



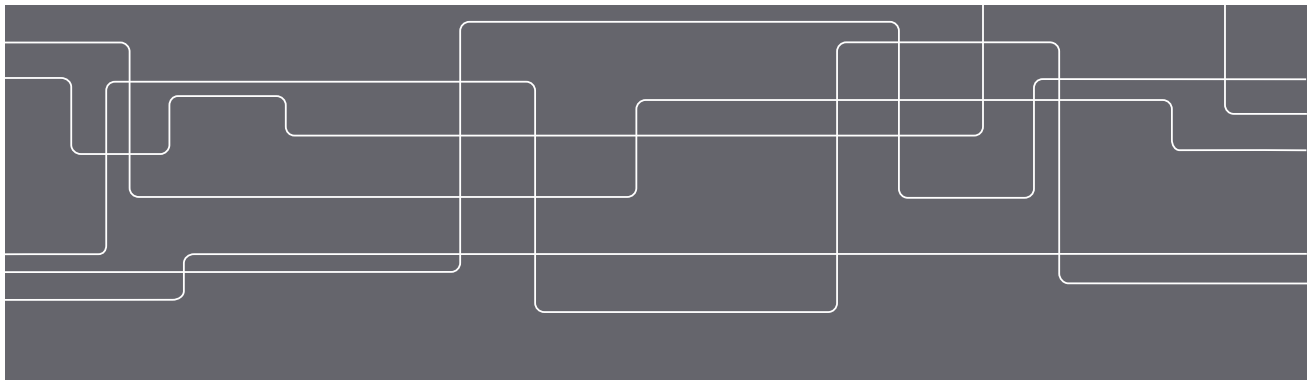
Introduction to Computer Architecture

PDC Summer School

August 20, 2019

David Broman

Associate Professor, KTH Royal Institute of Technology



2



Different Kinds of Computer Systems



**Embedded
Real-Time Systems**



Photo by Kyro

**Personal Computers and
Personal Mobile Devices**



Photo by Robert Harker

**Warehouse
Scale Computers**

Dependability

Energy

Performance

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

How is this computer revolution possible?



Moore's law:

- Integrated circuit resources (transistors) double every 18-24 months.
- By Gordon E. Moore, Intel's co-founder, 1960s.
- Possible because of refined manufacturing processes. E.g., Intel Core i7-6800 processors uses 14nm manufacturing.
- Sometimes considered a *self-fulfilling prophecy*. Served as a goal for the semiconductor industry.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

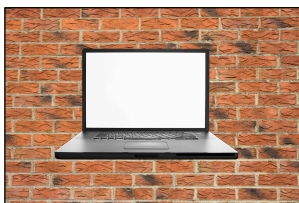
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Have we reached the limit?

Why?

The Power Wall



<http://www.publicdomainpictures.net/view-image.php?image=1261&picture=tegelvagg>

**Increased clock rate
implies increased power**

We cannot cool the system enough to
increase the clock rate anymore...



During the last decade, the clock rate has increased dramatically.

- 1989: 80486, 25MHz
- 1993: Pentium, 66Mhz
- 1997: Pentium Pro, 200MHz
- 2001: Pentium 4, 2.0 GHz
- 2004: Pentium 4, 3.6 GHz

2019: Intel Xeon W, 3.2 GHz, 8 Cores
(Turbo 4.2Ghz)

"New" trend since 2006: Multicore

- Moore's law still holds (but will end soon)
- More processors on a chip: multicore
- **"New" challenge: parallel programming**

David Broman
dbro@kth.se

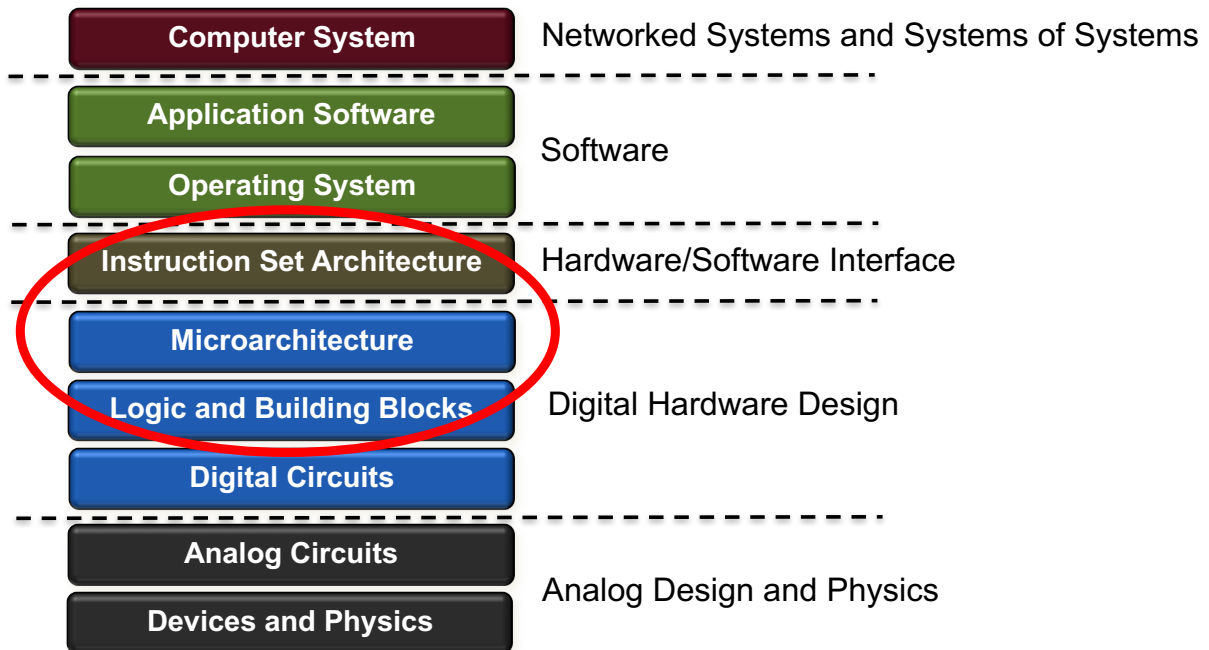
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Abstractions in Computer Systems



David Broman
dbro@kth.se

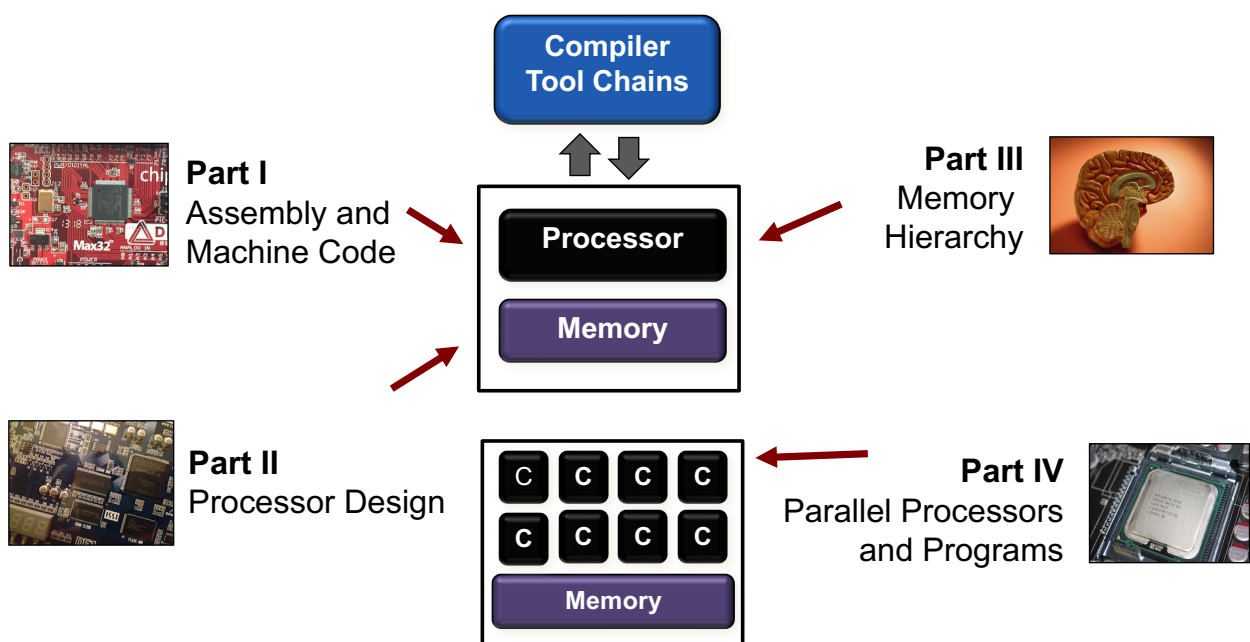
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

This Course Module in one Slide



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

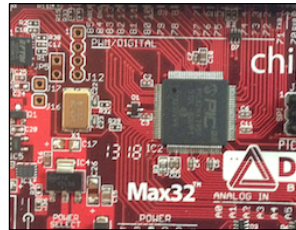
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part I

Assembly and Machine Code



Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

David Broman
dbro@kth.se



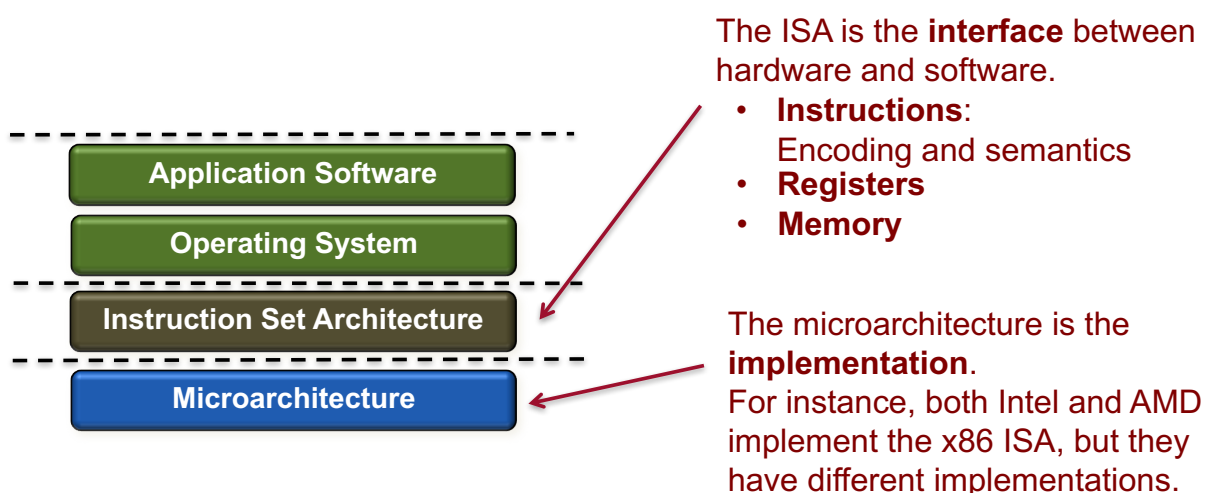
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

The Instruction Set Architecture (ISA) and its Surrounding



David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

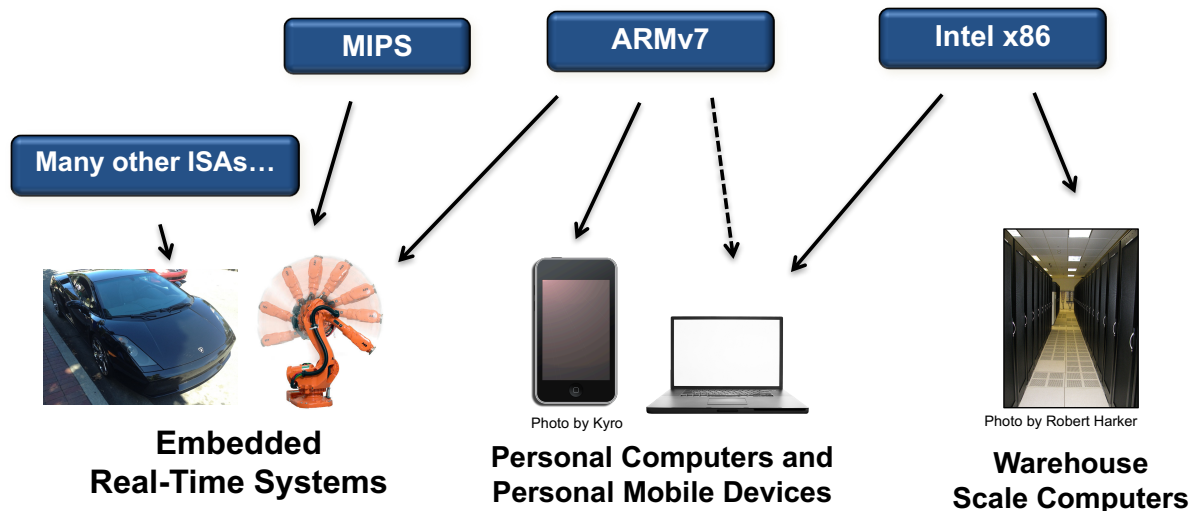
Part IV
Parallel Processors
and Programs

Different ISAs

MIPS is the focus in this course because

- i) it is relatively easy to understand
- ii) most text books focus on MIPS.

We will only briefly compare with ARM and x86, but they are complex...



David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Instructions (1/2) CISC vs. RISC

Each ISA has a set of instructions. Two main categories:

Complex Instruction Set Computers (CISC)

- Many special purpose instructions.
- Example: **x86**. Now almost 900 instructions.
- Typically various encoding lengths (x86, 1-15 bytes)
- Different number of clock cycles, depending on instruction.

Reduced Instruction Set Computers (RISC)

- Few, regular instructions. Minimize hardware complexity.
- **MIPS** is a good example (ARM mostly RISC)
- Typically fixed instruction lengths (e.g., 4 bytes for MIPS)
- Typically one clock cycle per instruction (excluding memory accesses and cache misses)

David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Instructions (2/2)

C code, Assembly Code, and Machine Code

C Code

```
a = b + c;
```

The compiler maps (if possible) C variables to **registers** (small fast memory locations)

For instance, **a** to **\$s0**, **b** to **\$s1**, and **c** to **\$s2**

MIPS Assembly Code

```
add $s0, $s1, $s2
```

The assembly code is in human readable form of the machine code

MIPS Machine Code

```
0x02328020
```

Each assembly instruction is mapped to one or more machine code instructions.
In MIPS, each instruction is 32 bits.

David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Registers

| Name | Number | Use |
|-----------|--------|--------------------------------|
| \$0 | 0 | constant value of 0 |
| \$at | 1 | assembler temporary |
| \$v0-\$v1 | 2-3 | function return value |
| \$a0-\$a3 | 4-7 | function arguments |
| \$t0-\$t7 | 8-15 | temporary (caller-saved) |
| \$s0-\$s7 | 16-23 | saved variables (callee-saved) |
| \$t8-\$t9 | 24-25 | temporary (caller-saved) |
| \$k0-\$k1 | 26-27 | reserved for OS kernel |
| \$gp | 28 | global pointer |
| \$sp | 29 | stack pointer |
| \$fp | 30 | frame pointer |
| \$ra | 31 | function return address |

David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Memory

Big problem if 32 registers set the limit of the number of variables in a program. Solution: memory.

Word address

| | | | | | |
|--------------|----|----|----|----|--------|
| 0000 000C | 0f | a0 | b0 | 12 | Word 3 |
| 0000 0008 | 44 | 93 | 4e | aa | Word 2 |
| 0000 0004 | 33 | fa | 01 | 23 | Word 1 |
| 0000 0000 | 21 | a0 | 1b | 33 | Word 0 |
| byte address | 0 | 1 | 2 | 3 | |

Memory

- Has many more data locations than registers.
- Accessing memory is slower than accessing registers.



David Broman
dbro@kth.se

Part I

Assembly and Machine Code

Part II

Processor Design

Part III

Memory Hierarchy

Part IV

Parallel Processors and Programs



MIPS Reference Sheet

MIPS Reference Sheet

David Brown, KTH Royal Institute of Technology
Version 1.1.0, January 20, 2016

INSTRUCTIONS (SUBSET)

| Name (format, op, funct) | Opcode | Operation |
|---------------------------------------|--|--|
| add (\$Rd, \$Rt, \$Rn) | add, <i>rs</i> , <i>rt</i> , <i>rd</i> , <i>imm</i> | $reg[rd] = reg[rt] + reg[rs]$ |
| add immediate (\$Rd, \$Rt, \$imm) | addi, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = reg[rt] + \text{sign-extended}(imm)$ |
| addu (\$Rd, \$Rt, \$Rn) | addu, <i>rs</i> , <i>rt</i> , <i>rd</i> , <i>imm</i> | $reg[rd] = reg[rt] + reg[rs]$ (no overflow) |
| addu immediate (\$Rd, \$Rt, \$imm) | addui, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = reg[rt] + \text{sign-extended}(imm)$ (no overflow) |
| and (\$Rd, \$Rt, \$Rn) | and, <i>rs</i> , <i>rt</i> , <i>rd</i> , <i>imm</i> | $reg[rd] = reg[rt] \& reg[rs]$ |
| and immediate (\$Rd, \$Rt, \$imm) | andi, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = reg[rt] \& \text{sign-extended}(imm)$ |
| andn (\$Rd, \$Rt, \$Rn) | andn, <i>rs</i> , <i>rt</i> , <i>rd</i> , <i>imm</i> | $reg[rd] = reg[rt] \& \sim reg[rs]$ |
| andn immediate (\$Rd, \$Rt, \$imm) | andni, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = reg[rt] \& \sim \text{sign-extended}(imm)$ |
| beq (\$Rt, \$Rf, \$Rn) | beq, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] == reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| beq immediate (\$Rt, \$Rf, \$imm) | beqi, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] == reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bne (\$Rt, \$Rf, \$Rn) | bne, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] != reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bne immediate (\$Rt, \$Rf, \$imm) | bneci, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] != reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| blt (\$Rt, \$Rf, \$Rn) | blt, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] < reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| blt immediate (\$Rt, \$Rf, \$imm) | blti, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] < reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bltu (\$Rt, \$Rf, \$Rn) | bltu, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] < \text{unsign}(reg[rt]) ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bltu immediate (\$Rt, \$Rf, \$imm) | bltui, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] < \text{unsign}(reg[rt]) ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bgt (\$Rt, \$Rf, \$Rn) | bgt, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] > reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bgt immediate (\$Rt, \$Rf, \$imm) | bgti, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] > reg[rt] ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bgtu (\$Rt, \$Rf, \$Rn) | bgtu, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] > \text{unsign}(reg[rt]) ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| bgtu immediate (\$Rt, \$Rf, \$imm) | bgtui, <i>rs</i> , <i>rt</i> , <i>label</i> | $reg[rs] > \text{unsign}(reg[rt]) ? \text{PC} = \text{PC} + 4 : \text{PC} = \text{PC} + \text{offset} \& 0x1f$ |
| lbu (\$Rt, \$Rn) | lbu, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu immediate (\$Rt, \$Rn, \$imm) | lbu, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu16 (\$Rt, \$Rn) | lbu16, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu16 immediate (\$Rt, \$Rn, \$imm) | lbu16, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu32 (\$Rt, \$Rn) | lbu32, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu32 immediate (\$Rt, \$Rn, \$imm) | lbu32, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu64 (\$Rt, \$Rn) | lbu64, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu64 immediate (\$Rt, \$Rn, \$imm) | lbu64, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu128 (\$Rt, \$Rn) | lbu128, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu128 immediate (\$Rt, \$Rn, \$imm) | lbu128, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu256 (\$Rt, \$Rn) | lbu256, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu256 immediate (\$Rt, \$Rn, \$imm) | lbu256, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu512 (\$Rt, \$Rn) | lbu512, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu512 immediate (\$Rt, \$Rn, \$imm) | lbu512, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu1024 (\$Rt, \$Rn) | lbu1024, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu1024 immediate (\$Rt, \$Rn, \$imm) | lbu1024, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu2048 (\$Rt, \$Rn) | lbu2048, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu2048 immediate (\$Rt, \$Rn, \$imm) | lbu2048, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu4096 (\$Rt, \$Rn) | lbu4096, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu4096 immediate (\$Rt, \$Rn, \$imm) | lbu4096, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu8192 (\$Rt, \$Rn) | lbu8192, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu8192 immediate (\$Rt, \$Rn, \$imm) | lbu8192, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu16384 (\$Rt, \$Rn) | lbu16384, <i>rs</i> , <i>rt</i> , <i>imm</i> | $reg[rt] = \text{unsign}(reg[rs])$ |
| lbu1 | | |

