Systems@ETH Zürich

Exercise Session 11

Computer Architecture and Systems Programming

Architectural Optimizations

Autumn Semester 2024

Disclaimer

Systems@ETH zürich

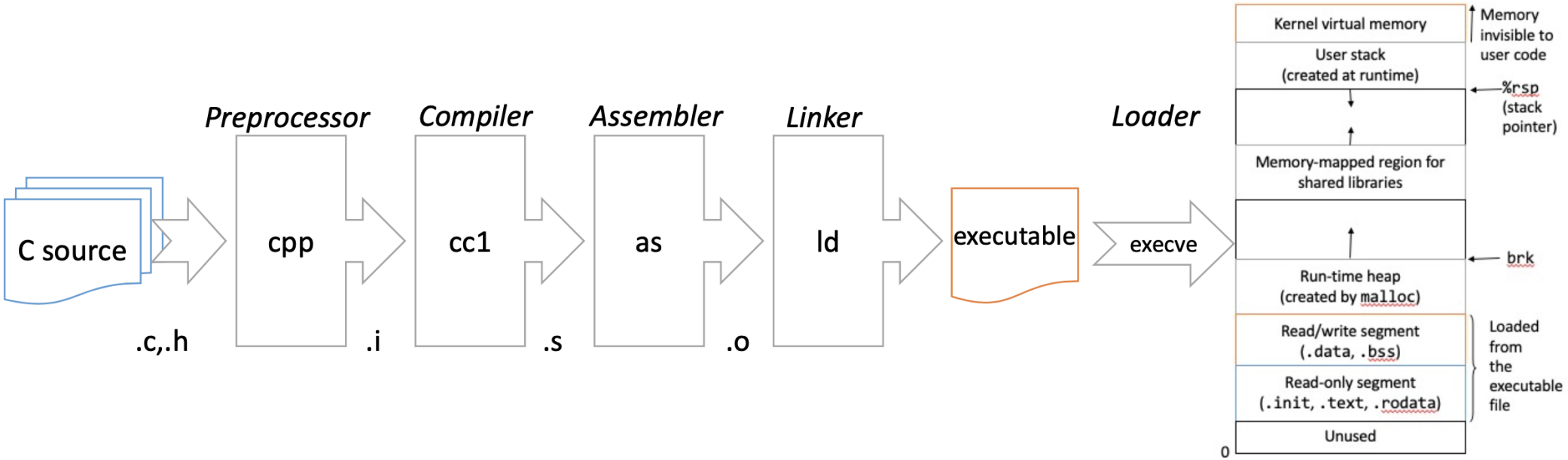
- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- **Kahoots**: now on website n.ethz.ch/~falkbe/
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

In this session…

- Processor Architecture (DDCA)
- Code and Compiler Optimisations (Strength reduction, Inlining etc.)
- Hardware Optimisation (Loop unrolling, reassociation, etc.)
- Exam Questions on Optimisations
- Measuring performance: Cycles Per Element (CPE) & friends
- Improving µ-arch performance pt.1: instruction parallelism
- Improving µ-arch performance pt.2: data parallelism
- Support your local µ-arch: compile-time optimizations
- Outlook on Assignment 9

Where are we in the course

Recall: how C code runs as a process on CPU



Where are we in the course

Systems@ETH zürich

- **Know by now**: How to write C, how this gets compiled down to an executable (preprocessor, compiler, assembler, linker and loader)
- **Compiler Optimisation Lecture**: How to create a **faster** executable (purely code based)
- **Computer Architecture**: How these instructions get **executed on hardware**, also possible improvements via better hardware (multiple execution units, OoO etc.)
- **Future**: Program gets loaded into memory, but **what is memory**? (Caches, Virtual Memory); **Exceptions**, **Multicore, Devices**

Recap

Processor Architecture

Remark

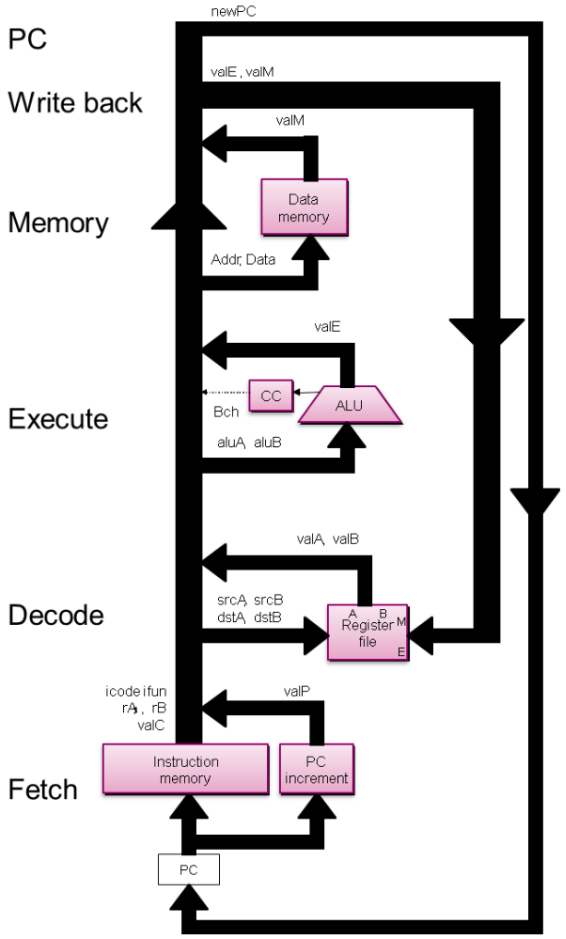
- What you will see now goes **at times a bit beyond** what they teach you in SPCA: it's a lot of **DDCA but its fundamental to understand** what happens with the assembly language inside a processor

What you saw in the lecture: seq proc.

Systems@ETH Zürich

Sequential processor stages

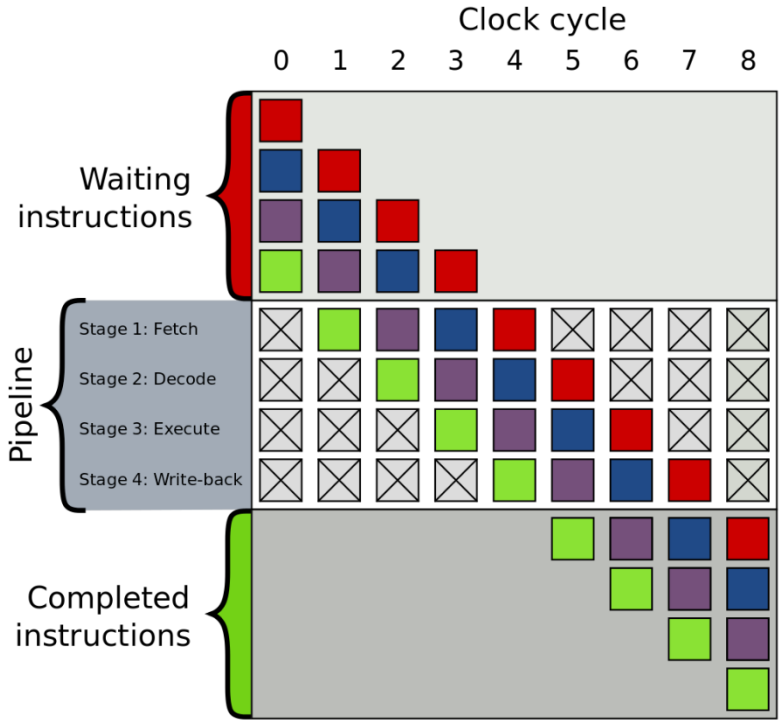
- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter

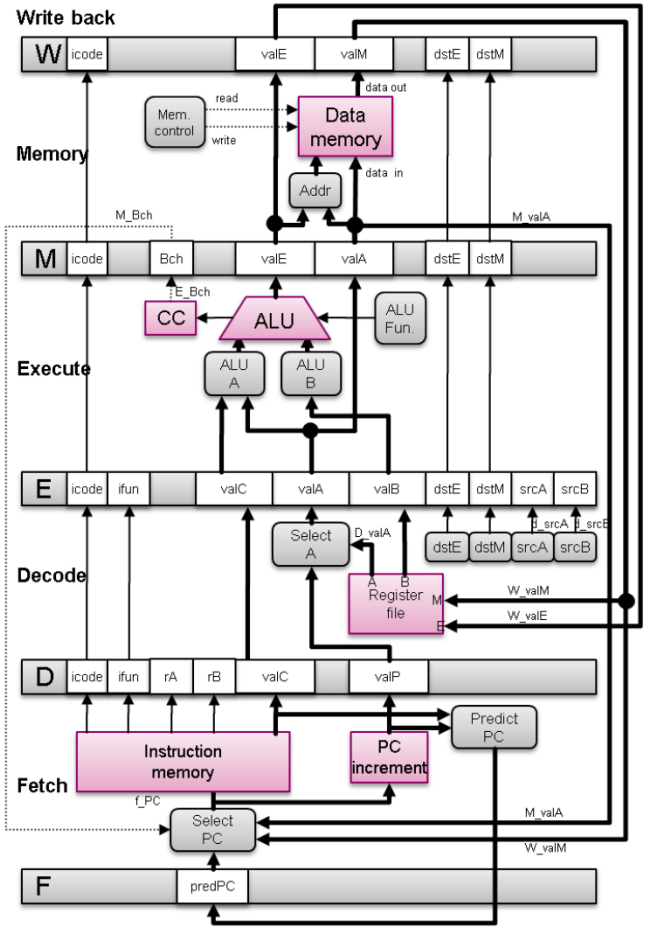


What you saw in the lecture: pipelined proc.

