Memory Module Diagram]

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 WE, read by setting OE
- Interface constraint: OE and WE are never both active

Write Cycle

- CS
- WE
- OE
- Address
- Data

Read Cycle

- CS
- WE
- OE
- Address
- Data

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
- RAS: Row Address Strobe, CAS: Column Address Strobe

RAS/CAS Write Cycle

- Address
- Row
- Col
- RAS
- CAS
- WE
- Data

valid
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CAS Latency (CL)

### Double Data Rate (DDR) RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Row</th>
<th>Col</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CAS Latency (CL)

### Timings

**Example: 2-2-2-5**

- CAS Latency
- RAS-to-CAS Delay
- RAS Precharge
- 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

- 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

- Bigger cache → better hit rate
- Bigger caches are also more expensive and have longer paths

Overview of Design Options for Caches

- Cache Size
  - Bigger cache → better hit rate
  - Bigger caches are also more expensive and have longer paths

We will have more options once hierarchy is added.

Caches: Hashing

Q: How to map the addresses?

Easiest answer: use least-significant bits

\[
\text{address} = \text{tag} \ | \ \text{index} \ | \ \text{offset}
\]

- Tag: distinguishes lines with same index
- Index: address in cache
- Offset: distinguishes words in cache line

Collisions

- Collisions occur when multiple addresses map to the same location.
- Various strategies exist to resolve these collisions, such as chaining or open addressing.

- Partially addressed by hierarchy (more on that later)
Observation: memory accesses are \textit{clustered}.
\begin{itemize}
\item I.e., the subsequent accesses are often next to each other.
\item Cache entries have overhead: address bits plus flag bits.
\item Also remember the latency of memory!
\end{itemize}

- Reduce overhead by making cache entry bigger.
- Typical size: 64 bytes (512 bits).

\begin{itemize}
\item Also called \textit{"ways"}
\item An \( n \)-way cache can store \( n \) entries with the same address hash.
\item Think of the length of the list in a hash table.
\item This reduces the number of collisions.
\end{itemize}

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>0.25 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>16 KB+16 KB</td>
<td>0.5 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon</td>
<td>1998</td>
<td>8 KB+8 KB</td>
<td>0.25–1 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>16 KB+16 KB</td>
<td>0.5 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>16 KB+16 KB</td>
<td>0.25–0.5 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Itanium 2</td>
<td>2002</td>
<td>16 KB+16 KB</td>
<td>1.5–9 MB</td>
<td>2 or 4 MB</td>
<td></td>
</tr>
<tr>
<td>core i7</td>
<td>2008</td>
<td>32 KB+32 KB</td>
<td>0.25 MB</td>
<td>8 MB</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>32 KB+32 KB</td>
<td>0.25 MB</td>
<td>8 MB</td>
<td></td>
</tr>
<tr>
<td>Core i5</td>
<td>2009</td>
<td>32 KB+32 KB</td>
<td>0.25 MB</td>
<td>8 MB</td>
<td></td>
</tr>
<tr>
<td>Core i3</td>
<td>2010</td>
<td>32 KB+32 KB</td>
<td>0.25 MB</td>
<td>8 MB</td>
<td></td>
</tr>
<tr>
<td>Atom SoC</td>
<td>2012</td>
<td>32 KB+24 KB</td>
<td>0.25 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\begin{itemize}
\item Pentium:
- M (2 KB)
- MMX (4 KB)
- Pro (8 KB)
- II (8 KB)
\item Xeon:
- 0.5 MB
- 1 MB
- 2 MB
\item Itanium:
- 2 MB
- 4 MB
\item Core:
- 2 MB
- 4 MB
\item Atom SoC:
- 0.5 MB
\end{itemize}

Numbers are per core unless shared.