PAUSE INSTRUCTIONS (SUBSET)

| Name | Example | Equivalent Base Instruction |
|---------------------------|---|---|
| nop | nop, <i>rs</i> , <i>rt</i> , <i>imm</i> | nop, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| sleep | sleep, <i>rs</i> , <i>rt</i> , <i>imm</i> | sleep, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| wait | wait, <i>rs</i> , <i>rt</i> , <i>imm</i> | wait, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| wait immediate | wait, <i>rs</i> , <i>rt</i> , <i>imm</i> | wait, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn | waitn, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn immediate | waitn, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16 | waitn16, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16 immediate | waitn16, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn32 | waitn32, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn32, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn32 immediate | waitn32, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn32, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn64 | waitn64, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn64, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn64 immediate | waitn64, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn64, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn128 | waitn128, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn128, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn128 immediate | waitn128, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn128, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn256 | waitn256, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn256, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn256 immediate | waitn256, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn256, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn512 | waitn512, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn512, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn512 immediate | waitn512, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn512, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1024 | waitn1024, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1024, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1024 immediate | waitn1024, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1024, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2048 | waitn2048, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2048, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2048 immediate | waitn2048, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2048, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4096 | waitn4096, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn4096, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4096 immediate | waitn4096, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn4096, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn8192 | waitn8192, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn8192, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn8192 immediate | waitn8192, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn8192, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16384 | waitn16384, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16384, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16384 immediate | waitn16384, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16384, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn32768 | waitn32768, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn32768, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn32768 immediate | waitn32768, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn32768, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn65536 | waitn65536, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn65536, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn65536 immediate | waitn65536, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn65536, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn131072 | waitn131072, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn131072, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn131072 immediate | waitn131072, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn131072, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn262144 | waitn262144, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn262144, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn262144 immediate | waitn262144, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn262144, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn524288 | waitn524288, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn524288, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn524288 immediate | waitn524288, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn524288, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1048576 | waitn1048576, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1048576, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1048576 immediate | waitn1048576, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1048576, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2097152 | waitn2097152, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2097152, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2097152 immediate | waitn2097152, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2097152, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4194304 | waitn4194304, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn4194304, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4194304 immediate | waitn4194304, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn4194304, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn8388608 | waitn8388608, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn8388608, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn8388608 immediate | waitn8388608, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn8388608, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16777216 | waitn16777216, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16777216, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn16777216 immediate | waitn16777216, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn16777216, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn33554432 | waitn33554432, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn33554432, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn33554432 immediate | waitn33554432, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn33554432, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn67108864 | waitn67108864, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn67108864, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn67108864 immediate | waitn67108864, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn67108864, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn134217728 | waitn134217728, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn134217728, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn134217728 immediate | waitn134217728, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn134217728, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn268435456 | waitn268435456, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn268435456, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn268435456 immediate | waitn268435456, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn268435456, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn536870912 | waitn536870912, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn536870912, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn536870912 immediate | waitn536870912, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn536870912, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1073741824 | waitn1073741824, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1073741824, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn1073741824 immediate | waitn1073741824, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn1073741824, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2147483648 | waitn2147483648, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2147483648, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn2147483648 immediate | waitn2147483648, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn2147483648, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4294967296 | waitn4294967296, <i>rs</i> , <i>rt</i> , <i>imm</i> | waitn4294967296, <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> or <i>rs</i> , <i>rt</i> , <i>imm</i> |
| waitn4294967296 immediate | waitn4294967 | |

- Summarizes an important **subset of the MIPS instructions** and their coding.



David Broman
dbro@kth.se

Part I

Assembly and Machine Code

Part II

Processor Design

Part III

Memory Hierarchy

Part IV

Parallel Processors and Programs

Conditional Branches (1/2)

beq and bne

Branch if equal (beq) branches if two operands have equal values.

Branch if not equal (bne) branches if two operands do not have equal values.

```
addi $s0, $0, 4
xori $s1, $s0, 1
sll $t0, $s1, 1
beq $t0, $s0, foo
add $s1, $s1, $s0
foo:
add $s5, $s1, $0
```

Set \$s0 to 4. XOR immediate results in \$s1=5. Shift logic left results in that \$t0 is 10. Hence, \$t0 and \$s0 are not equal, so the branch is not taken and add is executed. This results in that \$s1 is 9.

There is no MOV instruction in MIPS, but **add** can be used for this (as it is done here).

What is the value of \$s5?
Stand for 9, sleep for 10.



Answer: 9

Note: There is a **pseudoinstruction** called **move** in the MIPS assembler. It is implemented using add.



Stored Programs with Instruction Encoding Formats

Stored program concept

Code is data. Code is stored in memory as any other data, enabling *general purpose computing*.

Word address

| | | | | |
|-----------|----|----|----|--------|
| . | . | . | . | . |
| . | . | . | . | . |
| . | . | . | . | . |
| 0040 000C | 0f | a0 | b0 | 12 |
| 0040 0008 | 44 | 93 | 4e | aa |
| 0040 0004 | 33 | fa | 01 | 23 |
| 0040 0000 | 21 | a0 | 1b | 33 |
| | | | | Word 3 |
| | | | | Word 2 |
| | | | | Word 1 |
| | | | | Word 0 |

For MIPS, there is 3 instruction formats:

- R-Type (register-type)
- I-Type (immediate-type)
- J-Type (jump-type)

In MIPS, each instruction requires exactly one word (32 bits) of space.

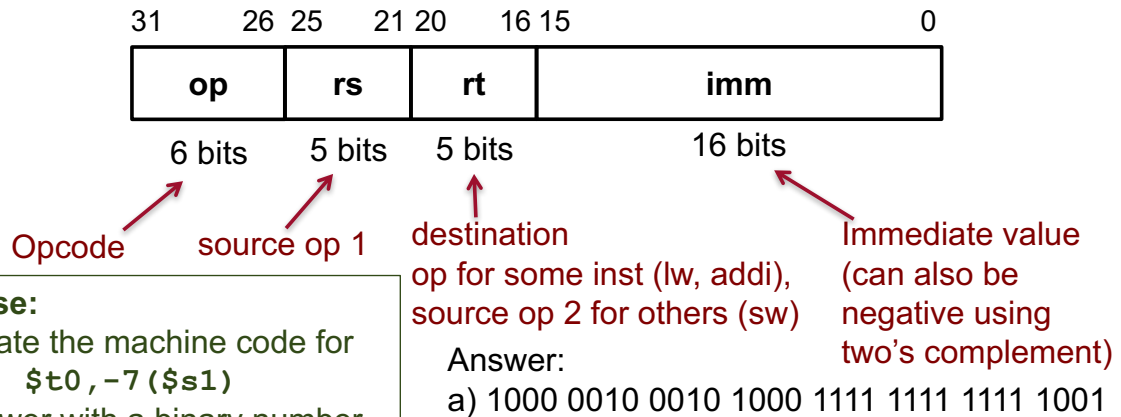
MIPS programs are typically stored from address 40 0000.

MIPS code must be word-aligned (start at addresses 0,4, 8, C etc.). X86 does not require word alignment.



I-Type Instructions

I-Type (immediate-type) instructions have two register operands and one immediate operand.



David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Summary Part I

Some key take away points:

- **Moore's law:** Integrated circuit resources (transistors) double every 18-24 months.
- **The Power Wall:** Clock rates cannot be increased anymore. Too high power; the chip gets too hot.
- An **Instruction Set Architecture (ISA)** defines the software/hardware interface, whereas a **microarchitecture** implements an ISA.
- It is important to understand **the concept of assembly programming**, although very few programs are actually written in assembly today.



David Broman
dbro@kth.se



Part I
Assembly and
Machine Code

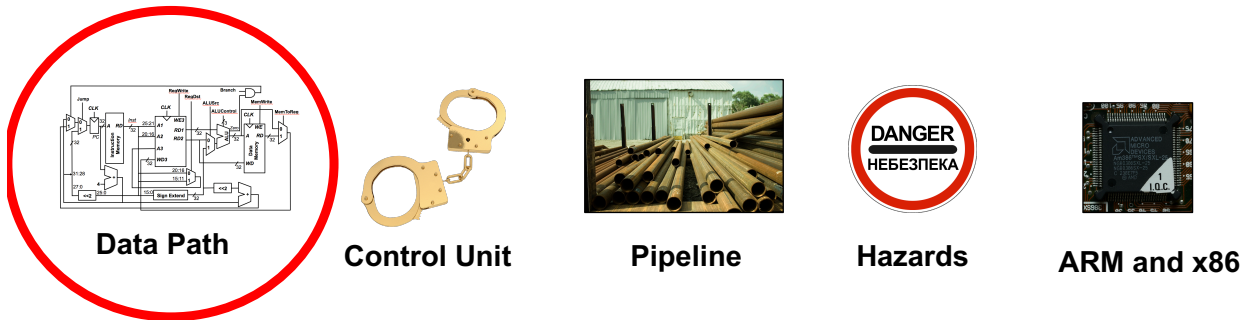
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part II

Processor Design



Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



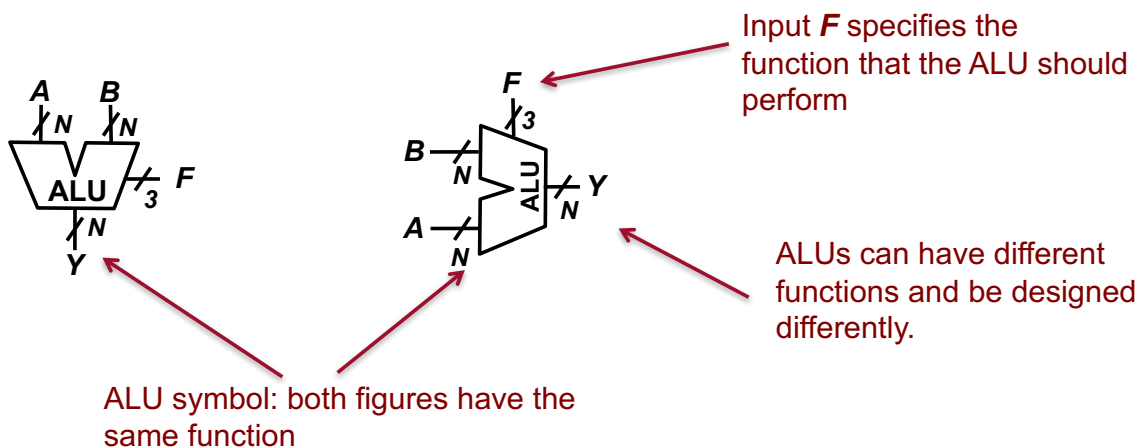
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Arithmetic Logic Unit (ALU)

An **ALU** saves hardware by combining different arithmetic and logic operations in one single unit/element.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



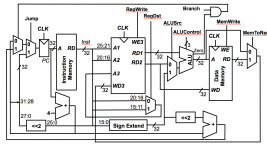
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Data Path and Control Unit

A processor is typically divided into two parts



Data Path

- Operates on a word of data.
- Consists of elements such as registers, memory, ALUs etc.



Control Unit

- Gets the current instruction from the data path and tells the data path how to execute the instruction.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Instructions

In this lecture, we construct a microarchitecture for a subset of a MIPS processor with the following instructions

R-Type: add, sub, and, or, slt

Arithmetic / logic instructions

Memory instructions

I-Type: addi, lw, sw, beq

Arithmetic
immediate
instruction

Branch instructions

J-Type: j



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

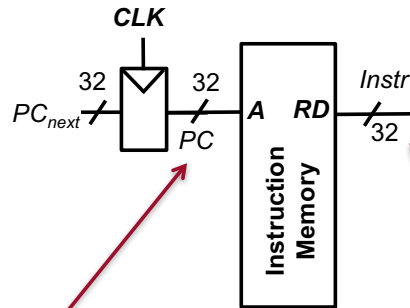


Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Read Instruction from the Current PC



First step. Read the instruction at the current PC address.

A 32-bit instruction *Inst* is **fetched**.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



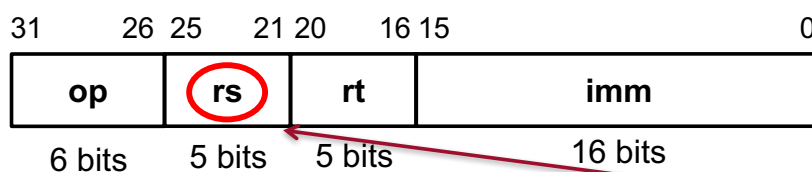
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs



lw instruction – Read Base Address



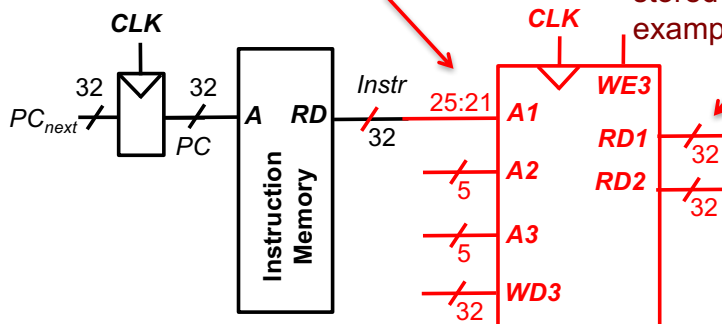
Example

`lw $s0, 4($s1)`

Read out the base address from the register file. 25:21 cuts out the 5 bits from the instruction.

Base address in **rs**

RD1 has now the address stored in \$s1 (in the above example).



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

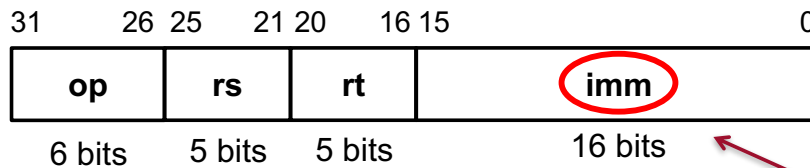


Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

1w instruction – Read Offset



Example
lw \$s0, 4(\$s1)

The offset is stored in the imm field.

The offset is signed. Sign extend to 32 bits.

That is:

$$Simm_{15:0} = Instr_{15:0}$$

$$Simm_{31:16} = Instr_{15}$$

The offset is found in the least significant 16 bits of the instruction.

15:0

Sign Extend

Simm

32

David Broman
dbro@kth.se

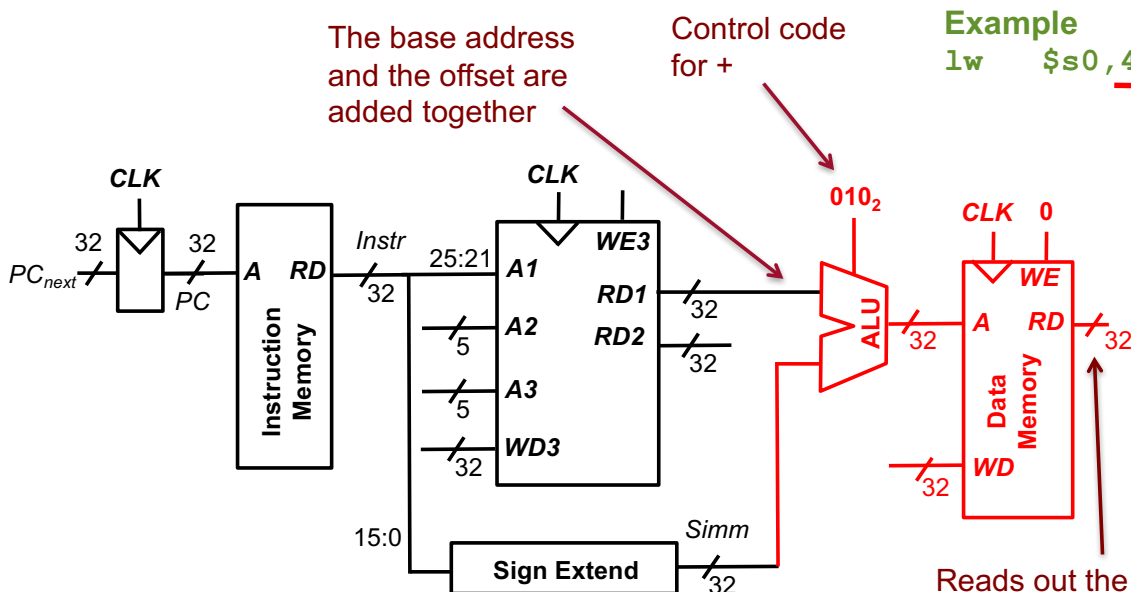
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

1w instruction – Read Data Word



Example
lw \$s0, 4(\$s1)

The base address and the offset are added together

Control code for +

010₂

Reads out the data word from data memory.