Systems@ETH Zürich

Pipelined hardware

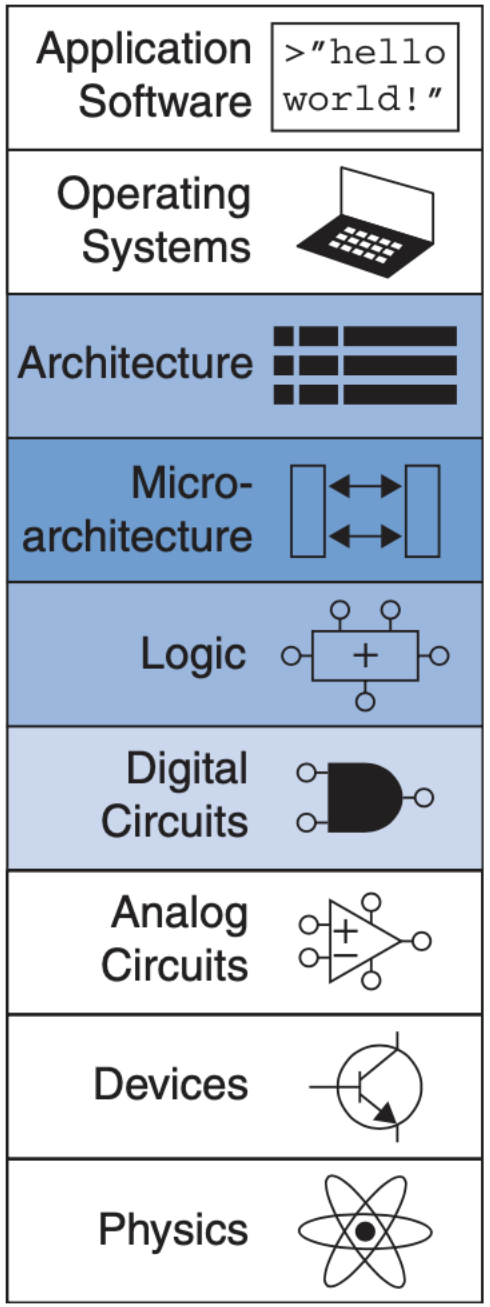




Remark

- What you saw until now: x86, an **architecture**
- Specified **instructions** (cmp, jmp, etc.), how to **address memory**, **addressing modes** etc.

- Now we look at **microarchitecture**: the actual **implementation of a architecture** (like x86, MIPS, ARM etc.) in hardware



Remark

- The processor we are building now is for a **different architecture** called MIPS, very similar to x86
- This simply means there are different **assembly instructions, different addressing modes, register names etc.** but the **underlying concepts** stay the same

Systems@ETH Zürich

Recap

ISAs: Assembly

Remark

- Before we look at how the code actually gets executed, lets before see how **assembly relates to this**
- It boils down to difference between:
  - **ISA:** abstract model, defining instructions, data types, registers
  - **Michroarchitecture:** the actual implementation of an ISA in hardware

- We are going to look at **ISA MIPS** for the processor I will build with you here

Instruction Set Architecture

Systems@ETH zürich

- **Examples for ISAs**: x86-64, ARM (Advanced RISC Machines), RISC-V, MIPS, PowerPC, SPARC, Z/Architecture (IBM Mainframes)

x86 / x86-64	CISC	Desktops, laptops, servers (Intel, AMD)
ARM	RISC	Mobile devices, embedded systems, some laptops
RISC-V	RISC	Open-source processors, research, IoT, embedded
MIPS	RISC	Embedded systems, networking devices (less common now)
PowerPC	RISC	Older Macs, embedded, some servers
SPARC	RISC	Enterprise servers, scientific computing

Instruction Set Architecture

- **ISAs** thus define how our assembly code looks (because this is inherently what an ISA describes):
- MIPS (LHS), x86 (RHS)

```
sll  $t0, $s0, 2     # $t0 = f * 4
add  $t0, $s6, $t0   # $t0 = &A[f]
sll  $t1, $s1, 2     # $t1 = g * 4
add  $t1, $s7, $t1   # $t1 = &B[g]
lw   $s0, 0($t0)     # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