David Broman
dbro@kth.se

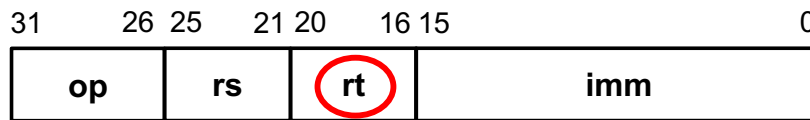
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

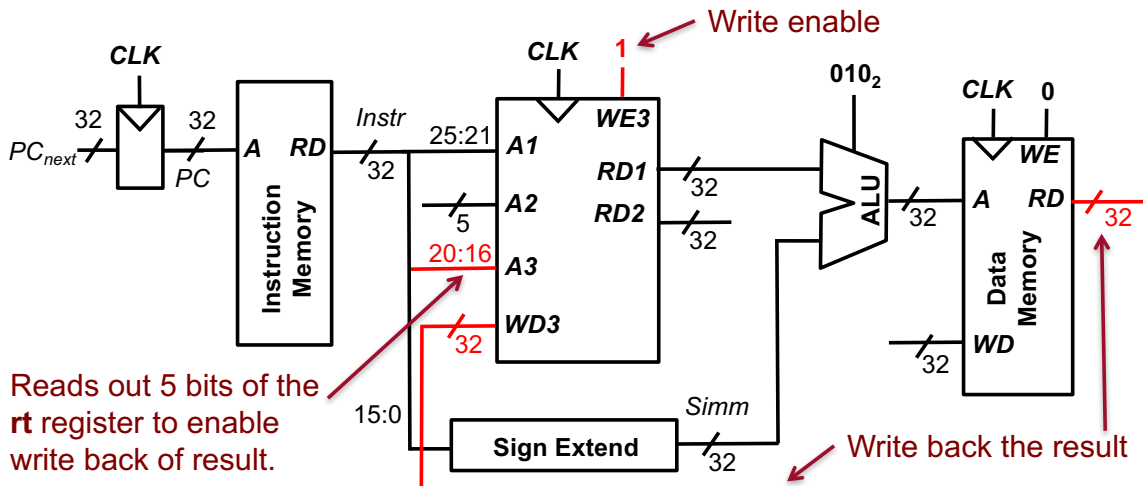
Part IV
Parallel Processors
and Programs

1w instruction – Write Back



Example

lw \$s0, 4(\$s1)



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

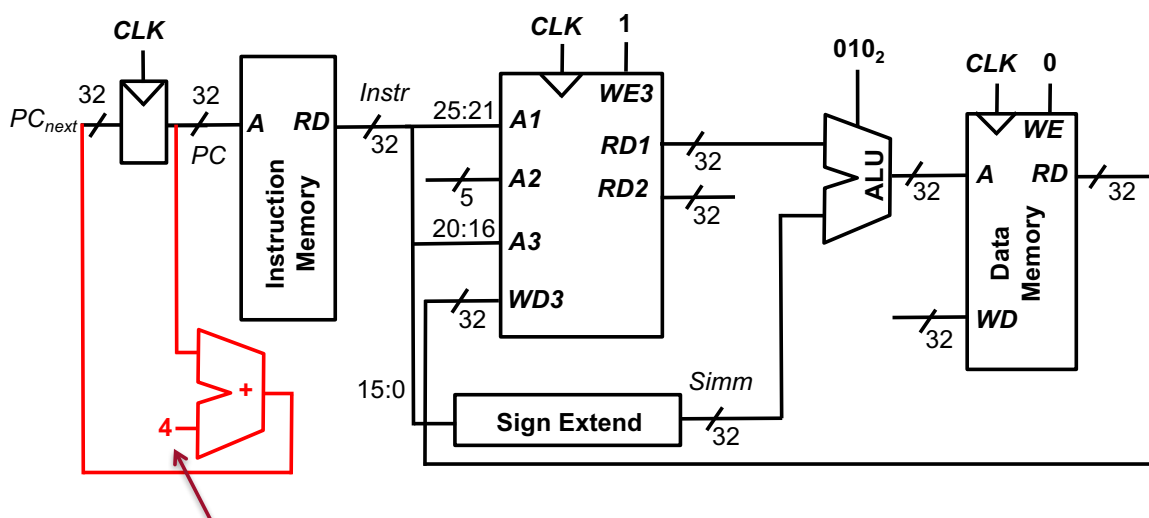
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

28

1w instruction – Increment PC



This is the complete data path for the load word (lw) instruction.

David Broman
dbro@kth.se

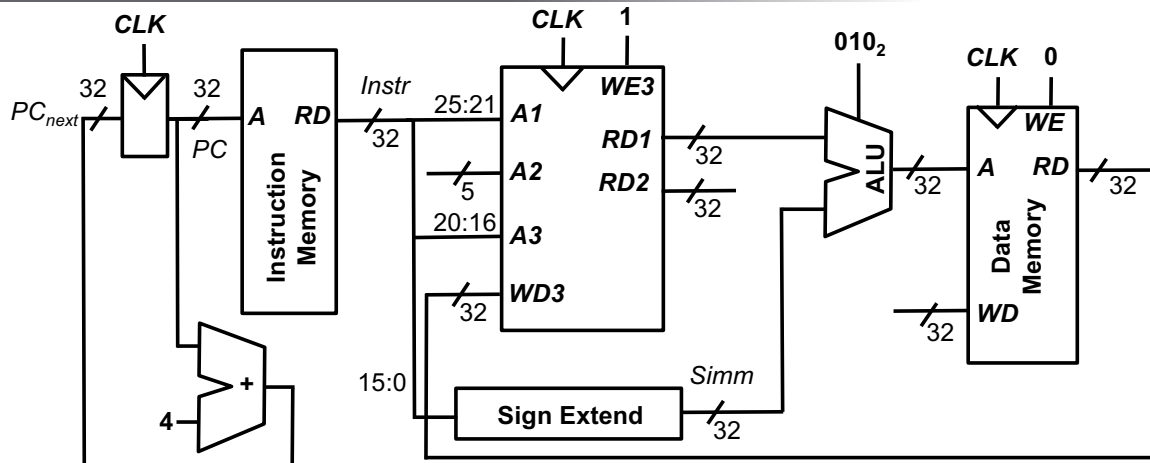
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

1w instruction – Timing



Combinational logic during clock cycle:
read instruction, sign extend, read from
register file, perform ALU operation, and
read from the data memory.

At the raising clock edge:
Write to the register file
and update the PC.

CLK

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

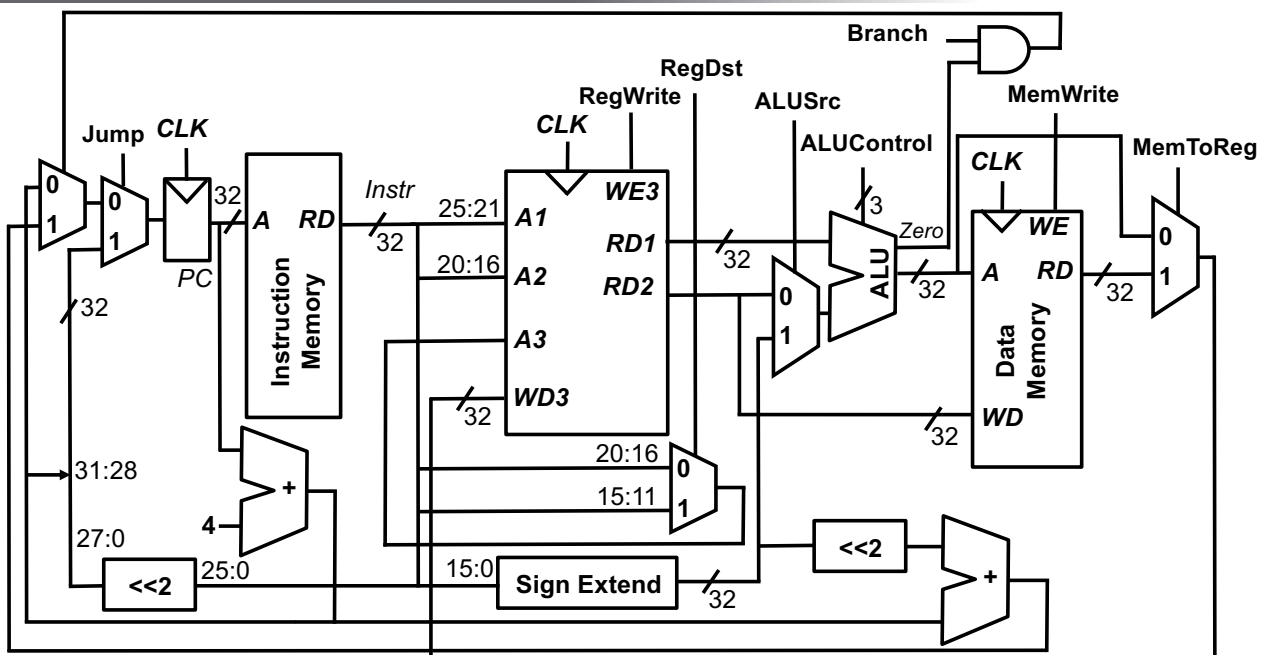
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Data Path for Instructions

add, sub, and, or, slt, addi, lw, sw, beq, j



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

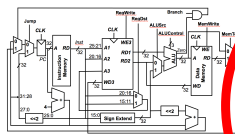
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part II

Processor Design



Data Path



Control Unit



Pipeline



Hazards



ARM and x86

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

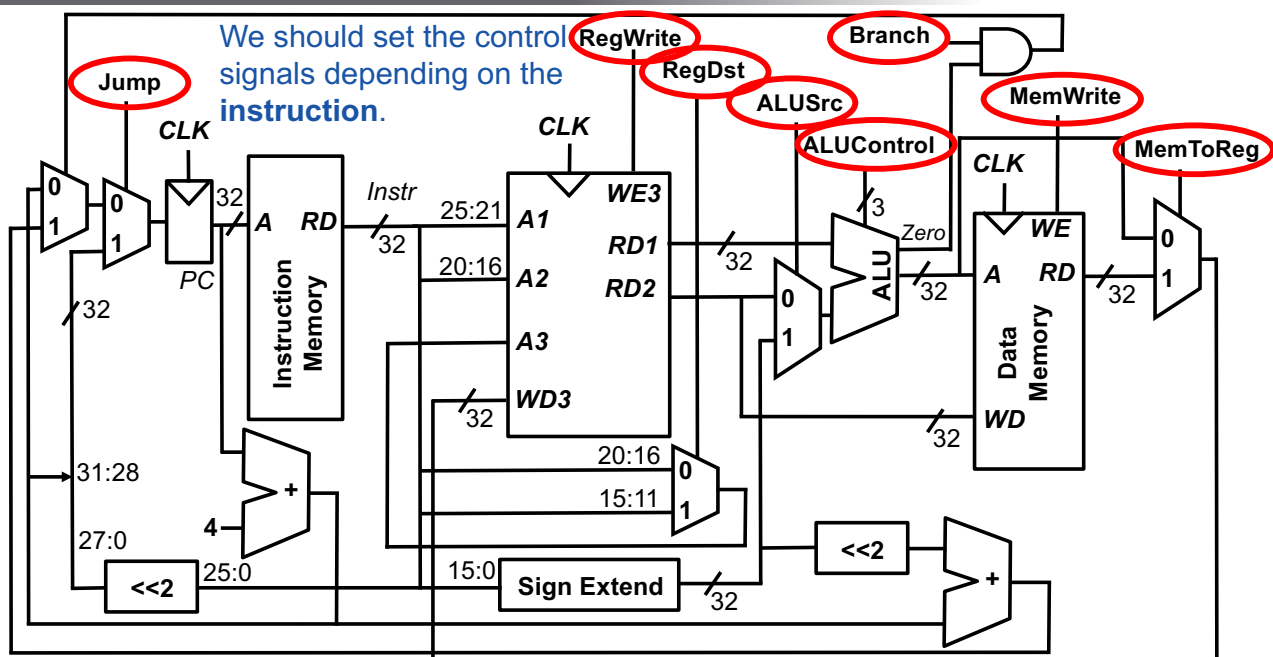


Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

What to Control?



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



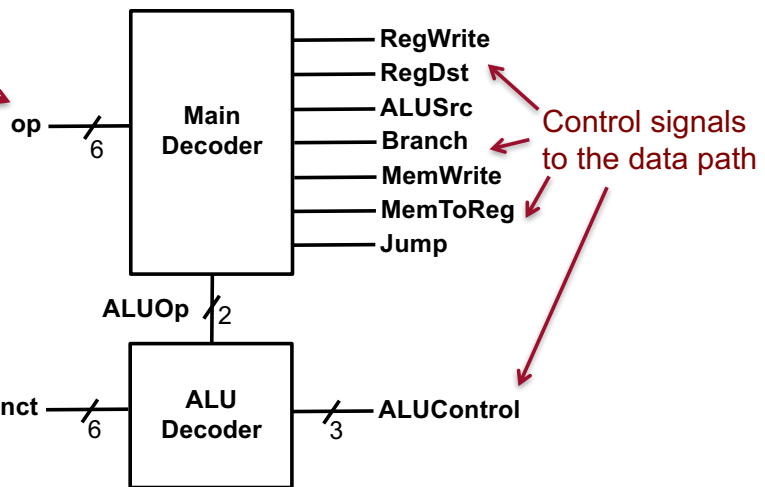
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

The 6 bits **op** field from all instruction types

The 6 bits **funct** field from the R-type. Ignored if other types.



How should we analyze the performance of a computer?

- By clock frequency?
- By instructions per program?

$$\text{Execution time (in seconds)} = \# \text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Number of instructions in a program (# = number of)

Determined by programmer or the compiler or both.

Average **cycles per instruction (CPI)**

Determined by the micro-architecture implementation.

Seconds per cycle = **clock period T_c** .

Determined by the critical path in the logic.

Problem:

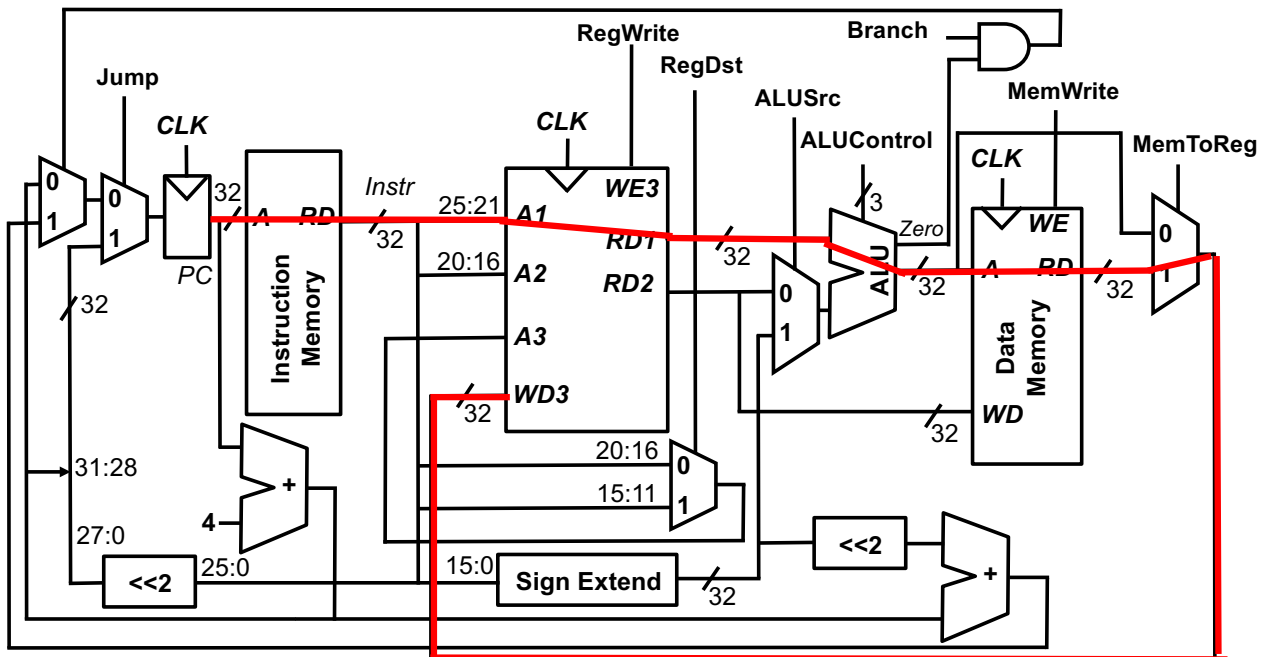
- Your program may have many inputs.
- Not only one specific program might be interesting.

Solution:

Use a **benchmark** (a set of programs).
Example: SPEC CPU Benchmark

Critical Path Example: Load Word (lw) Instruction

35



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Performance Analysis (Revisited)

36

$$\text{Execution time (in seconds)} = \# \text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Number of instructions in a
program (# = number of)

Determined by programmer
or the compiler or both.

Average **cycles per
instruction (CPI)**

Determined by the micro-
architecture implementation.

For the single-cycle
processor, each
instruction takes one
clock cycle. That is,
CPI = 1.

Seconds per cycle =
clock period T_c .

Determined by the
critical path in the logic.

The main problem with the
single-cycle processor
design (last lecture) is the
long critical path.

Solution: Pipelining

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

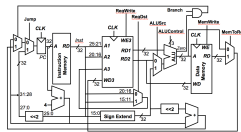
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part II

Processor Design



Data Path



Control Unit



Pipeline



Hazards



ARM and x86

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Parallelism and Pipelining (1/6)

Definitions

38

Processing System: A system that takes input and produces outputs.



Token: An input that is processed by the processing system and results in an output.

Latency: The time it takes for the system to process one token.

Throughput: The number of tokens that can be processed per time unit.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

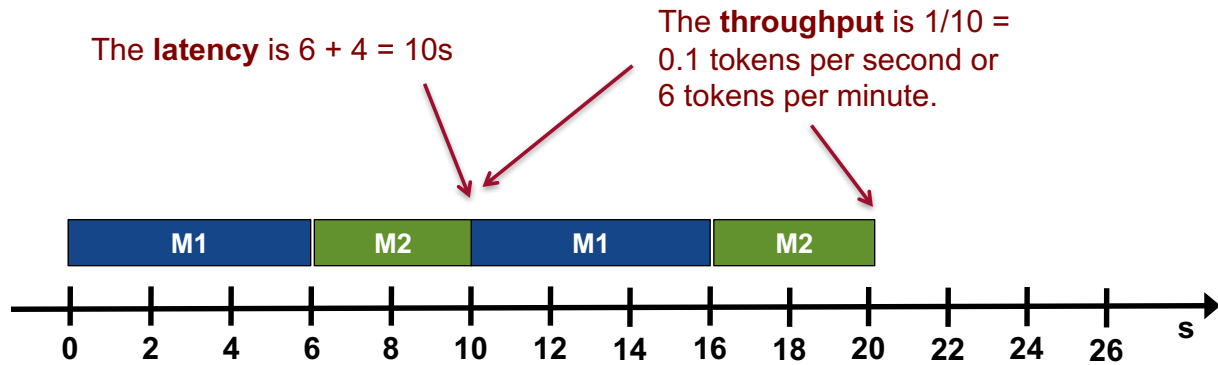
Example: Assume we have a Christmas card factory with two machines (M1 and M2).

M1: Prints out the card (takes 6s)

M2: Puts on a stamp (takes 4s)

Approach 1. Process tokens **sequentially**.

In this case a token is a card.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Example: Assume we have a Christmas card factory with four machines.

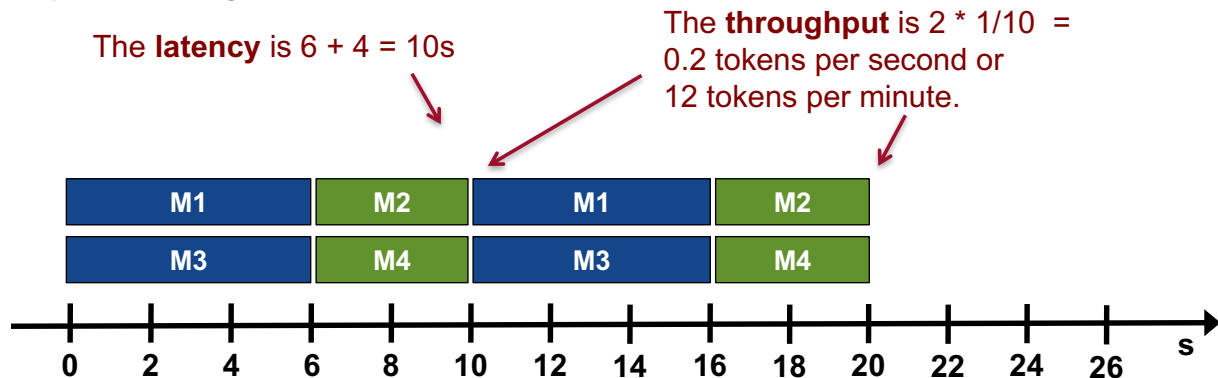
M1: Prints out the card (takes 6s)

M2: Puts on a stamp (takes 4s)

M3: Prints out the card (takes 6s)

M4: Puts on a stamp (takes 4s)

Approach 2. Process tokens **in parallel** using more machines.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Example: Assume we have a Christmas card factory with two machines.

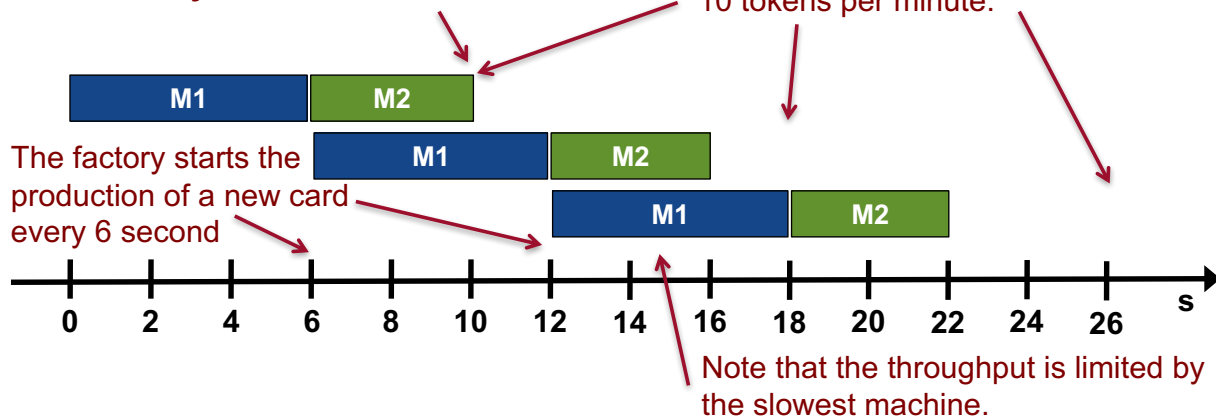
M1: Prints out the card (takes 6s)

M2: Puts on a stamp (takes 4s)

Approach 3. Process tokens by **pipelining** using only two machines.

The **latency** is still $6 + 4 = 10\text{s}$

The **throughput** is $1/6$ (on average) = $0.1666\dots$ tokens per second or 10 tokens per minute.



Idea: We introduce a pipeline in the processor

How does this affect the execution time?

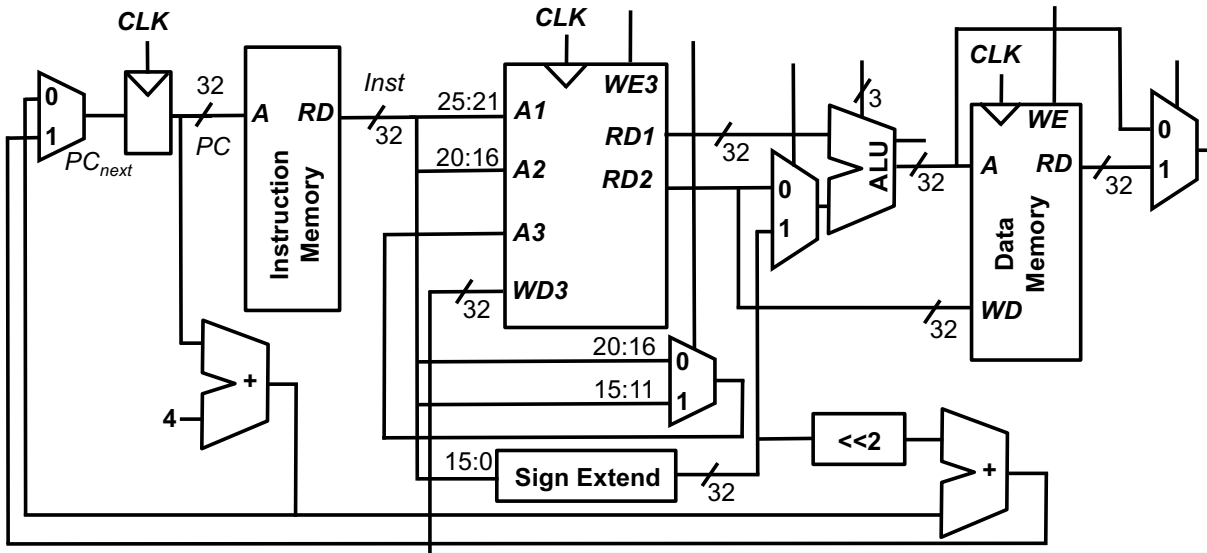
$$\text{Execution time (in seconds)} = \# \text{ instructions} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Pipelining does not change the number of instructions

Pipelining will not improve the CPI (actually, make it slightly worse)

Pipelining will improve the cycle period (make the critical path shorter)

Recall the single-cycle data path (the logic for the `j` and `beq` instructions is hidden)



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



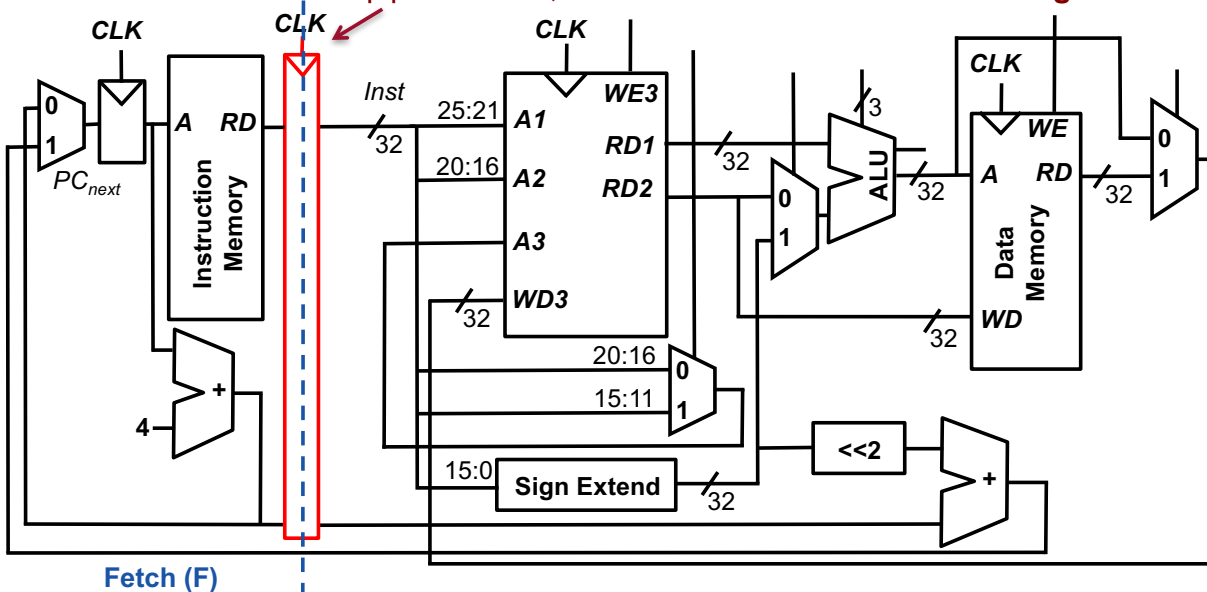
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Towards a Pipelined Datapath (2/8) Fetch Stage

A register splits the datapath into stages, forming a pipeline. First, we introduce a instruction **fetch stage**.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

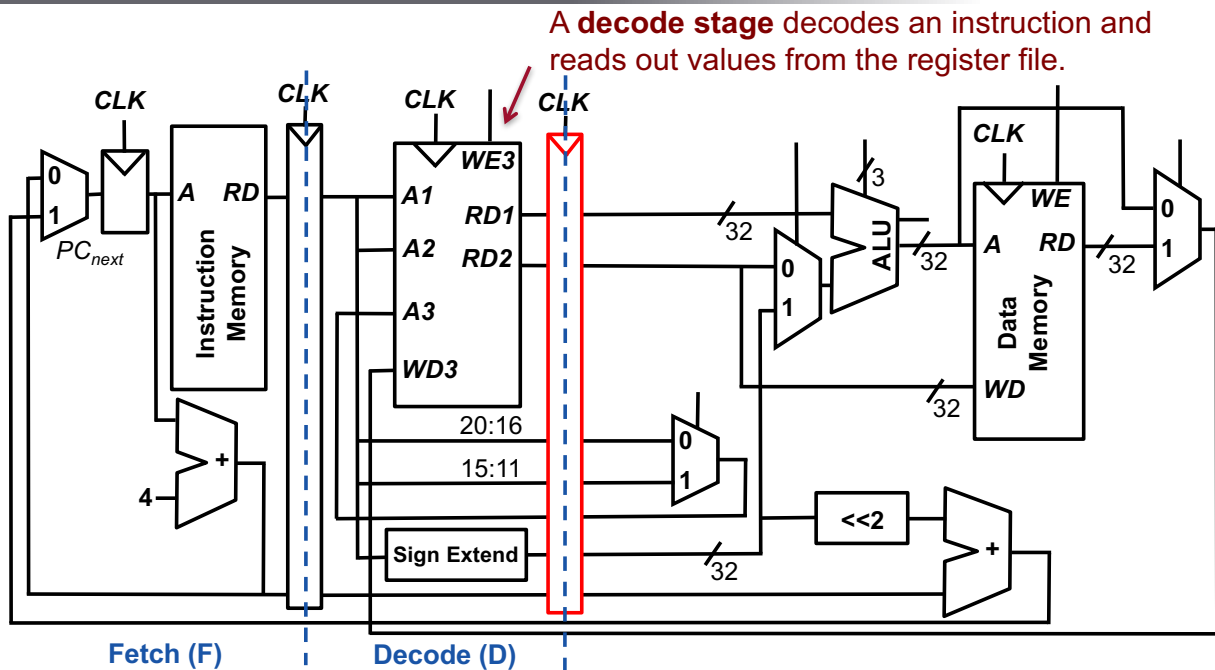


Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Towards a Pipelined Datapath (3/8) Decode Stage



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

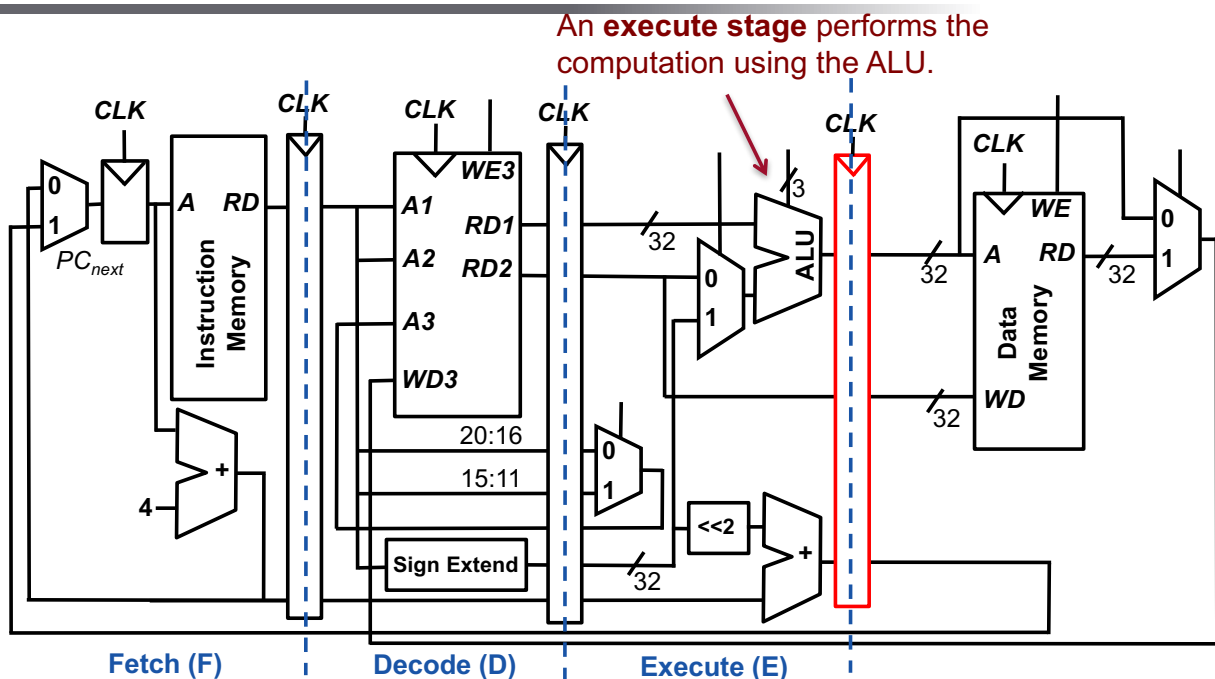


Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Towards a Pipelined Datapath (4/8) Execute Stage



David Broman
dbro@kth.se

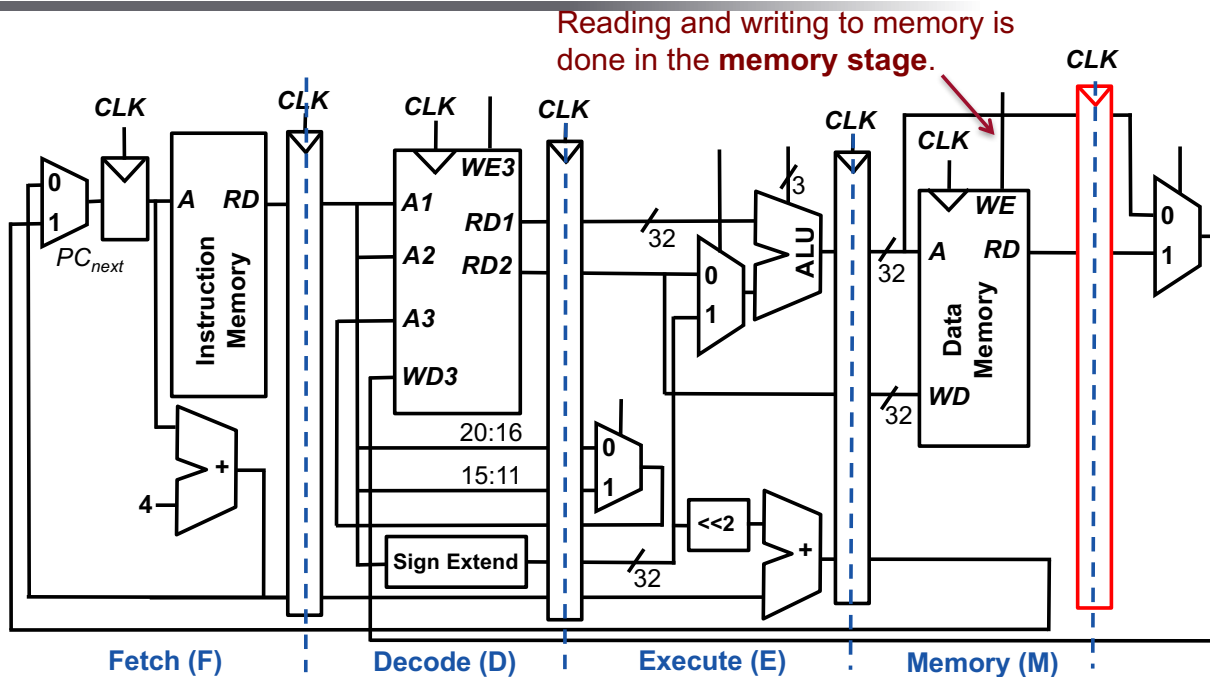
Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

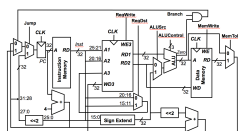
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part II

Processor Design



Data Path



Control Unit



Pipeline



Hazards



ARM and x86

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

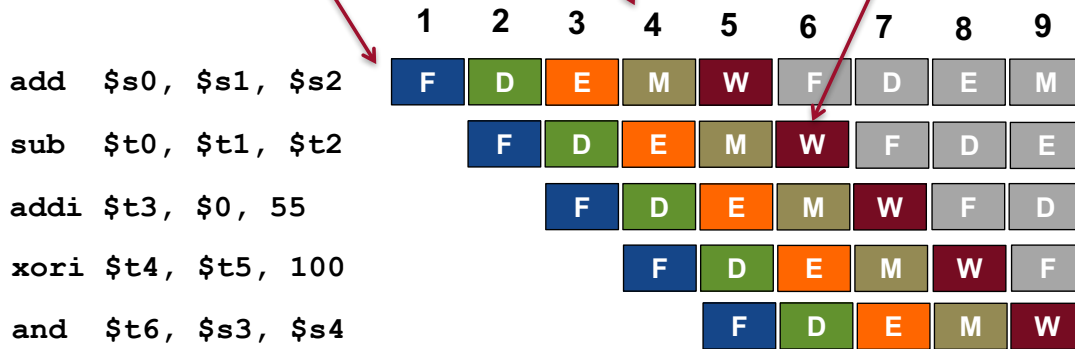
Part IV
Parallel Processors
and Programs

A Five-Stage Pipeline

In each cycle, a new instruction is fetched, but it takes 5 cycles to complete the instruction.

In each cycle all stages are handling different instructions in parallel.

Example. In cycle 6, the result of the **sub** instruction is written back to register **\$t0**.



We can fill the pipeline because there are no dependencies between instructions

Exercise: What is the ALU doing in cycle 5?

Answer: Adding together values 0 and 55

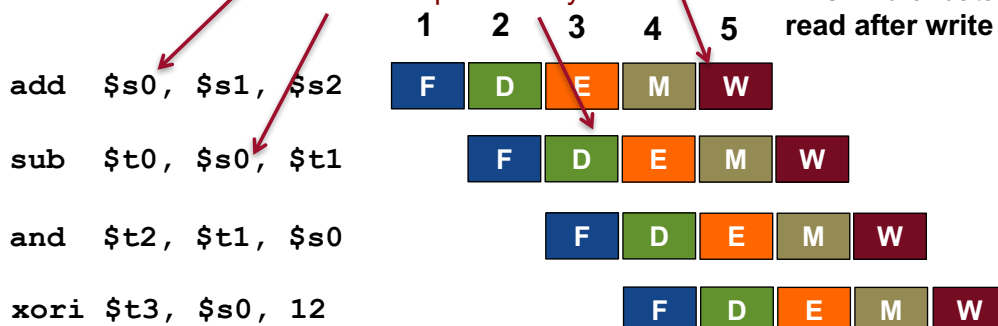
Data Hazards (1/4) Read after Write (RAW)

The **add** instruction writes back the value **\$s0** in cycle 5

But **\$s0** is used in the decode phase in cycle 3.

A **data hazard** occurs when an instruction reads a register that has not yet been written to.

This kind of data hazard is called **read after write (RAW)** hazard.



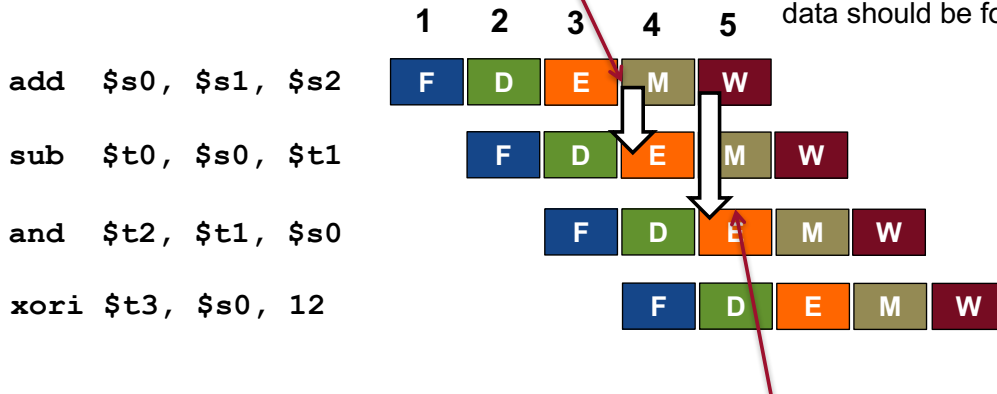
Data Hazards (2/4)

Solution 1: Forwarding

51

The result from the execute stage for **add** can be **forwarded** (also called **bypassing**) to the execute stage for **sub**.