```
Q1:     proc near
        push    sPassword
        call    _strlen
        pop     ecx
        mov     esi, eax
        mov     ebx, offset sMyPassword
        push    ebx
        call    _strlen
        pop     ecx
        cmp     esi, eax
        jz      short loc_4012B2
        xor     eax, eax
        jmp     short end_proc
loc_4012B2:
        push    esi
        push    ebx
        push    sPassword
        call    _strcmp
        add     esp, 8
        test    eax, eax
        jnz     short loc_4012CC
        mov     eax, 1
        jmp     short end_proc
loc_4012CC:
        xor     eax, eax
end_proc:
        pop     esi
        pop     ebx
        pop     ebp
        retn
endp
```

Instruction Set Architecture (ISA): MIPS

Systems@ETH Zürich

- **Instructions Examples**: Left code is high level language (C, C++, Java), RHS in MIPS
- First Part: called **mnemonic** indicates what to perform, operation is performed on b,c the **source operands** and stored in the **destination** operand

Code Example 6.1 ADDITION

High-Level Code	MIPS Assembly Code
`a = b + c;`	`add a, b, c`

Instruction Set Architecture (ISA): MIPS

Systems@ETH zürich

- The machine (michroarchitecture) provides "**registers**", things were we can store stuff: we can access them with $ in MIPS, with % in x86 => we have 32 registers in mips

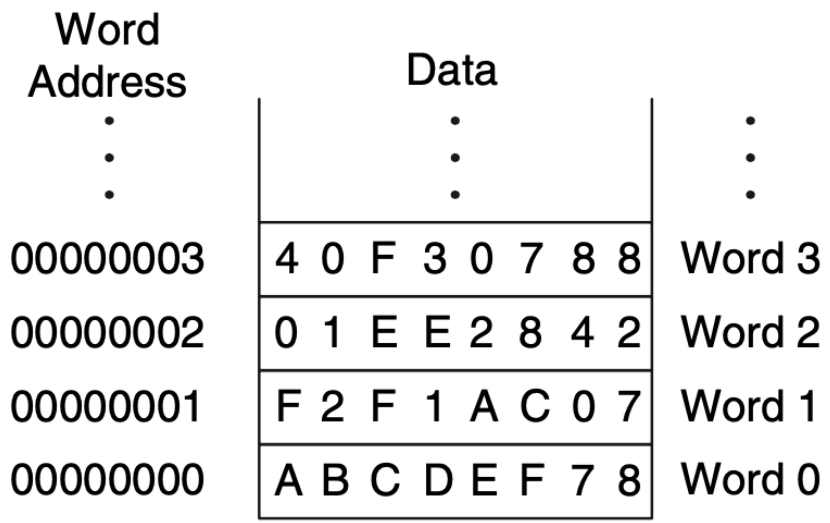
Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	MIPS Assembly Code

`a = b + c – d;`	`# $s0 = a, $s1 = b, $s2 = c, $s3 = d`
	
	`sub $t0, $s2, $s3  # t = c – d`
	`add $s0, $s1, $t0  # a = b + t`

Instruction Set Architecture (ISA): MIPS

- Since we only have limited number of registers: also have memory which we can access (Stack, Heap in x86)



Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Instruction Set Architecture (ISA): MIPS

Systems@ETH zürich

- Assembly is for **humans** to read: but machines only understand **machine code**: need to bring assembly into **machine language**
- **Idea:** encode all instruction as words that can be stored in memory, all 32bit
- MIPS has 3 type of Instructions: R-type, I-type, J-type (Register, Immediate, Jump)

Instruction Set Architecture (ISA): MIPS

- **R-type (Register type)**: uses 3 registers as operands, 2 as source 1 as destination
- Operation to be performed is encoded in “op” and “funct” field: all R-type instr. Have opcode=0 and funct is 32 for add and 34 for substract