Hazard detection is implemented using a **hazard detection unit** that gives control signals to the datapath if data should be forwarded.



Can all data hazards be solved using forwarding?

The **and** instruction's hazard is solved by forwarding as well.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Data Hazards (3/4)

Solution 1: Forwarding (partially)

52

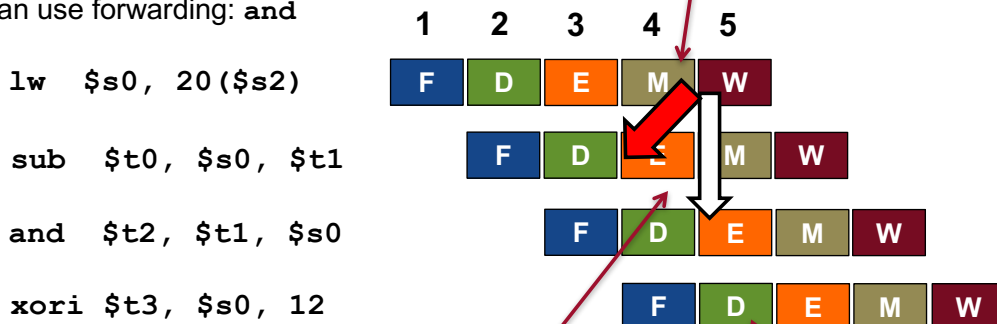


Exercise: Which of the instructions **sub**, **and**, and **xori** have data hazards? Which can be solved using forwarding?

Answer:

Hazards: **sub** and **and**

Can use forwarding: **and**



The **sub** instruction cannot be solved using forwarding because the memory access is available at the end of cycle 4, but is needed in the beginning of cycle 4.

The **and** instruction memory result can be forwarded after the memory stage to execution.

xori can read the data from the write stage (writes in first part of cycle, reads in second part)

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



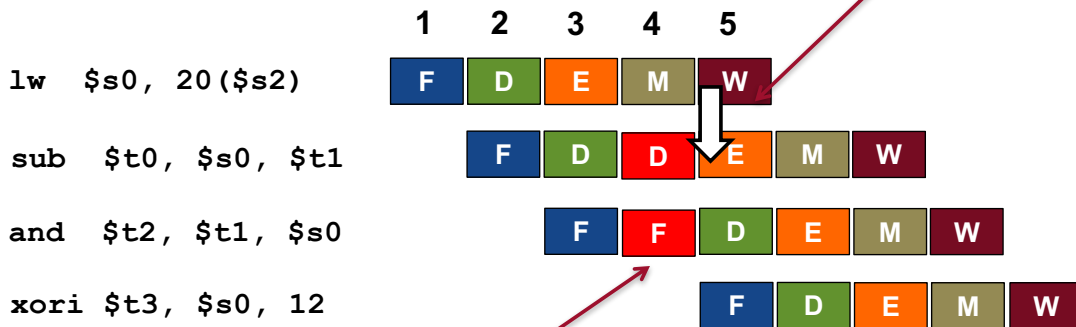
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

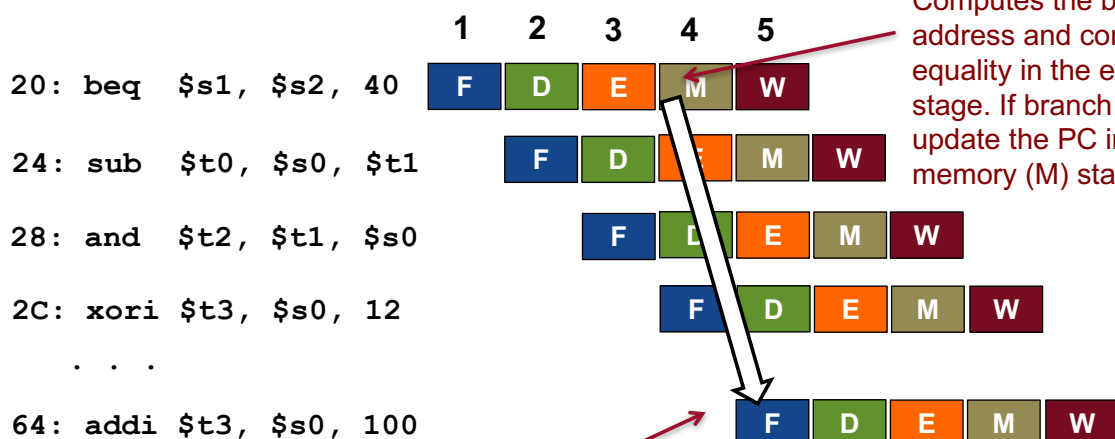
Solution when forwarding does not work: **stalling**

After stalling, the result can be forwarded to the execute stage.



We need to stall the pipeline. Stages are repeated and the fetch of **xori** is delayed.

Stalling results in more than one cycle per instruction. The unused stage is called a **bubble**.



Computes the branch target address and compares for equality in the execute (E) stage. If branch taken, update the PC in the memory (M) stage.

If the branch is taken, we need to flush the pipeline. We have a **branch misprediction penalty** of 3 cycles.

Why do we sometimes want more stages than 5?

The critical path can be shorter with less logic in the slowest stage.

The processor can have higher clock frequency.

For instance, Intel's Core 2 duo has more than 10 pipeline stages.

Why not always have more pipeline stages?

Adds hardware (registers)

The branch misprediction penalty increases!

How can we handle deep pipelines, and minimize misprediction?



Static Branch Predictors

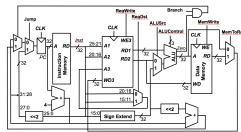
- Statically (at compile time) determine if a branch is taken or not. For instance, predict branch not taken.

Dynamic Branch Predictors

- Dynamically (at runtime) predict if a branch will be taken or not.
- Operates in the fetch state.
- Maintains a table, called the **branch target buffer**, that contains hundreds or thousands of executed branch instructions, their destinations, and information if the branches were taken or not.

Part II

Processor Design



Data Path



Control Unit



Pipeline



Hazards



ARM and x86

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



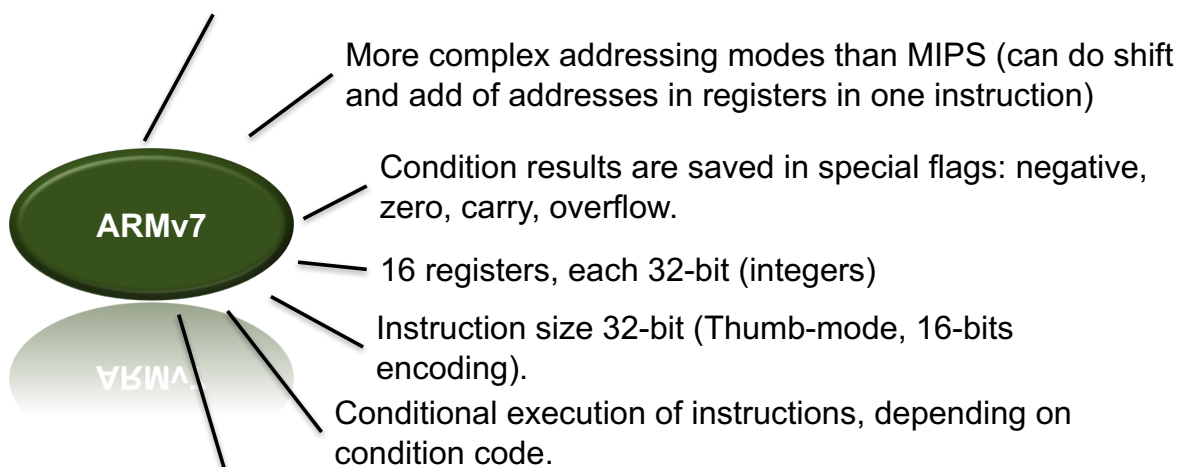
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Armv7

The most popular ISA for **embedded devices**



Example1: ARM Cortex-A8, a processor at 1GHz, 14-stage pipeline, with branch predictor.

Example 2: A6 by Apple, manufactured by Samsung, used in iPhone 5.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

x86

Standard in laptops, PCs, and in the cloud

CISC –instructions are more powerful than for ARM and MIPS, but requires more complex hardware

x86 architecture has evolved over the last 35 years,

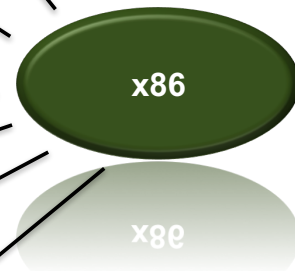
There are 16, 32, and 64 bits variants.

8 general purpose registers (eax, ebx, ecx, edx, esp, ebp, esi, edi).

Variable length of instruction encoding (between 1 and 15 bytes)

Arithmetic operations allow destination operand to be in memory.

Major manufacturers are Intel and AMD.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Summary Part II

Some key take away points:

- The **data path** consists of sequential logic that performs processing of words in the processor.
- The **control unit** decodes instructions and tells the data path what to do.
- Pipelining is a **temporal** way of achieving parallelism
- Pipelining introduces pipeline hazards. There are two main kind of hazards: **data hazards** and **control hazards**.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code



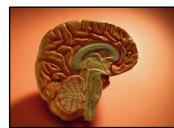
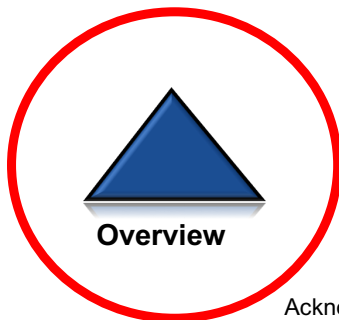
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part III

Memory Hierarchies



Instruction and Data Caches.



Virtual memory

Acknowledgement: The structure and several of the good examples are derived from the book "Digital Design and Computer Architecture" (2013) by D. M. Harris and S. L. Harris.

David Broman
dbro@kth.se

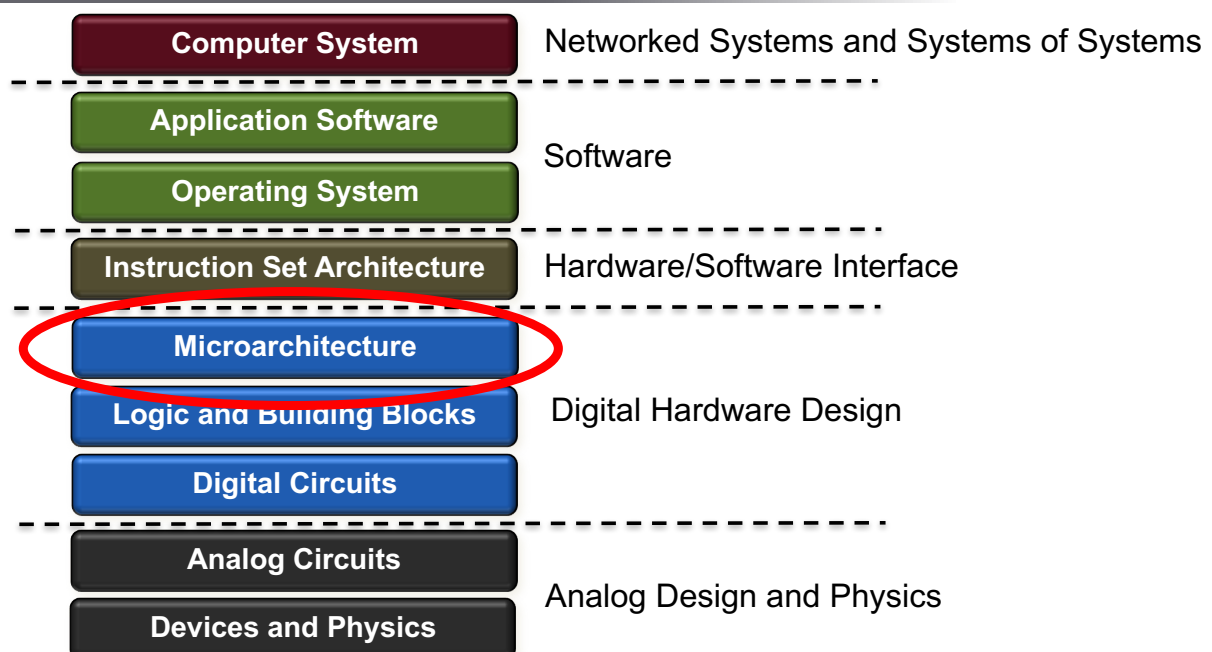
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Abstractions in Computer Systems



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

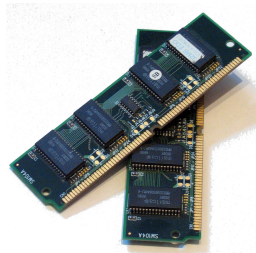
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs



SRAM (Static Random Access Memory)

- Simple integrated circuit, usually with one access port.
- Today, on-chip memory (same as processor)
- Access time 0.5-2.5ns Cost per GiB in 2012: \$500-\$1000



Douglas Whitaker, Wikipedia, CC BY-SA 2.5

DRAM (Dynamic Random Access Memory)

- Memory stored in capacitors – need to be refreshed
- One transistor per bit – much cheaper than SRAM
- SDRAM (synchronous DRAM). Uses clocks. Transfer data in bursts.
- DDR (Double Data Rate) SDRAM. Transfer data both on rising and falling clock edge.
- Access time: 50-70ns, Cost per GiB in 2012: \$10-\$20

Source: Patterson and Hennessy, 2012

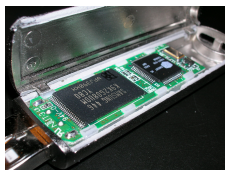
David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs



Wikipedia, CC BY-SA 3.0

Flash Memory

- Electrically erasable programmable read-only memory (EEPROM)
- Can wear out
- Used in solid state drives (SSD)
- Access time: 5,000-50,000 ns, Cost per GiB in 2012 \$0.75-\$1



Wikipedia Evan Amos, CC BY-SA 3.0

Magnetic Disk

- Collection of platters that spin 5,400 to 15,000 revolutions per minutes (rpm).
- Access time: 5,000,000-50,000,000 ns
Cost per GiB in 2012 \$0.05-\$0.10

Clearly, there is a tradeoff between cost, access time, and size

How can we utilize these differences? Source: Patterson and Hennessy, 2012

David Broman
dbro@kth.se

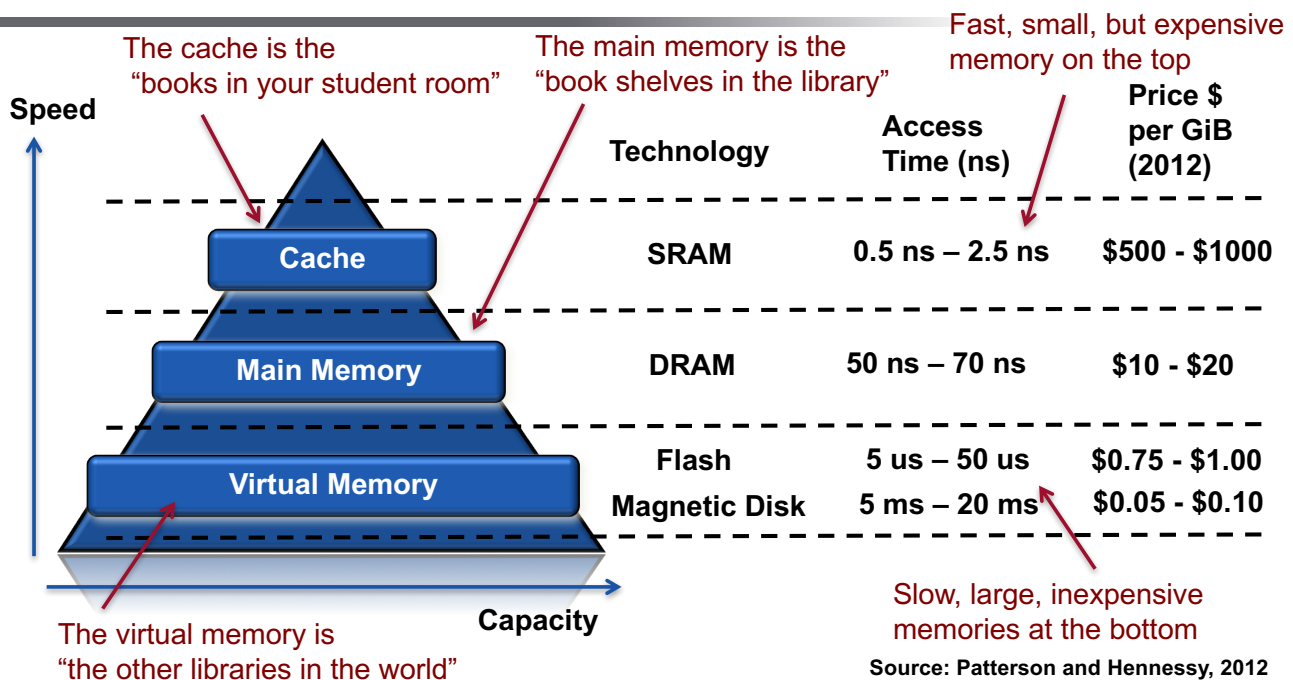
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Memory Hierarchy



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

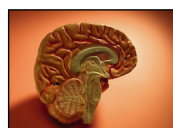
Part IV
Parallel Processors
and Programs

Part III

Memory Hierarchies



Overview



Instruction and
Data Caches.



Virtual memory

David Broman
dbro@kth.se

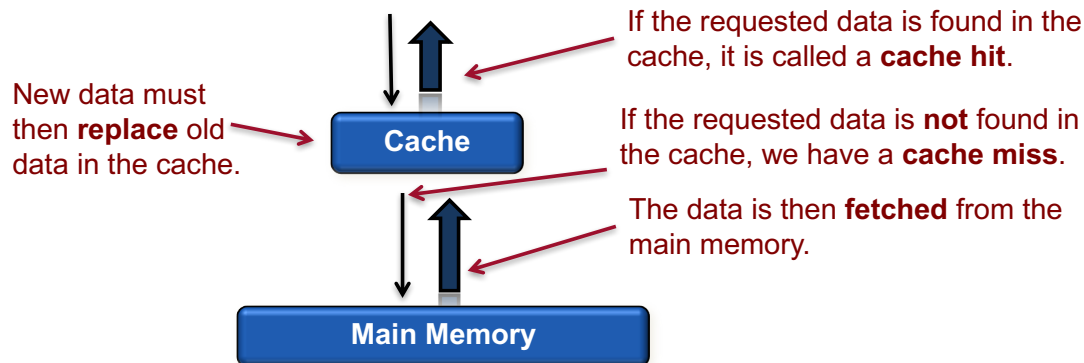
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Reading From Memory



$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Total number of memory accesses}}$$

$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Total number of memory accesses}}$$

What data should be in the cache so that we maximize the hit rate?

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

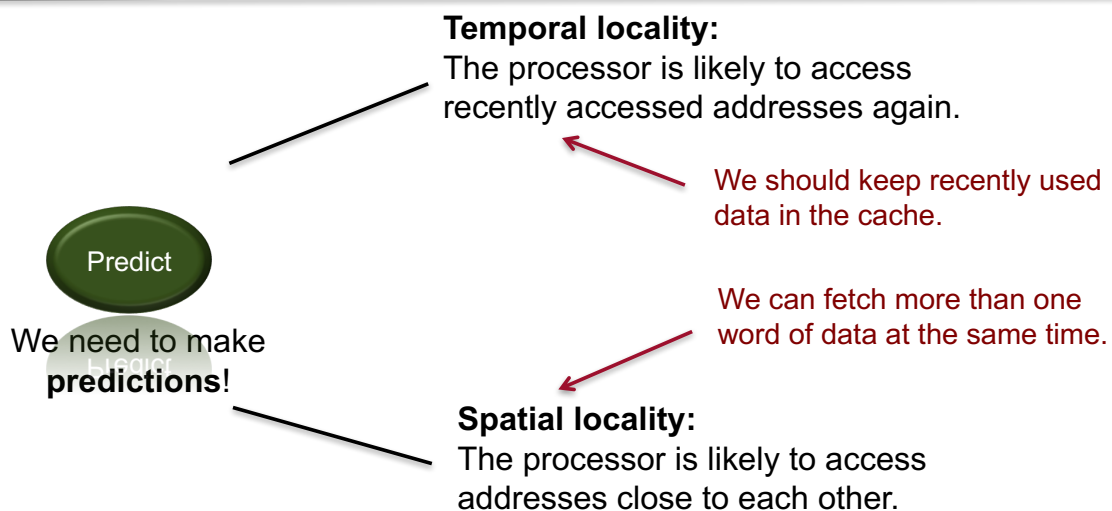
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

How to achieve low miss rates?

“It’s difficult to make predictions, especially about the future”

Attributed to various persons in the history



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Cache Terminology

Capacity (C)

Number of words or bytes in the cache.

Number of Sets (S)

Each set holds one or more blocks of data. (Sometimes the term **row** is used instead of set)

Block size (b)

Number of words or bytes in a block.

Number of Blocks (B)

The total number of blocks.
Always: $B \geq S$

Degree of associativity (N)

$N = B/S$

Minimal Cache Example

$N = 1$

Here $B=8$,
i.e., $B=S$.

The block size is
one word, $b=1$.

The **capacity** of cache
is $2^3 = 8$ words. The
word size is 32-bit.

| | |
|-------------|-------------------------|
| 0x0000 0000 | Set 7, 111 ₂ |
| 0x0000 0000 | Set 6, 110 ₂ |
| 0x0000 0000 | Set 5, 101 ₂ |
| 0x0000 0000 | Set 4, 100 ₂ |
| 0x0000 0000 | Set 3, 011 ₂ |
| 0x0000 0000 | Set 2, 010 ₂ |
| 0x0000 0000 | Set 1, 001 ₂ |
| 0x0000 0000 | Set 0, 000 ₂ |

There are 8
sets, i.e., $S=8$.

David Broman
dbro@kth.se

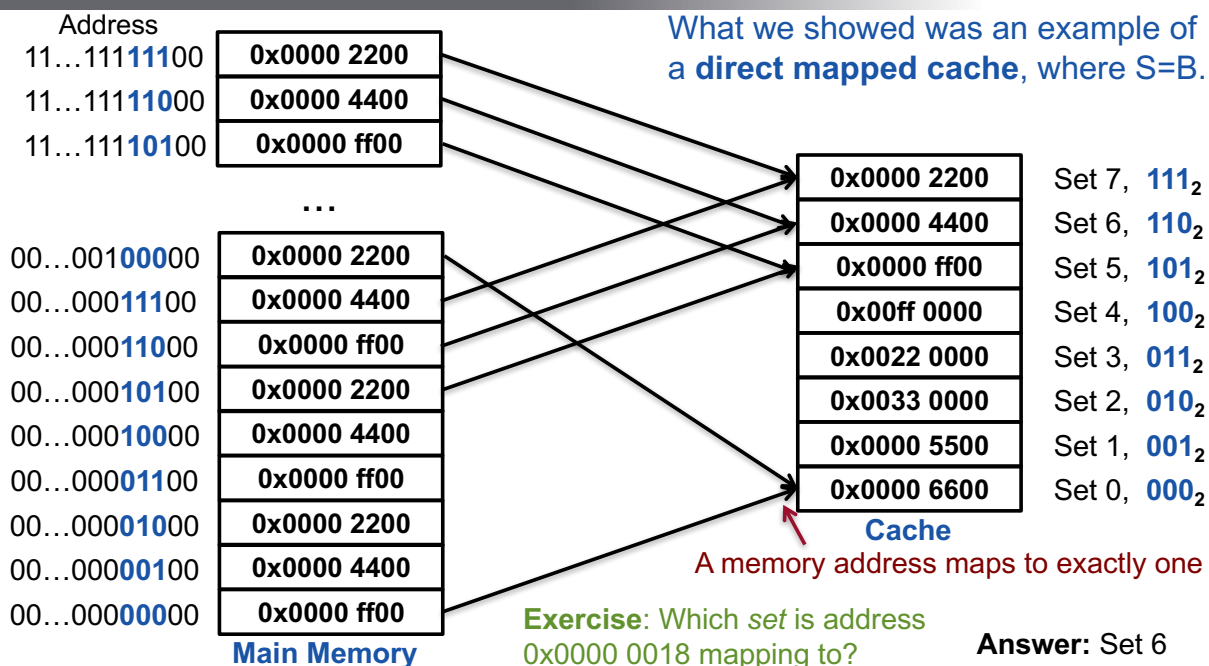
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Direct Mapped Cache (1/3) The mapping



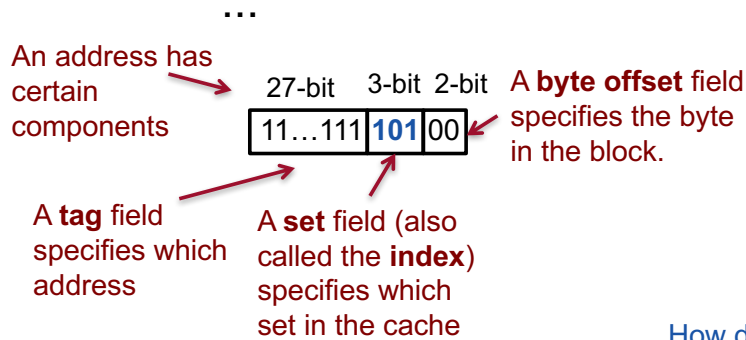
David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

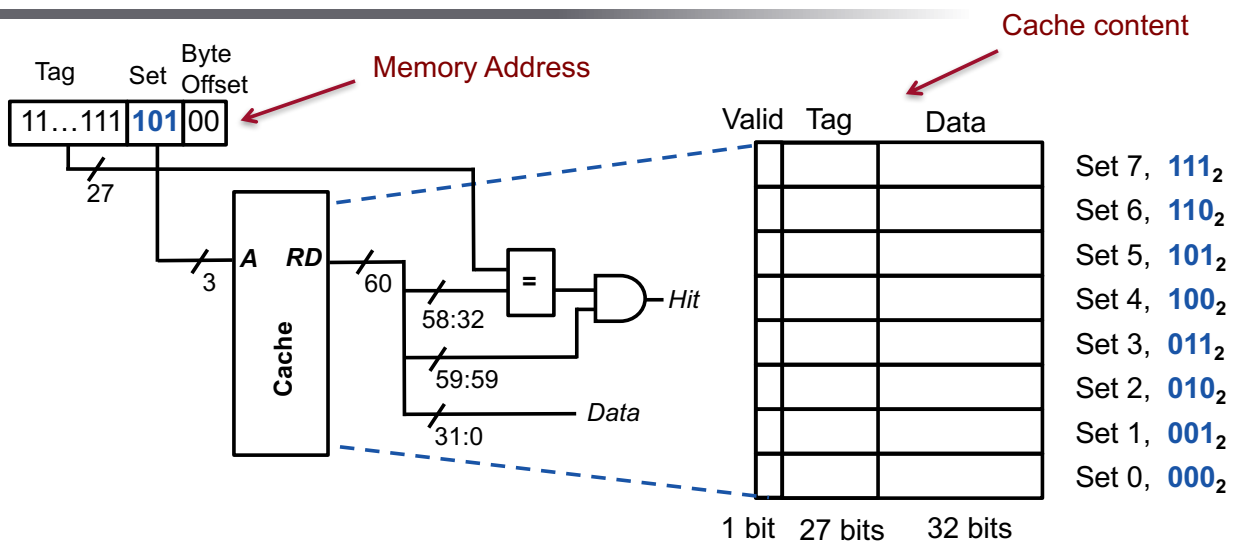


How do we know which memory blocks that are stored in the cache?

Answer: We store the tag field in the cache.

How do we know if the data stored in the cache is valid?

Answer: We add a valid bit in the cache.



Loop Example 1

Data Cache – Temporal Locality

73



```

addi $t0, $0, 5
loop:
    beq $t0, $0, done
    lw  $t1, 0x4($0)
    lw  $t2, 0xC($0)
    addi $t0, $t0, -1
    j   loop
done:
    
```

| Valid | Tag | Data | |
|-------|--------|---------------|--------------------------------|
| 0 | | | Set 7, 111 ₂ |
| 0 | | | Set 6, 110 ₂ |
| 0 | | | Set 5, 101 ₂ |
| 0 | | | Set 4, 100 ₂ |
| 1 | 00..00 | mem[0x00..0C] | Set 3, 011 ₂ |
| 0 | | | Set 2, 010 ₂ |
| 1 | 00..00 | mem[0x0004] | Set 1, 001 ₂ |
| 0 | | | Set 0, 000 ₂ |

1 bit 27 bits 32 bits

Exercise: Assume that the cache is empty when entering the program. What is the *data* cache miss rate and the cache contents when reaching program point *done*.

Answer: The missrate is $2/10 = 20\%$.

$$4_{16} = 00100_2 \quad C_{16} = 12_{10} = 01100_2$$

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Loop Example

Instruction Cache – Spatial Locality

74



```

addi $t0, $0, 5
loop:
    beq $t0, $0, done
    lw  $t1, 0x4($0)
    lw  $t2, 0xC($0)
    addi $t0, $t0, -1
    j   loop
done:
    
```

Note the **spatial locality**. Since we load 4 instructions each time, we do not get cache misses for each instruction.

Answer: Two cache misses

First, when loading the first 4 instructions, then when loading the two last instructions.

4 bits for representing 16 bytes block
 0x0000 4000 // Address to first addi
 0x0000 4010 // Address to second addi
 The mapping does not conflict.

Exercise: Assume that the first *addi* instruction starts at address 0x0000 4000. The instruction cache has $S = B, C = 4096$ bytes and $b = 16$ bytes. How many instruction cache misses occurs when executing the program, presupposed that the cache was empty from the beginning.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

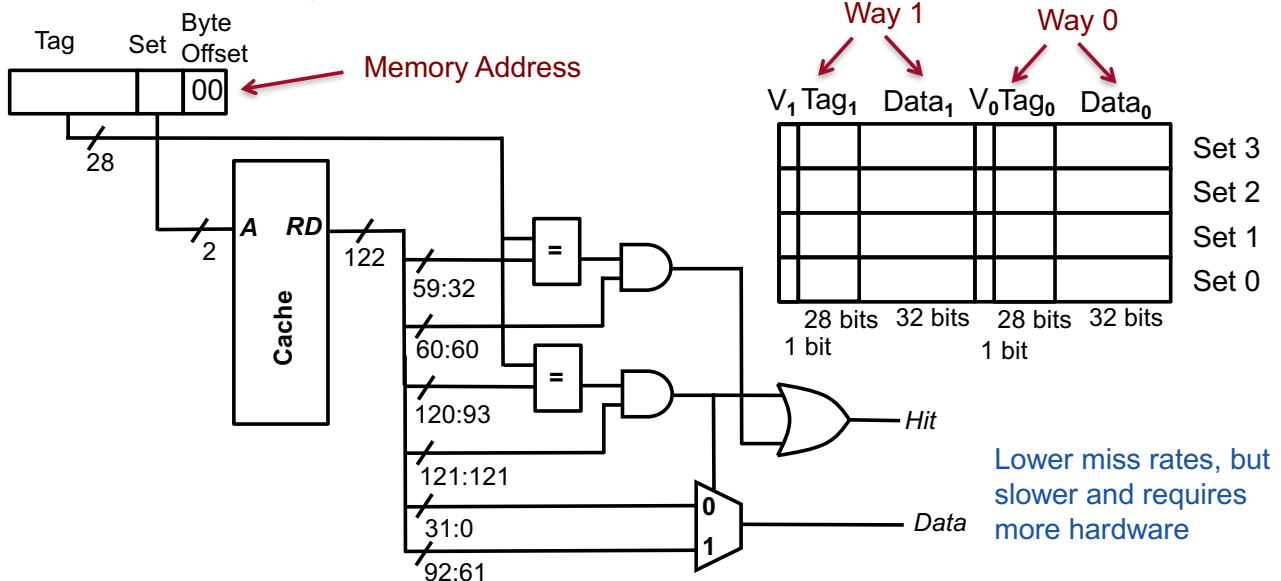
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

N-way Set Associative Cache

We can reduce conflicts by having N blocks in each set.

This is called an **N-way Set Associative Cache**.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Replacement Policy

Direct Mapped Cache

Each address maps to a unique block and set.

Hence, when a set is full, it must be replaced with the new data.

N-way Set Associative Cache where $N > 1$

- **Least Recently Used (LRU) Policy**. Simple with a 2-way set associative cache by using a *use bit U*. Commonly used.
- **Pseudo-LRU Policy**. For N-ways where $N > 2$. Indicate the least recently used *group* and upon replacement, randomly selects a way in the group.
- **First-in First-out (FIFO)** replacement policy.
- **Random** replacement policy.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

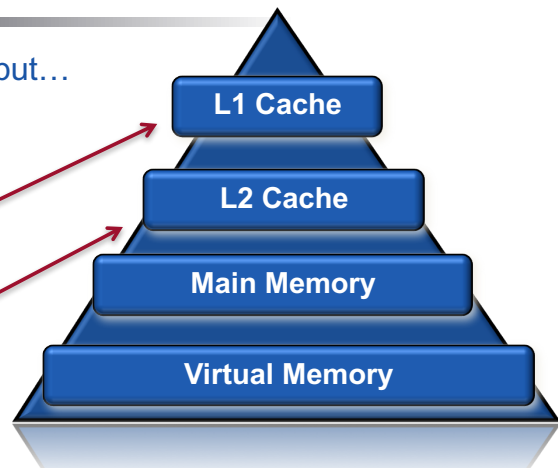
Multi-Level Caches

Large caches tend give lower miss rates, but...
Large caches tend to be slower.

Solution: Multi-Level Caches

L1 cache, small enough to get
1-2 cycle times.

The L2 cache is larger
and therefore slower.



Examples from Reality

ARM Cortex-A8

- L1, 4-way, 32KiB, split instruction/data, random replacement
- L2, 8-way, 128KiB, unified inst./data, random replacement
- No L3 cache

Intel Core-i7 920

- L1, 4-way (i), 8-way (d), 32KiB, split instruction/data, Approximate LRU
- L2, 8-way, 256KiB, unified inst./data
- L3, 16-way, 8MiB, Approximate LRU

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

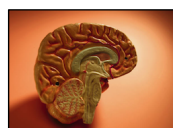
Part IV
Parallel Processors
and Programs

Part III

Memory Hierarchies



Overview



Instruction and
Data Caches.



Virtual memory

David Broman
dbro@kth.se

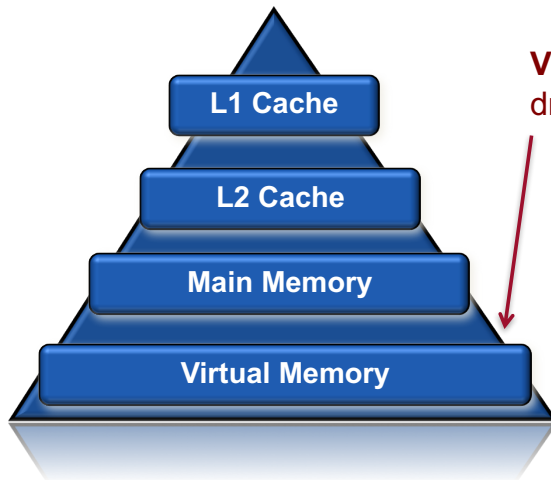
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Virtual Memory



Virtual memory uses the hard drive. Slow, but large memory.

Why?

- Gives the illusion of a **very big memory**.
- Gives **memory protection** between concurrent running programs (each process has its own virtual memory space).

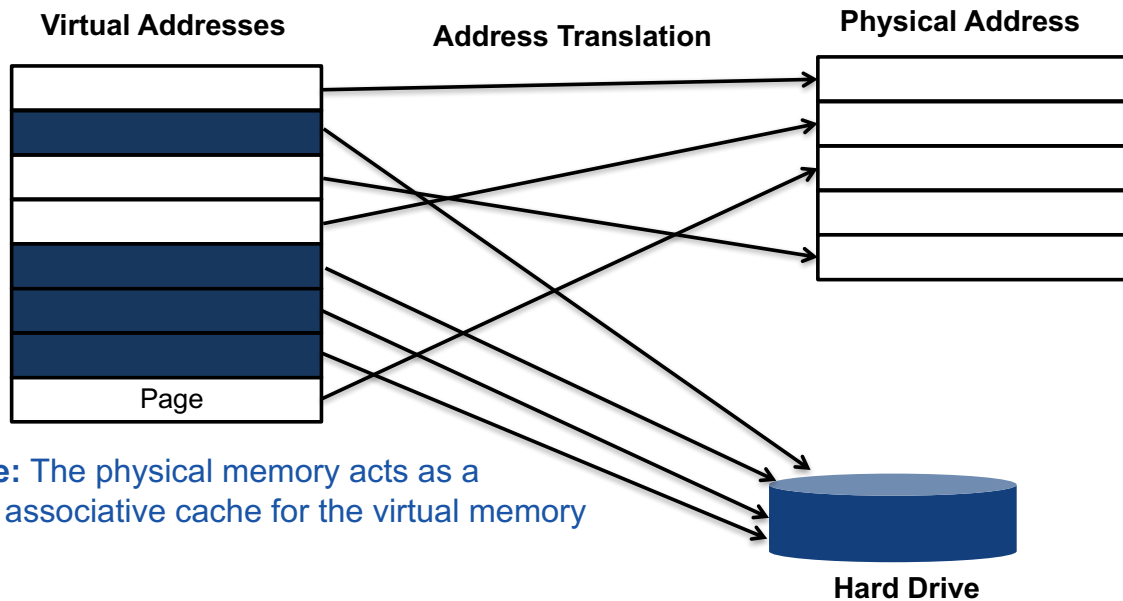
Virtual Memory vs. Caches

Virtual memory has similarities to caches, but uses another terminology.

| Cache | Virtual Memory |
|--------------|---------------------|
| Block | Page |
| Block size | Page Size |
| Block offset | Page offset |
| Miss | Page fault |
| Tag | Virtual Page number |

Typically, 4KB or more

Virtual Memory Overview



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Summary Part III

Some key take away points:

- **Memory hierarchies** are used because memories have different cost, size, and speed.
- There are two kinds of caches: **instruction caches** and **data caches**.
- Two important properties that make caches useful: **temporal locality** and **spatial locality**.
- Caches can be **direct mapped**, **N-way**, or **fully associative**.
- **Virtual memories** enable large virtual address spaces and enable memory protection between different concurrent programs.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

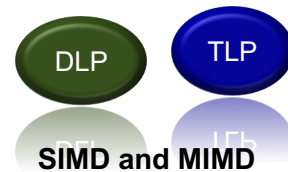
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part IV

Parallel Processors and Programs



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

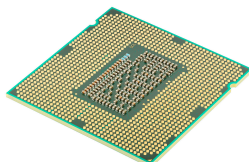
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

What is a multiprocessor?