- Operands encoded in rs, rt and rd (rs and rt source, rd dest)

R-type

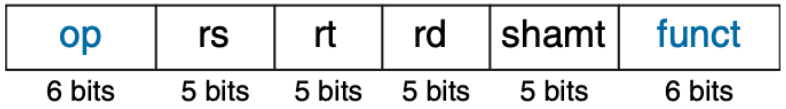


Figure 6.5 R-type machine instruction format

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
`sub $t0, $t3, $t5`	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.6 Machine code for R-type instructions

Instruction Set Architecture (ISA): MIPS

- **I-type instructions**: different interpretation of the bits but conceptually the same

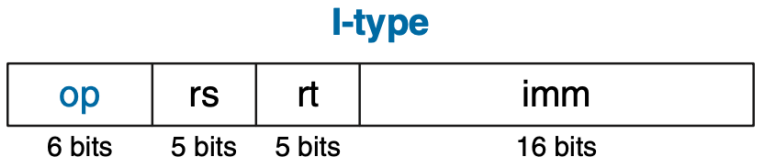


Figure 6.8 I-type instruction format

Assembly Code	op	rs	rt	imm	op	rs	rt	imm	
	Field Values				Machine Code				
`addi $s0, $s1, 5`	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
`addi $t0, $s3, -12`	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
`lw $t2, 32($0)`	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
`sw $s1, 4($t1)`	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

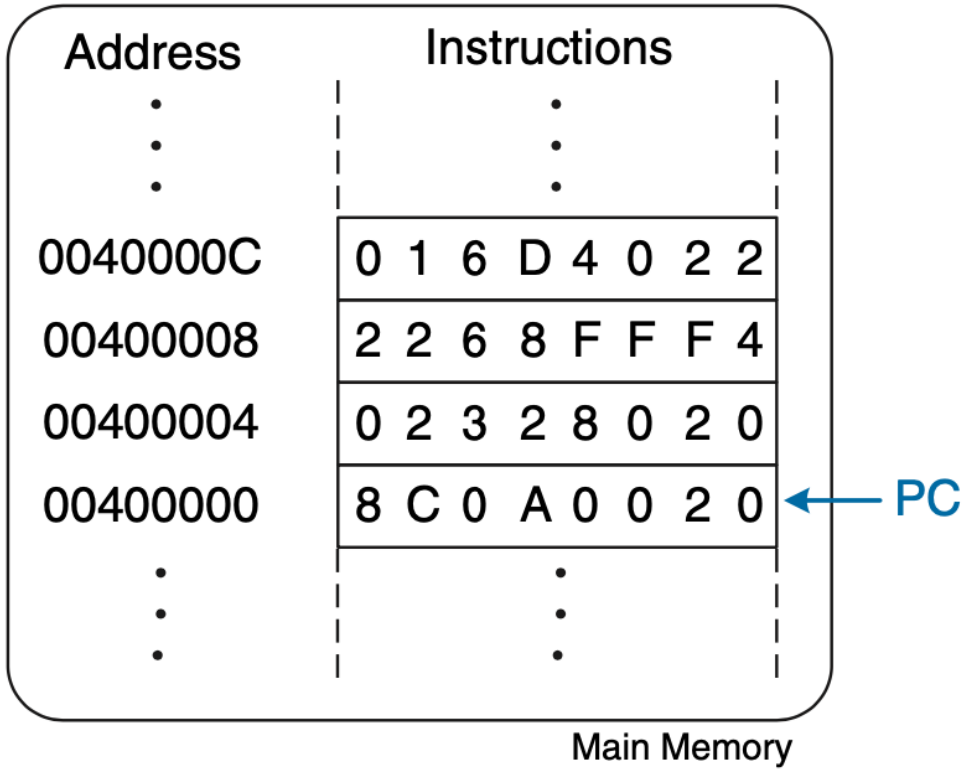
Figure 6.9 Machine code for I-type instructions

Instruction Set Architecture (ISA): MIPS

- Now we can store an entire program in memory

Assembly Code	Machine Code
`lw $t2, 32($0)`	0x8C0A0020
`add $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub $t0, $t3, $t5`	0x016D4022

Stored Program



Systems@ETH Zürich

Recap

Michroarchitecture: Single Cycle Processor

Michroarchitecture

Systems@ETH Zürich

- **Microarchitecture**: specific arrangement of ALUs, FSMs, Memories etc.

- One ISA Like MIPS can have **many different microarchitectures** with different performance, cost and complexity
- All run the same programs since all architectures share the same “language” (ISA) but they can vary in cost, performance and complexity

Type of Processors

Systems@ETH Zürich

- **Single Cycle**: executes an entire instruction in one cycle
- Easy to explain, simple control
- Cycle time is limited by the slowest instruction

- **Pipelined Microarchitecture**: applied pipelining to the single cycle michroarchitecture
- Can execute multiple instruction simultaneously, must handle **dependencies** between those instructions

- **All commercial high-performance processors use pipelining today**

Type of Processors

Systems@ETH zürich

- **Single Cycle**: executes an entire instruction in one cycle
- Easy to explain, simple control
- Cycle time is limited by the slowest instruction

- **Pipelined Microarchitecture**: applied pipelining to the single cycle michroarchitecture