A **multiprocessor** is a computer system with two or more processors.

By contrast, a computer with one processor is called a **uniprocessor**.



Multicore microprocessors are multiprocessors where all processors (cores) are located on a single integrated circuit.



A **cluster** is a set of computers that are connected over a local area network (LAN). May be viewed as one large multiprocessor.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Parallelism and Concurrency – what is the difference?

85

Concurrency is about **handling** many things at the same time.
Concurrency may be viewed from the **software** viewpoint.

Parallelism is about **doing (executing)** many things at the same time. Parallelism may be viewed from the **hardware** viewpoint.


| | | Software | |
|----------|----------|--|---|
| Hardware | Serial | Sequential Example: matrix multiplication on a uniprocessor. | Concurrent Example: A Linux OS running on a uniprocessor. |
| | Parallel | Example: matrix multiplication on a multicore processor. | Example: A Linux OS running on a multicore processor. |

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

 Part IV
Parallel Processors
and Programs

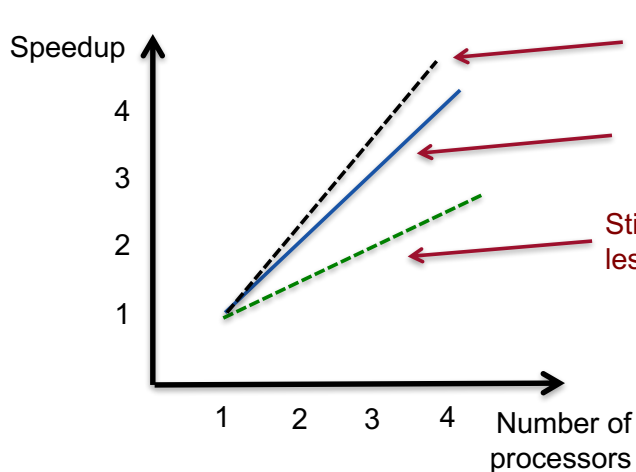
Speedup

86

How much can we improve the performance using parallelization?

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}}$$

Execution time of one program **before** improvement
Execution time **after** improvement



Superlinear speedup. Either wrong, or due to e.g. cache effects.

Linear speedup (or ideal speedup)

Still increased speedup, but less efficient


Danger: **Relative speedup** measures only the same program
True speedup compares also with the best known sequential program,

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

 Part IV
Parallel Processors
and Programs

Amdahl's Law (1/4)

Can we achieve linear speedup?

Divide execution time before improvement into two parts.

Time affected by the improvement of parallelization

Time unaffected of improvement (sequential part)

$$T = T_{\text{affected}} + T_{\text{unaffected}}$$

Execution time after improvement

$$T_{\text{after}} = \frac{T_{\text{affected}}}{N} + T_{\text{unaffected}}$$

Amount of improvement (N times improvement)

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{T_{\text{before}}}{\frac{T_{\text{affected}}}{N} + T_{\text{unaffected}}}$$

This is sometimes referred to as **Amdahl's law**

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Amdahl's Law (2/4)

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{T_{\text{before}}}{\frac{T_{\text{affected}}}{N} + T_{\text{unaffected}}}$$

Exercise: Assume a program consists of an image analysis task, sequentially followed by a statistical computation task. Only the image analysis task can be parallelized. How much do we need to improve the image analysis task to be able to achieve 4 times speedup?

Assume that the program takes 80ms in total and that the image analysis task takes 60ms out of this time.

Solution:

$$4 = 80 / (60 / N + 80 - 60)$$

$$60/N + 20 = 20$$

$$60/N = 0$$

It is impossible to achieve this speedup!

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Amdahl's Law (3/4)

89

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{T_{\text{before}}}{\frac{T_{\text{affected}}}{N} + T_{\text{unaffected}}}$$

Assume that we perform 10 scalar integer additions, followed by one matrix addition, where matrices are 10x10. Assume additions take the same amount of time and that we can only parallelize the matrix addition.

Exercise A: What is the speedup with 10 processors?

Exercise B: What is the speedup with 40 processors?

Exercise C: What is the maximal speedup?

Solution A:

$$(10+10*10) / (10*10/10 + 10) = 5.5$$

Solution B:

$$(10+10*10) / (10*10/40 + 10) = 8.8$$

Solution C:

$$(10+10*10) / (10*10/N + 10) = 11 \text{ when } N \rightarrow \text{infinity}$$

Alternative solution for C

$$(10+10*10) / (10*10/100 + 10) = 10$$

if we assume that one add instruction cannot be parallelized

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Amdahl's Law (4/4)

90

Example continued. What if we change the size of the problem (make the matrices larger)?

| | | Number of processors | |
|------------------|-------|----------------------|-----------------|
| | | 10 | 40 |
| Size of matrices | 10x10 | Speedup 5.5 | Speedup 8.8 |
| | 20x20 | Speedup 8.2 | Speedup 20.5 |

But was not the maximal speedup 11 when $N \rightarrow \text{infinity}$?

Strong scaling = measuring speedup while keeping the problem size fixed.

Weak scaling = measuring speedup when the problem size grows proportionally to the increased number of processors.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Part IV

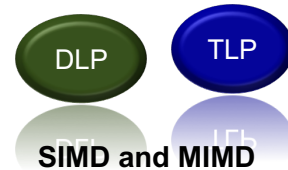
Parallel Processors and Programs



Concurrency and Speedup



Instruction-Level Parallelism



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

What is Instruction-Level Parallelism?

Instruction-Level Parallelism (ILP) may increase performance without involvement of the programmer.

Two main approaches:



1. Deep pipelines with more pipeline stages

If the length of all pipeline stages are balanced, we may increase performance by increasing the clock speed.



2. Multiple issue

A technique where multiple instructions are issued in each in cycle.

ILP may decrease the CPI to lower than 1, or using the inverse metric *instructions per clock cycle (IPC)* increase it above 1.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

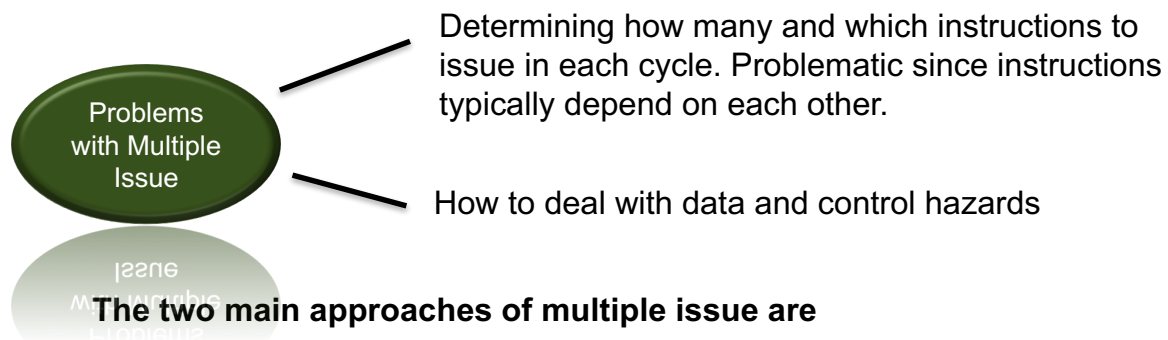
Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

Multiple Issue Approaches



1. Static Multiple Issue

Decisions on when and which instructions to issue at each clock cycle is determined by the compiler.

2. Dynamic Multiple Issue

Many of the decisions of issuing instructions are made by the processor, dynamically, during execution.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Static Multiple Issue (1/2) VLIW

Very Long Instruction Word (VLIW)

processors issue several instructions in each cycle using **issue packages**.

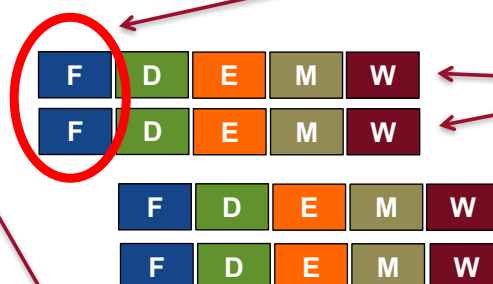
An **issue package** may be seen as one large instruction with multiple operations.

add \$s0, \$s1, \$s2

add \$t0, \$t0, \$0

and \$t2, \$t1, \$s0

lw \$t0, 0(\$s0)



Each issue package has two **issue slots**.

The compiler may insert **no-ops** to avoid hazards.

How is VLIW affecting the hardware implementation?

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

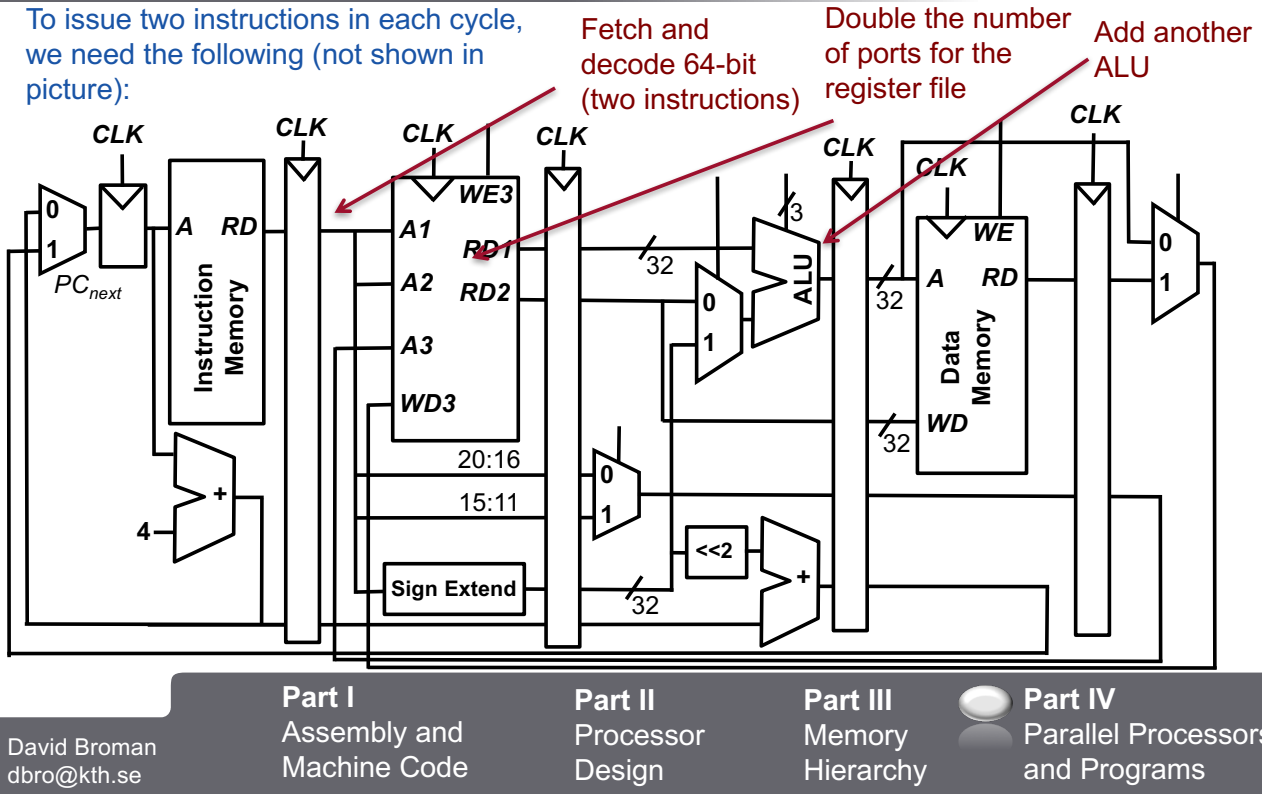
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Static Multiple Issue (2/2) Changing the hardware

95

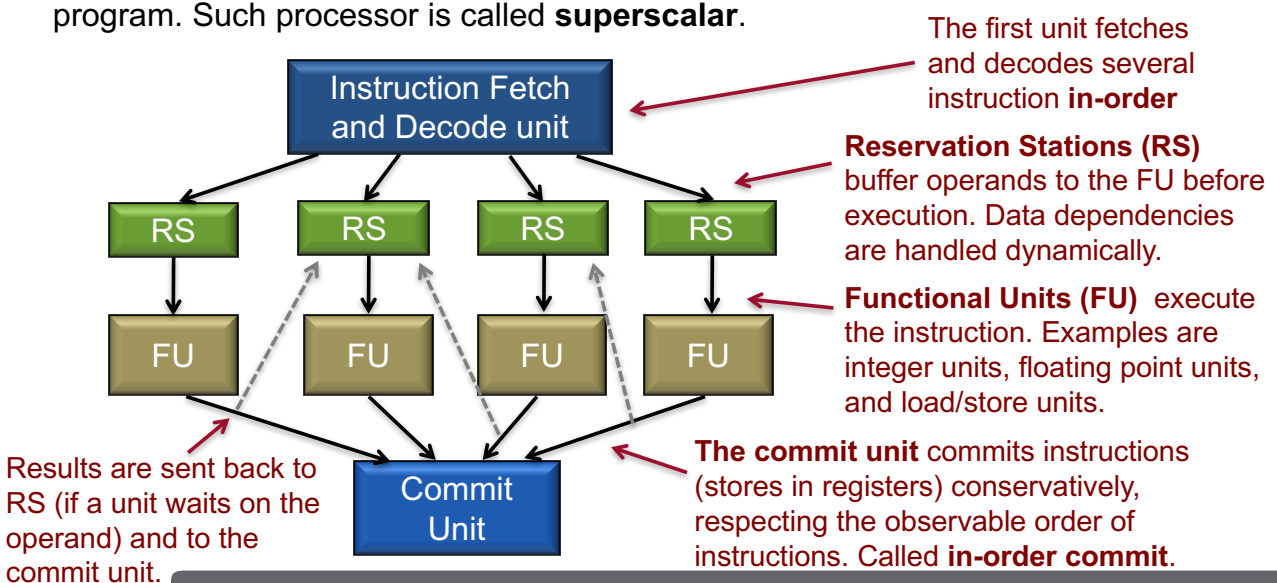
To issue two instructions in each cycle, we need the following (not shown in picture):



Dynamic Multiple Issue Processors (1/2) Superscalar Processors

96

In many modern processors (e.g., Intel Core i7), instruction issuing is performed dynamically by the processor while executing the program. Such processor is called **superscalar**.



If the superscalar processor can reorder the instruction execution order, it is an **out-of-order execution** processor.

```
lw    $t0, 0($s2)
addi  $t1, $t0, 7
```

Example of a **Read After Write (RAW)** dependency (dependency on \$t0). The superscalar pipeline must make sure that the data is available before read.

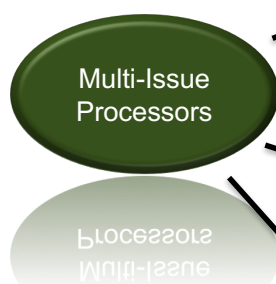
```
sub    $t0, $t1, $s0
addi  $t1, $s0, 10
```

Example of a **Write After Read (WAR)** dependency (dependency on \$t1). If the order is flipped due to out-of-order execution, we have a hazard.

```
addi  $r1, $s0, 10
sub    $t0, $t1, $s0
```

WAR dependencies can be resolved using **register renaming**, where the processor writes to a nonarchitectural renaming register (here in the pseudo asm code called \$r1, not accessible to the programmer)

Note that out-of-order processor is not rewriting the code, but keeps track of the renamed registers during execution.



VLIW processors tend to be more **energy efficient** than superscalar out-of-order processors (less hardware, the compiler does the job)

Superscalar processors with dynamic scheduling can **hide some latencies** that are not statically predictable (e.g., cache misses, dynamic branch predictions).

Although modern processors issues 4 to 6 instructions per clock cycle, few applications results in IPC over 2. The reason is dependencies.

Intel Microprocessors, some examples

| Processor | Year | Clock Rate | Pipeline Stages | Issue Width | Cores | Power |
|----------------------------|------|------------|-----------------|-------------|-------|-------|
| Intel 486 | 1989 | 25 MHz | 5 | 1 | 1 | 5 W |
| Intel Pentium | 1993 | 66 MHz | 5 | 2 | 1 | 10W |
| Intel Pentium Pro | 1997 | 200 MHz | 10 | 3 | 1 | 29 W |
| Intel Pentium 4 Willamette | 2001 | 2000 MHz | 22 | 3 | 1 | 75W |
| Intel Pentium 4 Prescott | 2004 | 3600 MHz | 31 | 3 | 1 | 103W |
| Intel Core | 2006 | 2930 MHz | 14 | 4 | 2 | 75W |
| Intel Core i5 Nehalem | 2010 | 3300 MHz | 14 | 4 | 1 | 87W |
| Intel Core i5 Ivy Bridge | 2012 | 3400 MHz | 14 | 4 | 8 | 77W |

Clock rate increase stopped (the power wall) around 2006

Pipeline stages first increased and then decreased, but the number of cores increased after 2006.

The power consumption peaked with Pentium 4

Source: Patterson and Hennessey, 2014, page 344.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

Part IV

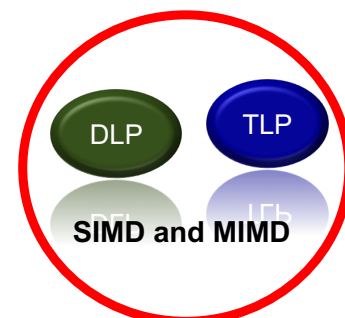
Parallel Processors and Programs



**Concurrency and
Speedup**



**Instruction-Level
Parallelism**



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

Main Classes of Parallelisms



Data-Level Parallelism (DLP)

Many data items can be processed at the same time.



Example – Sheep shearing

Assume that sheep are data items and the task for the farmer is to do sheep shearing (remove the wool). Data-level parallelism would be the same as using several farm hands to do the shearing.