- Can execute multiple instruction simultaneously, must handle **dependencies** between those instructions

- **All commercial high-performance processors use pipelining today**

Single Cycle Processor

Systems@ETH Zürich

- Divide microarchitecture in: **datapath** and **controlpath**

- **Datapath**: operates on words of data (here 32bit): contains memories, registers, ALUs
- **Controlpath**: Recieves the current instruction from datapath and tells the datapath how to execute the instruction: selects multiplexer select, register enable, memory write signal

Single Cycle Processor

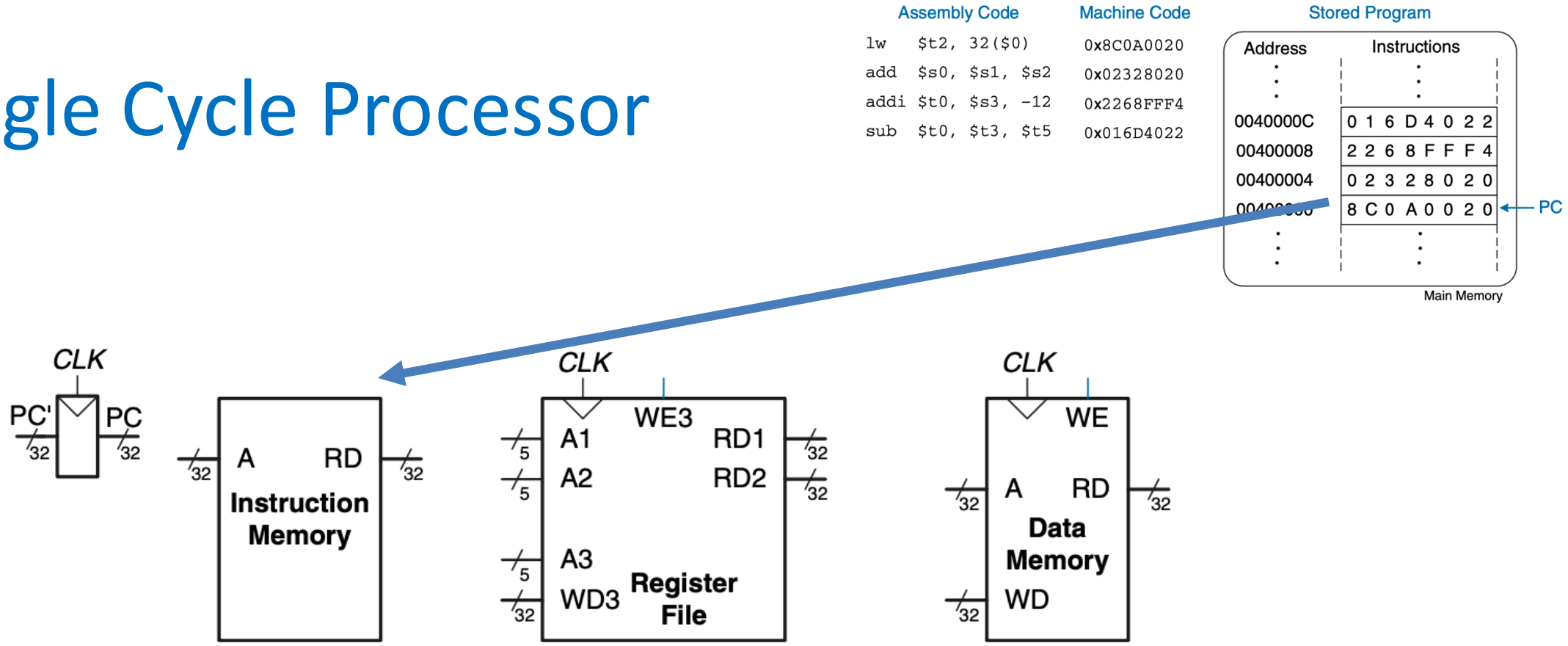


Figure 7.1 **State elements of MIPS processor**

- **Instruction memory**: Holds the instructions (movq %rbp, %rsp, encoded in bit i.e. 0/1s)
- **Data memory**: holds data
- **Program Counter PC** (%rip in x86): contains address of the instruction to be executed:

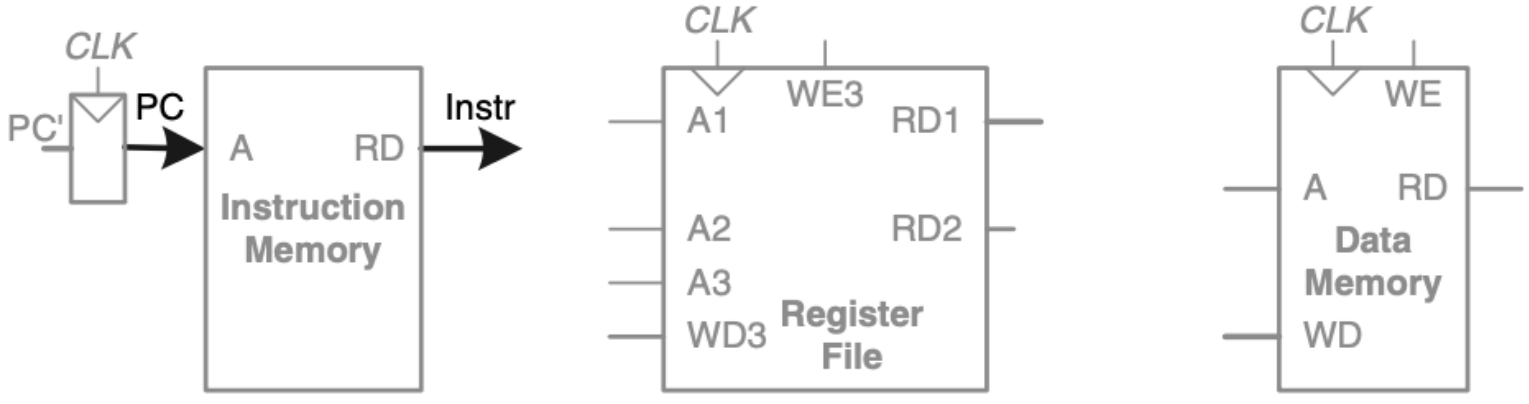
Single Cycle Processor

Systems@ETH Zürich

- **Lets** implement the MIPS instructions: **R-type, I-type** s.t. our processor can execute them

- I-type
  - Lw (load woard)
  - Sw (store woard)
- R-type (binary ops: add, sub, …)
- Beq instruction (beq, bne, ..)

Single Cycle Processor



- **PC** register contains instruction to execute: fetch instruction (read it)

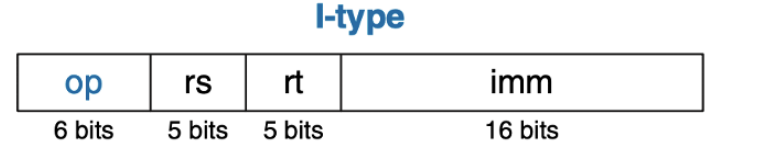
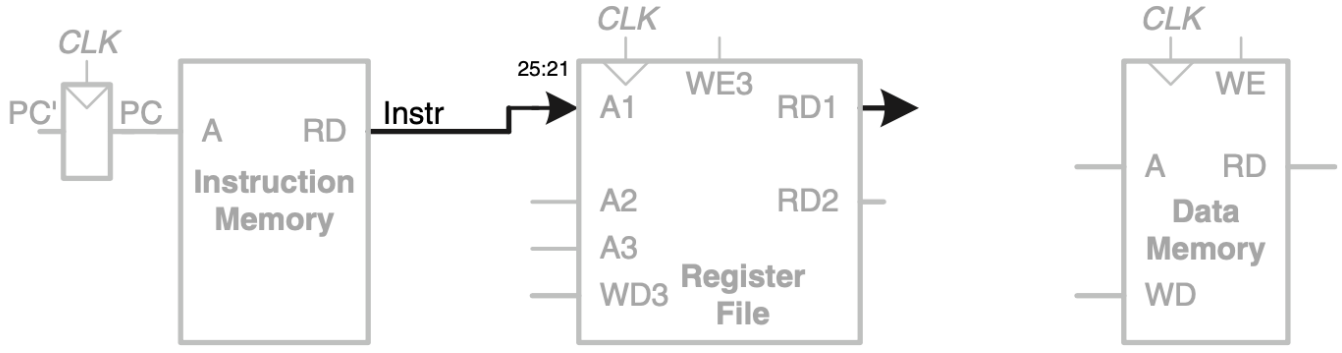


Figure 6.8 **I-type instruction format**

`lw $t2, 32($0)`	35	0	10	32	100011 00000 01010 0000 0000 0010 0000	(0x8C0A0020)

Single Cycle Processor



- **Read** source register containing the base address in **register file**($0): specified in “rs” field, i.e. Instr 25:21

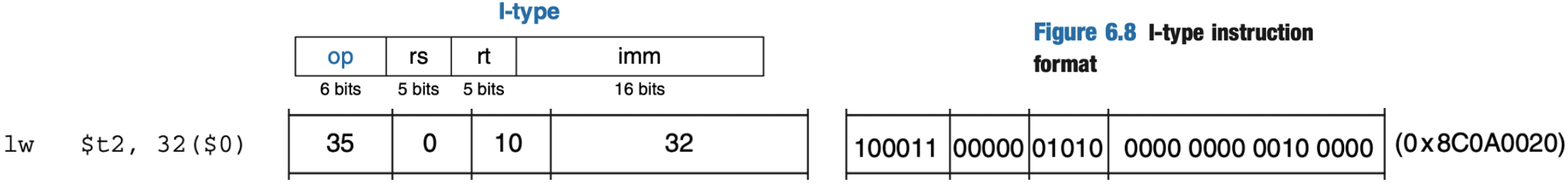
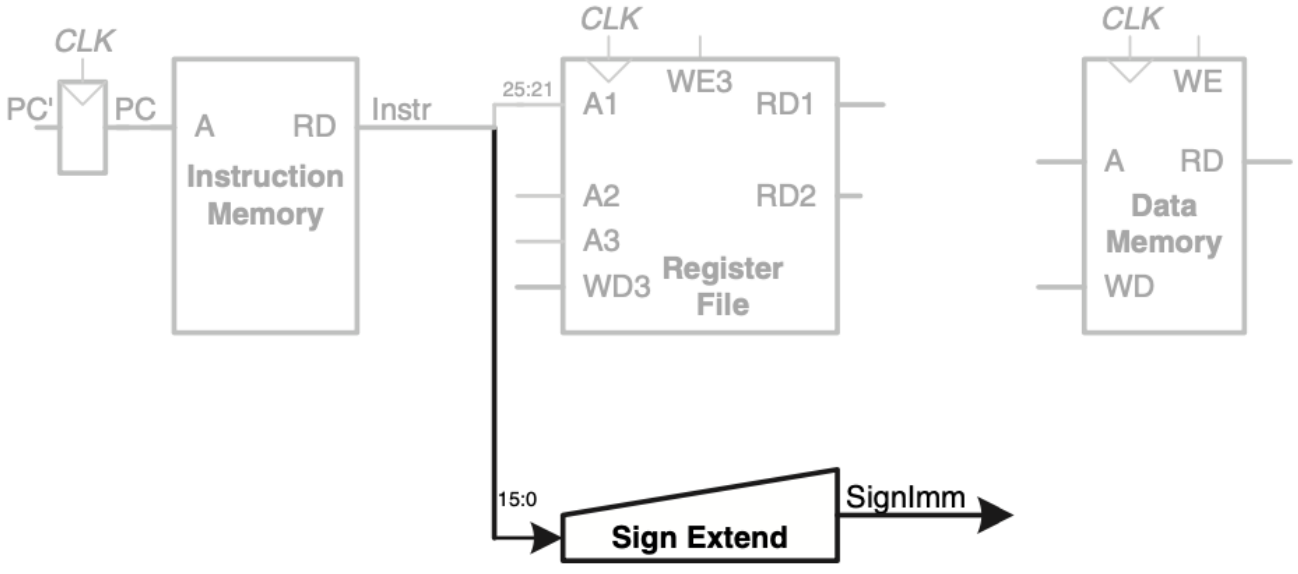


Figure 6.8 I-type instruction format

Single Cycle Processor



- **Lw** instruction also needs offset (here 32): offset is 16-bit immediate, needs to be **sign extended to 32 bits**: Immediate is in bits 15:0

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Figure 6.8 I-type instruction format

`lw $t2, 32($0)`

35	0	10	32

100011	00000	01010	0000 0000 0010 0000

(0x8C0A0020)

Single Cycle Processor

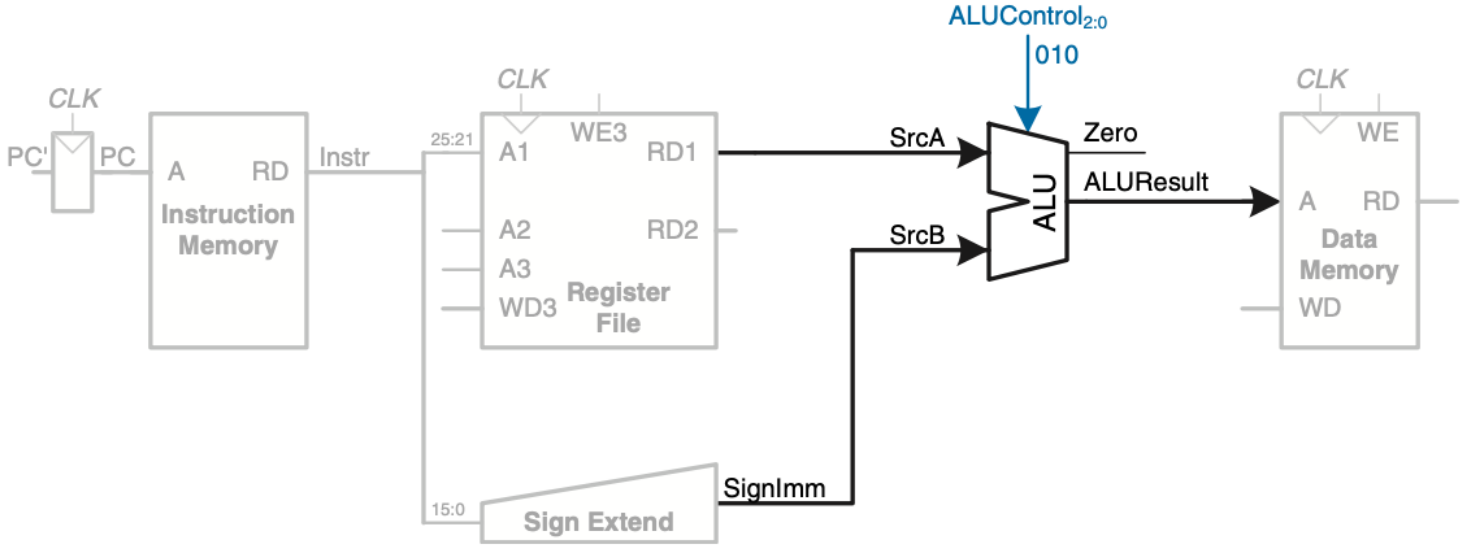


Figure 7.5 **Compute memory address**

- **Now** we need to **add sign extended offset** (32) **to base address** ($0):

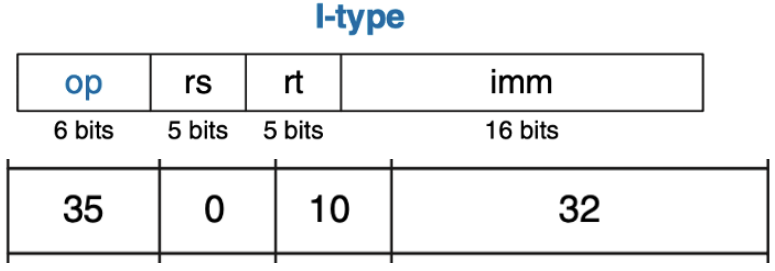


Figure 6.8 **I-type instruction format**