Task-Level Parallelism (TLP)

Different tasks of work that can work in independently and in parallel



Example – Many tasks at the farm

Assume that there are many different things that can be done on the farm (fix the barn, sheep shearing, feed the pigs etc.) Task-level parallelism would be to let the farm hands do the different tasks in parallel.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

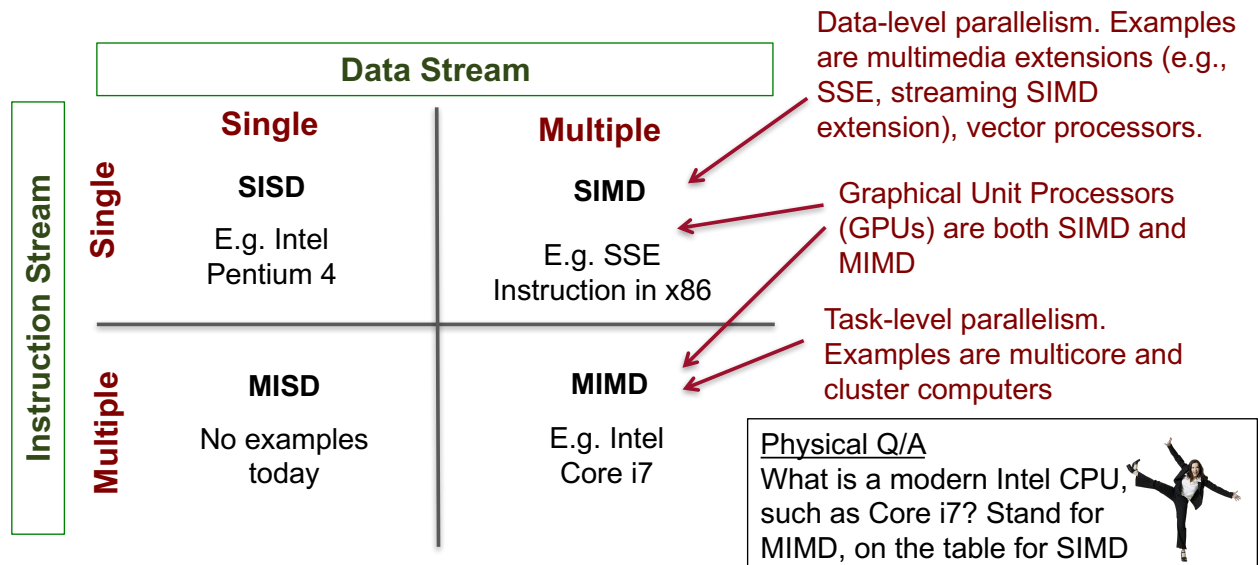
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

SISD, SIMD, and MIMD

An old (from the 1960s) but still very useful classification of processors uses the notion of instruction and data streams.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

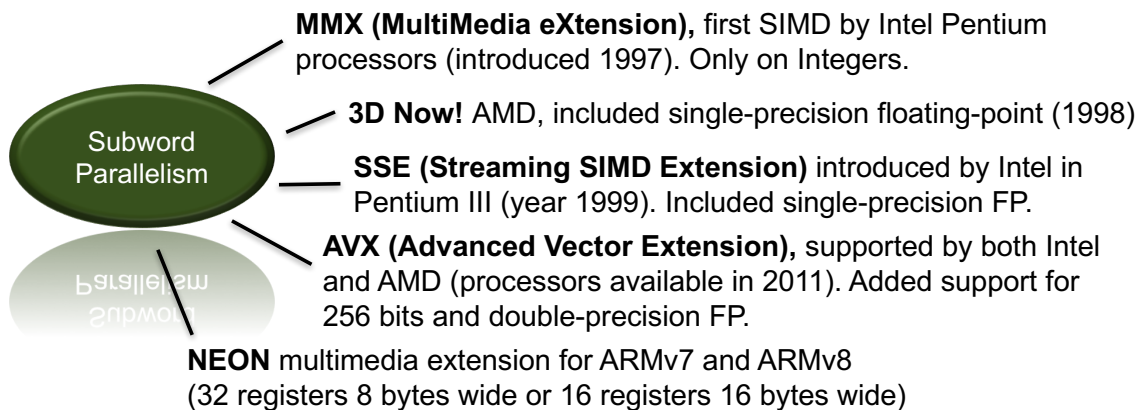
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Subword parallelism is when a wide data word is operated on in parallel.

This is the same as SIMD or data-level parallelism.

One instruction operates on multiple data items.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

In SSE (and the later version SSE2), assembly instructions are using two-operand format.

```
addpd %xmm0, %xmm4
```

meaning: $\%xmm4 = \%xmm4 + \%xmm0$
Note the reversed order.

Registers (e.g. $\%xmm4$) are 128-bits in SSE/SSE2.

Added the "v" for vector to distinguish AVX from SSE and renamed registers to $\%ymm$ that are now 256-bit

"pd" means Packed Double precision FP. It can operate on as many FP that fits in the register

```
vaddpd %ymm0, %ymm1, %ymm4
vmovapd %ymm4, (%r11)
```

Question: How many FP additions does **vaddpd** perform in parallel?

Answer: 4

Moves the result to the memory address stored in $\%r11$ (a 64-bit register). Stores the four 64-bit FP in consecutive order in memory.

AVX introduced three-operand format
Meaning: $\%ymm4 = \%ymm0 + \%ymm1$

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

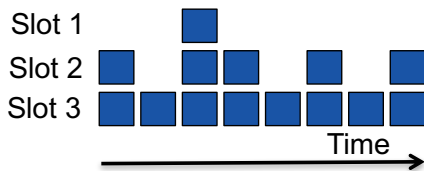
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Recall the idea of a multi-issue uniprocessor

105

Thread A Thread B Thread C



Typically, all functional units cannot be fully utilized in a single-threaded program (white space is unused slot/functional unit).

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

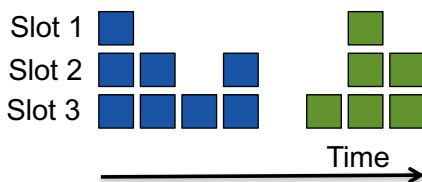
Hardware Multithreading

106

In a **multithreaded processor**, several hardware threads share the same functional units.

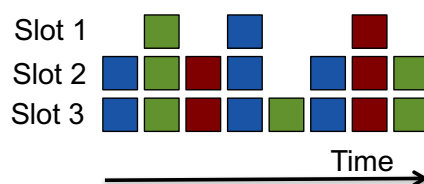
The purpose of multithreading is to hide latencies and avoid stalls due to cache misses etc.

Thread A Thread B Thread C



Coarse-grained multithreading, switches threads only at costly stalls, e.g., last-level cache misses.

Cannot overcome throughput losses in short stalls.



Fine-grained multithreading switches between hardware threads every cycle. Better utilization.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

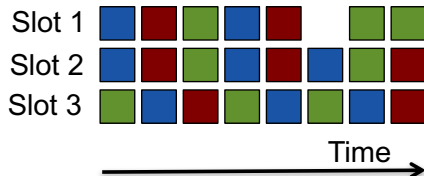


Part IV
Parallel Processors
and Programs

Simultaneous multithreading (SMT)

Simultaneous multithreading (SMT) combines multithreading with a multiple-issue, dynamically scheduled pipeline.

Thread A Thread B Thread C



Can fill in the holes that multiple-issue cannot utilize with cycles from other hardware threads. Thus, better utilization.

Example: **Hyper-threading** is Intel's name and implementation of SMT. That is why a processor can have 2 real cores, but the OS shows 4 cores (4 hardware threads).

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Shared Memory Multiprocessor (SMP)

A Shared Memory Multiprocessor (SMP) has a *single physical address space* across all processors.

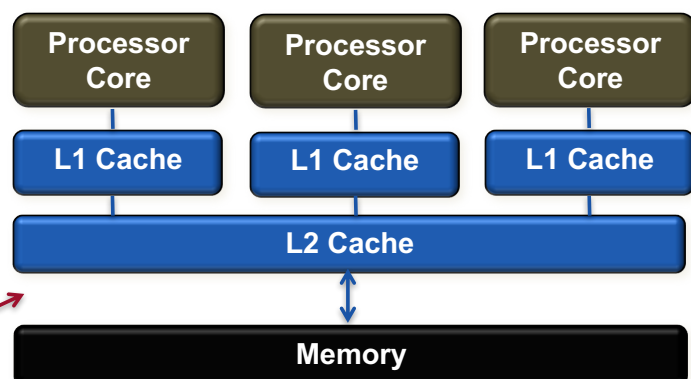
An SMP is almost always the same as a **multicore processor**.

In a **uniform memory access (UMA)** multiprocessor, the latency of accessing memory does not depend on the processor.

In a **nonuniform memory access (NUMA)** multiprocessor, memory can be divided between processor and result in different latencies.

Processors (cores) in a SMP communicate via **shared memory**.

Alternative: Network on Chip (NoC)



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

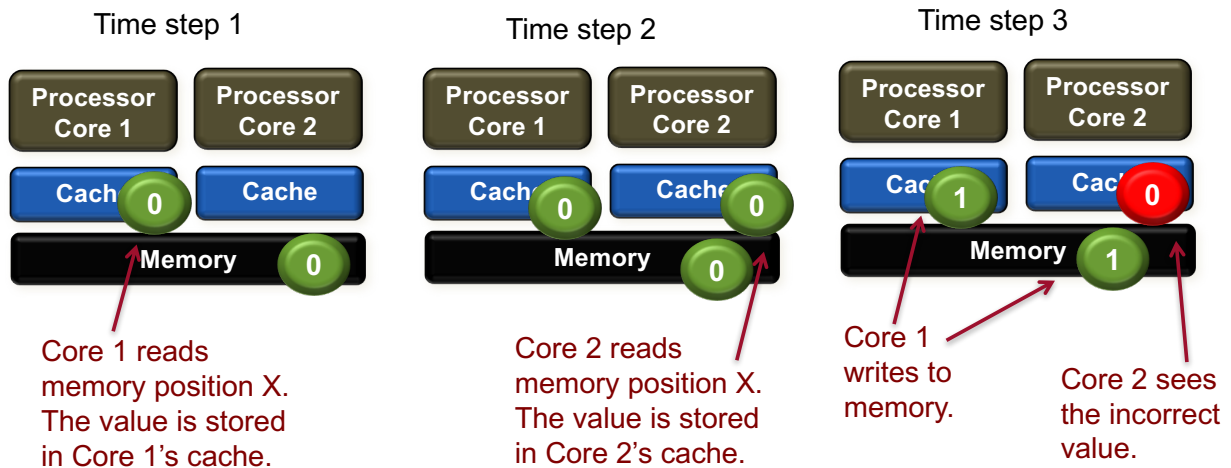
Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Cache Coherence

Different cores' local caches could result in that different cores see different values for the same memory address.

This is called the **cache coherency problem**.



David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

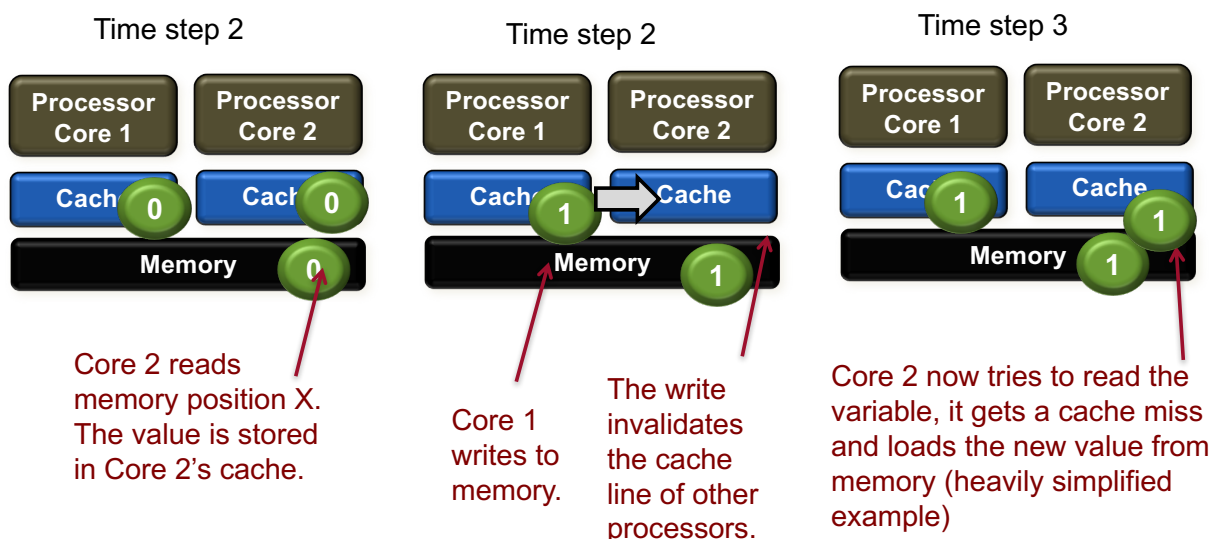
Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Snooping Protocol

Cache coherence can be enforced using a cache coherence protocol. For instance a *write invalidate protocol*, such as the **snooping protocol**.



David Broman
dbro@kth.se

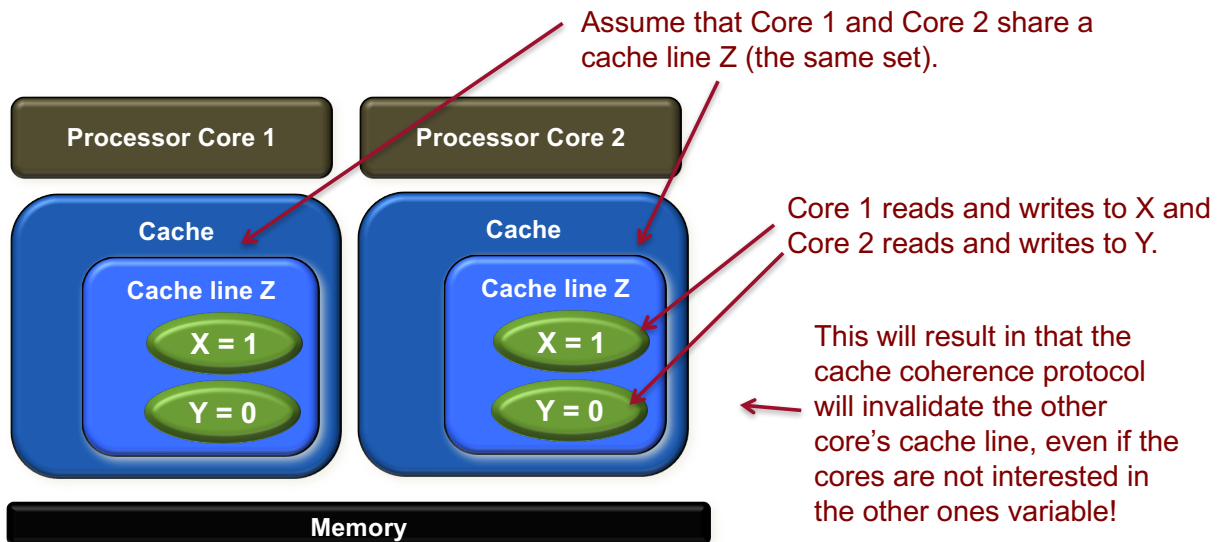
Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

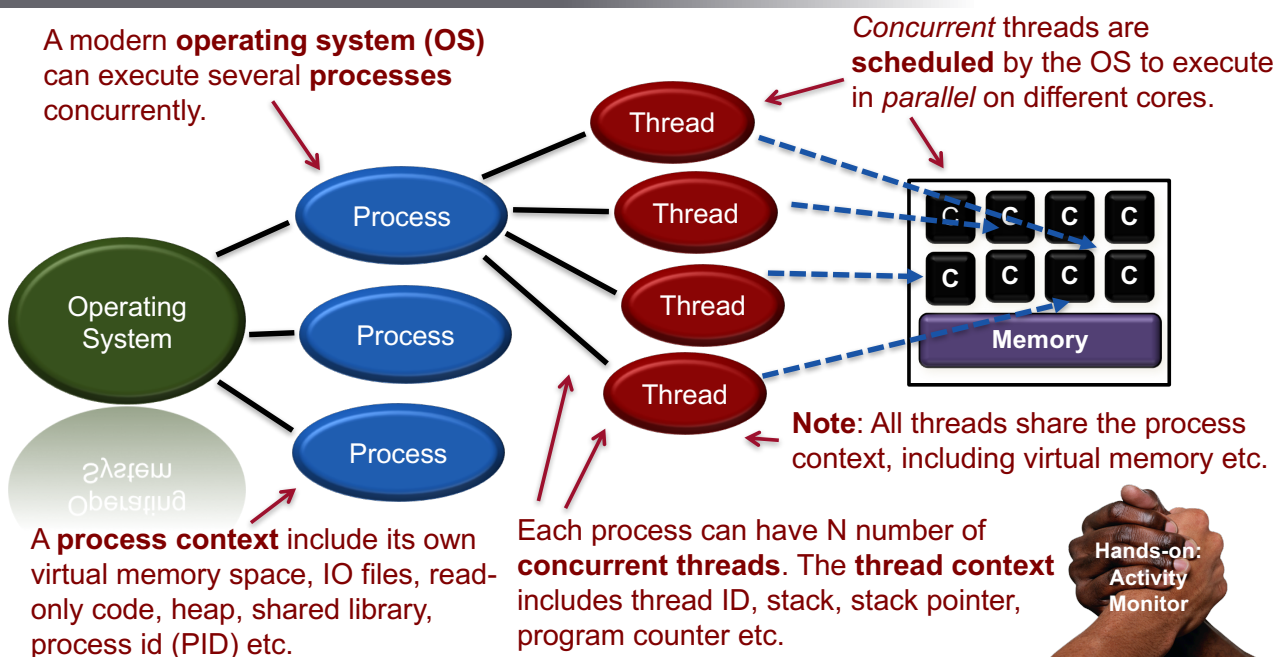
Part IV
Parallel Processors
and Programs

False Sharing



This is called **false sharing**.

Processes, Threads, and Cores



General Matrix Multiplication (GEMM)

Simple matrix multiplication

Uses matrix size n as a parameter and single dimension for performance.

```
void dgemm(int n, double* A, double* B, double* C){
    for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j){
            double cij = C[i+j*n];
            for(int k = 0; k < n; k++)
                cij += A[i+k*n] * B[k+j*n];
            C[i+j*n] = cij;
        }
}
```

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

Parallelizing GEMM

Unoptimized

Unoptimized C version (previous page). Using one core.

1.7 GigaFLOPS (32x32)
0.8 GigaFLOPS (960x960)

SIMD

Use AVX instructions `vaddpd` and `vmulpd` to do 4 double precision floating-point operations in parallel.

6.4 GigaFLOPS (32x32)
2.5 GigaFLOPS (960x960)

ILP

AVX + unroll parts of the loop, so that the multiple-issue, out-of-order processor have more instructions to schedule.

14.6 GigaFLOPS (32x32)
5.1 GigaFLOPS (960x960)

Cache

AVX + unroll + blocking (dividing the problem into submatrices). This avoids cache misses.

13.6 GigaFLOPS (32x32)
12.0 GigaFLOPS (960x960)

Multi-core

AVX + unroll + blocking + multi core (multithreading using OpenMP)

23 GigaFLOPS (960x960, 2 cores)
44 GigaFLOPS (960x960, 4 cores)
174 GigaFLOPS (960x960, 16 cores)

Experiment by P&H on a 2.6GHz Intel Core i7 with Turbo mode turned off.

For details see P&H, 5th edition, sections 3.8, 4.12, 5.14, and 6.12

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy



Part IV
Parallel Processors
and Programs

“For x86 computers, we expect to see two additional cores per chip every two years and the SIMD width to double every four years.”

Hennessy & Patterson, Computer Architecture – A Quantitative Approach, 5th edition, 2013 (page 263)



We must understand and utilize **both MIMD and SIMD** to gain maximal speedups in the future, although MIMD (multicore) has gained much more attention lately.

The previous example showed that **the way** we program for **ILP** and **caches**, also matters significantly.

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs

Some key take away points:

- **Amdahl's law** can be used to estimate maximal speedup.
- **Instruction-Level Parallelism (ILP)** has been very important for performance improvements over the years.
- **SIMD** can efficiently parallelize problems that have data-level parallelism
- **MIMD, Multicores, and Clusters** can be used to parallelize problems that have task-level parallelism.
- In the future, we should try to **combine** and use both **SIMD** and **MIMD**!



Thanks for listening!

David Broman
dbro@kth.se

Part I
Assembly and
Machine Code

Part II
Processor
Design

Part III
Memory
Hierarchy

Part IV
Parallel Processors
and Programs