```
lw   $t2, 32($0)
```

35	0	10	32
100011	00000	01010	0000 0000 0010 0000

(0x8C0A0020)

Single Cycle Processor

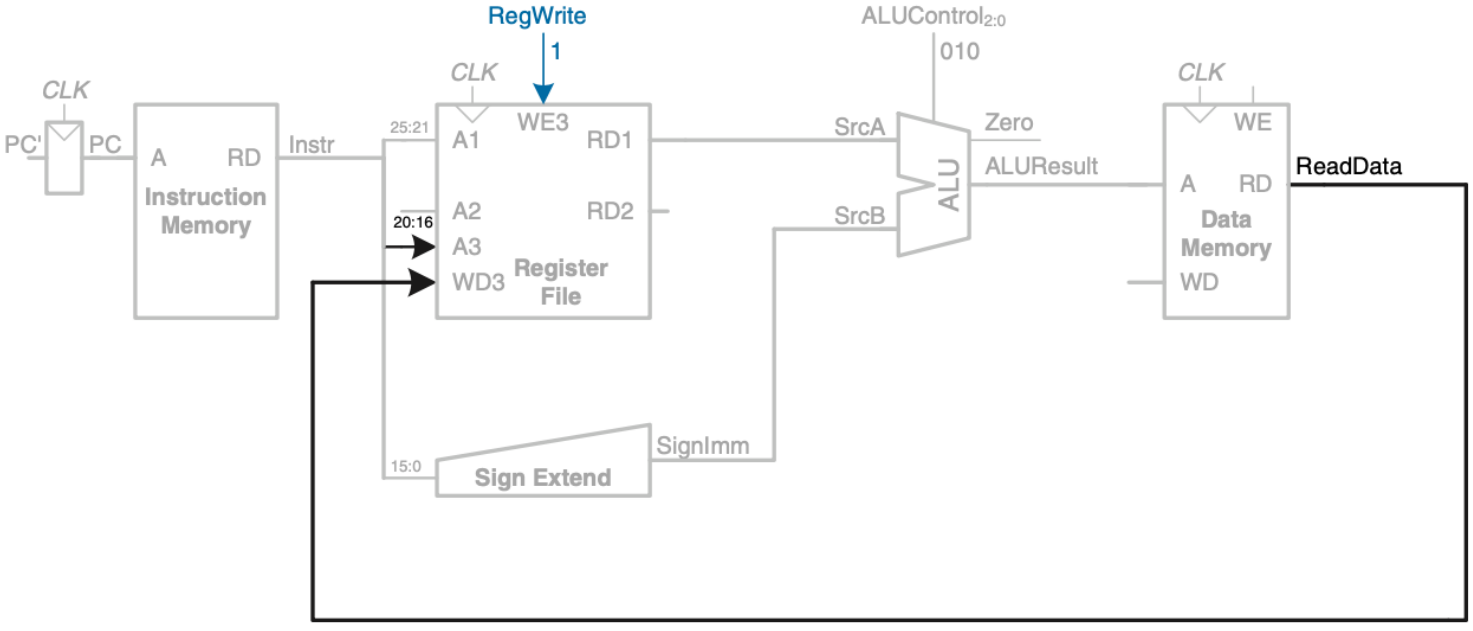


Figure 7.6 Write data back to register file

- **Read** data from memory, and load it into the **second register** which is specified in "rt" i.e. bits 20:16

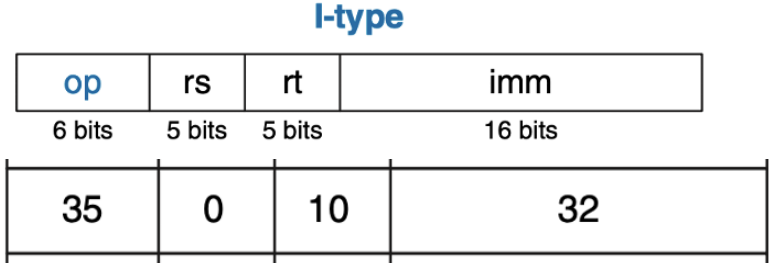


Figure 6.8 I-type instruction format

```
lw   $t2, 32($0)
```

35	0	10	32
100011	00000	01010	0000 0000 0010 0000

(0x8C0A0020)

Single Cycle Processor

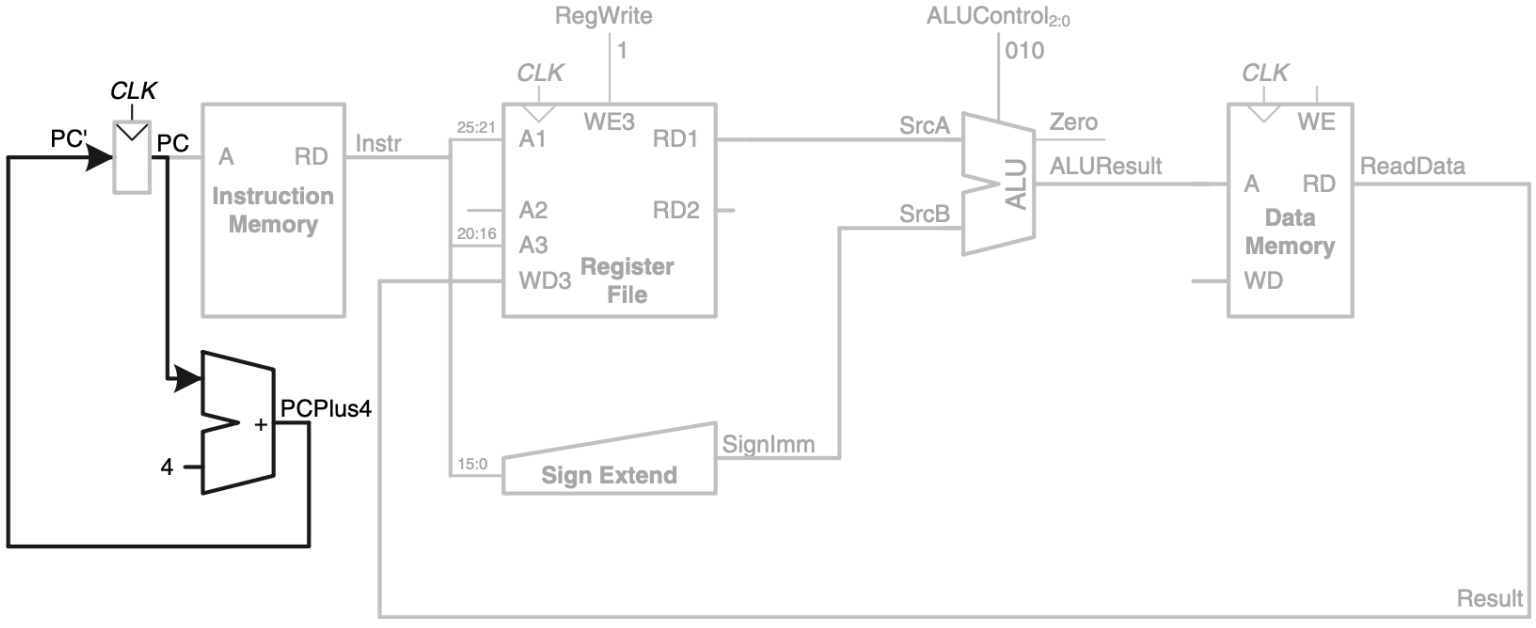


Figure 7.7 Determine address of next instruction for PC

- **Instruction** has been executed: to fetch next instruction we need to increment instruction pointer (here PC, in x86 %rip) by 4

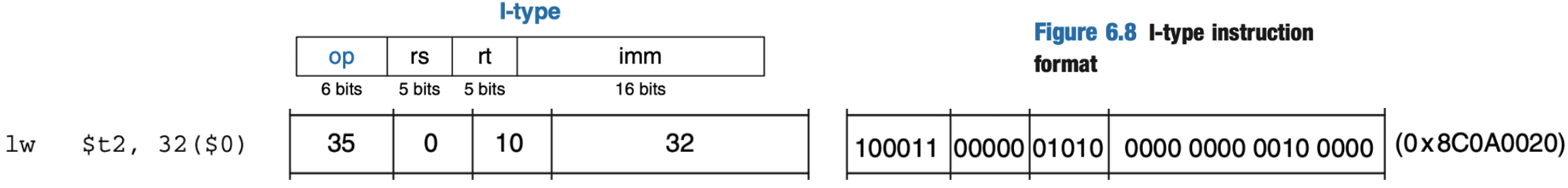


Figure 6.8 I-type instruction format

Single Cycle Processor

Systems@ETH Zürich

- **Lets** implement the MIPS instructions: **R-type, I-type** s.t. our processor can execute them

- I-type
  - Lw (load woard)
  - Sw (store woard)
- R-type (binary ops: add, sub, …)
- Beq instruction (beq, bne, ..)

Single Cycle Processor

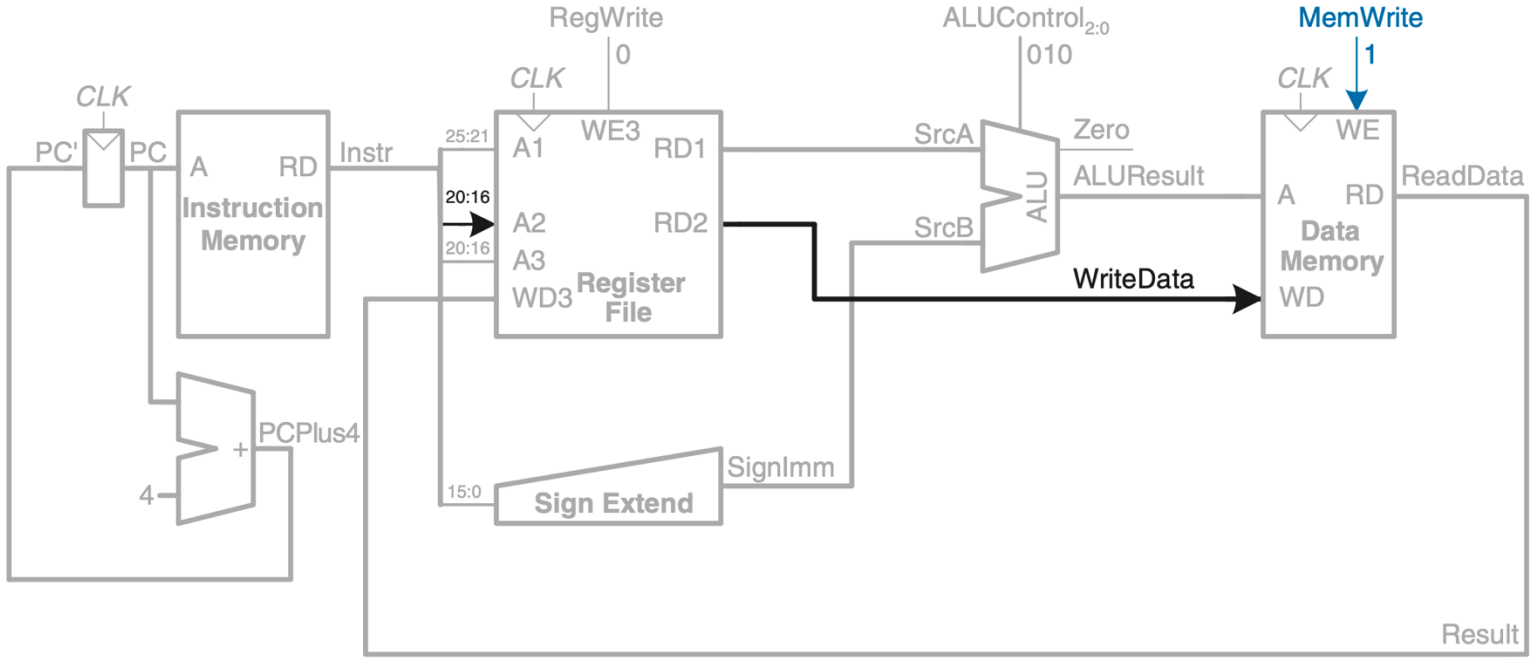


Figure 7.8 **Write data to memory for `SW` instruction**

- Processor can already calculate address (SEXT, add register to immediate)
- **Sw** reads second register from register file (A2); add path to **write it to memory**

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Figure 6.8 **I-type instruction format**

`sw $s1, 4($t1)`	43	9	17	4	101011 | 01001 | 10001 | 0000 0000 0000 0100 (0xAD310004)

Single Cycle Processor

Systems@ETH zürich

- **Lets** implement the MIPS instructions: **R-type, I-type** s.t. our processor can execute them

- I-type
  - Lw (load woard)
  - Sw (store woard)
- R-type (binary ops: add, sub, …)
- Beq instruction (beq, bne, ..)

Single Cycle Processor

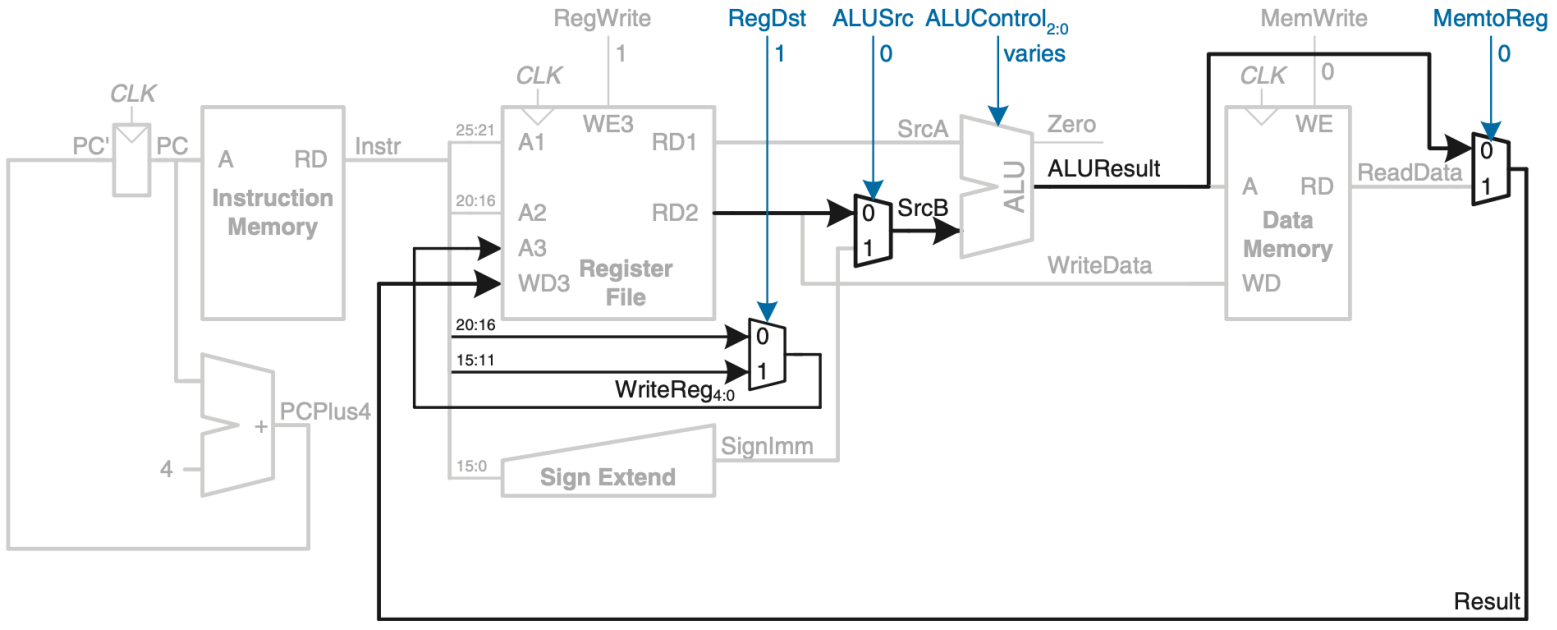


Figure 7.9 **Datapath enhancements for R-type instruction**

- **Register File**: reads two registers, ALU performs operation on these 2 registers
- **Need MUX** to choose between **SEXT and 2nd Reg (RD2)**
- **Need MUX** to choose whether to write to memory (sw,lw) or to **register**
- **Register to write to** specified by "rd" 15:11 (for sw it was specified by "rt" 20:16 => MUX)

R-type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

	op	rs	rt	rd	shamt	funct		op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32		000000	10001	10010	10000	00000	100000	(0x02328020)

Single Cycle Processor

Systems@ETH zürich

- **Lets** implement the MIPS instructions: **R-type, I-type** s.t. our processor can execute them

- I-type
  - Lw (load woard)
  - Sw (store woard)
- R-type (binary ops: add, sub, …)
- Beq instruction (beq, bne, ..)

Single Cycle Processor

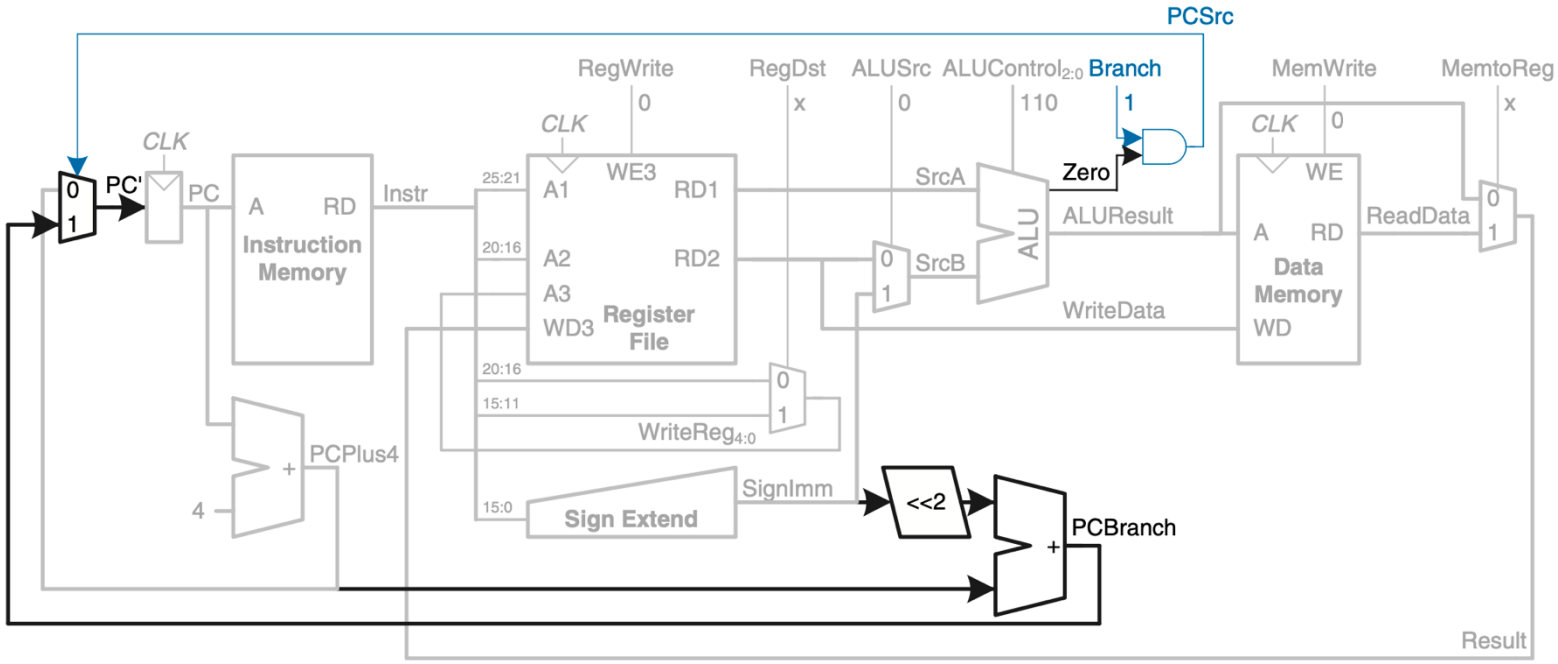


Figure 7.10 **Datapath enhancements for `beq` instruction**

- **Branch** instruction: compares two registers, if they are equal adds the branch offset to the PC
- Offset stored in **immediate field**: 15:0, gets **SEXT** and added to PC if chosen to branch

Single Cycle Processor

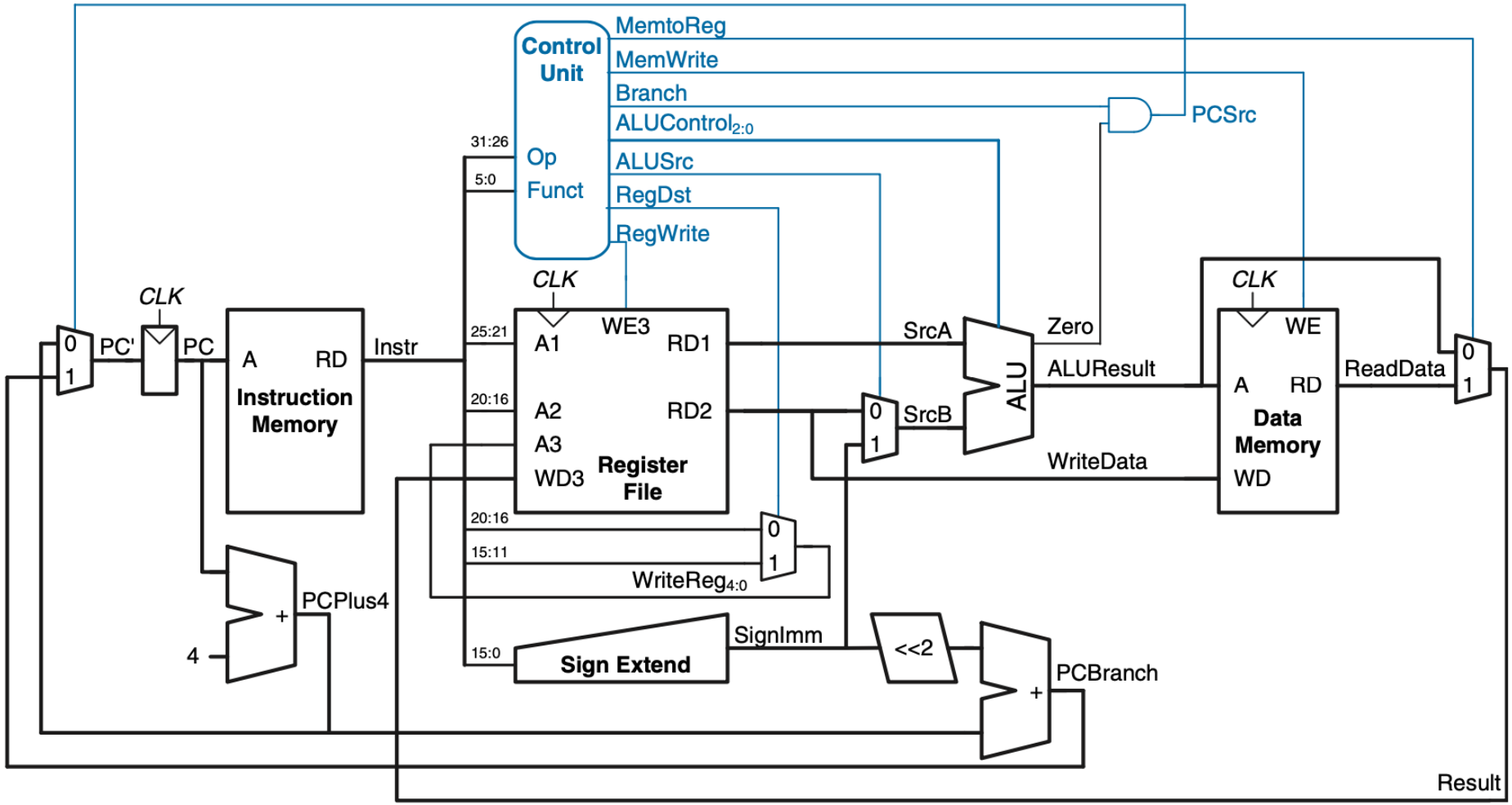


Figure 7.11 Complete single-cycle MIPS processor

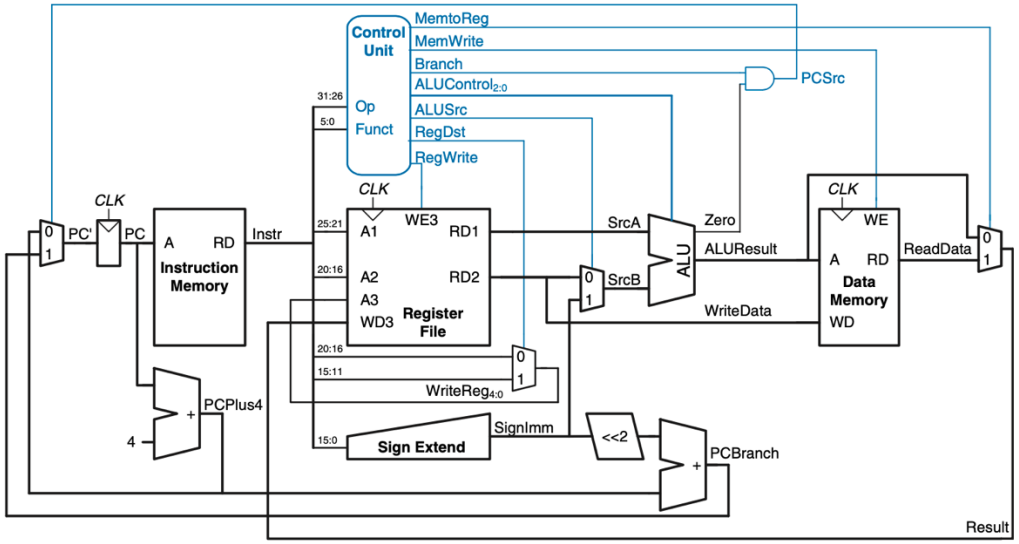


Figure 7.11 Complete single-cycle MIPS processor

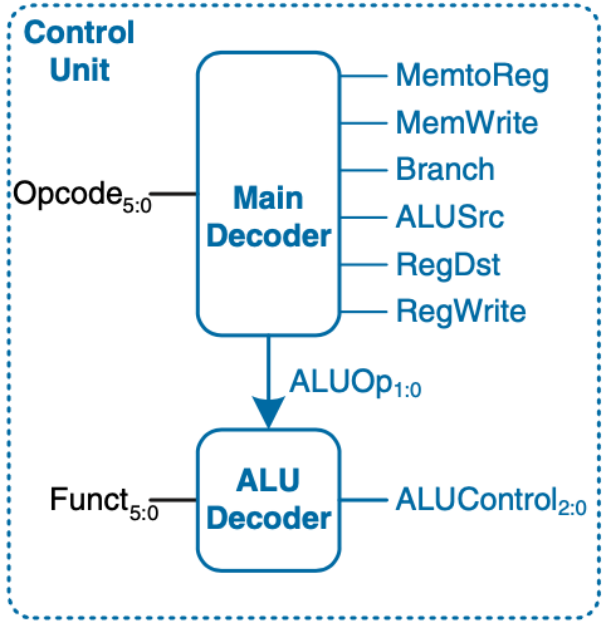


Figure 7.12 Control unit internal structure

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

- **Recall**: different instructions have different opcodes
- **Opcodes** decide which signals to give
  - Lw/sw: **MemToReg**=1; Add/sub: **MemToReg**=0 etc.

Systems@ETH Zürich

Recap

Michroarchitecture: Pipelined Processor

Type of Processors

Systems@ETH zürich

- **Single Cycle**: executes an entire instruction in one cycle
- Easy to explain, simple control
- Cycle time is limited by the slowest instruction

- **Pipelined Microarchitecture**: applied pipelining to the single cycle michroarchitecture
- Can execute multiple instruction simultaneously, must handle **dependencies** between those instructions

- **All commercial high-performance processors use pipelining today**

Pipelined processor

Systems@ETH zürich

- **Idea**: subdivide single-cycle processor into 5 pipeline stages: then we can execute instruction simultaneously, one in each stage
  - **Since** each stage has 1/5 of the **entire logic**: clock frequency is **almost five times faster**, ideally: latency unchanged, but throughput 5x higher

- **Call stages**: 1. Fetch, 2. Decode, 3. Execute, 4. Memory, 5. Writeback
- 1. **Fetch**: processor reads instruction from instruction memory
- 2. **Decode**: Processor reads source operands from register file and decodes instruction to procdue control signals
- 3. **Execute**: ALU computation
- 4. **Memory**: Read/Writes to memory
- 5. **Writeback**: Processor writes results to register file

Pipelined processor

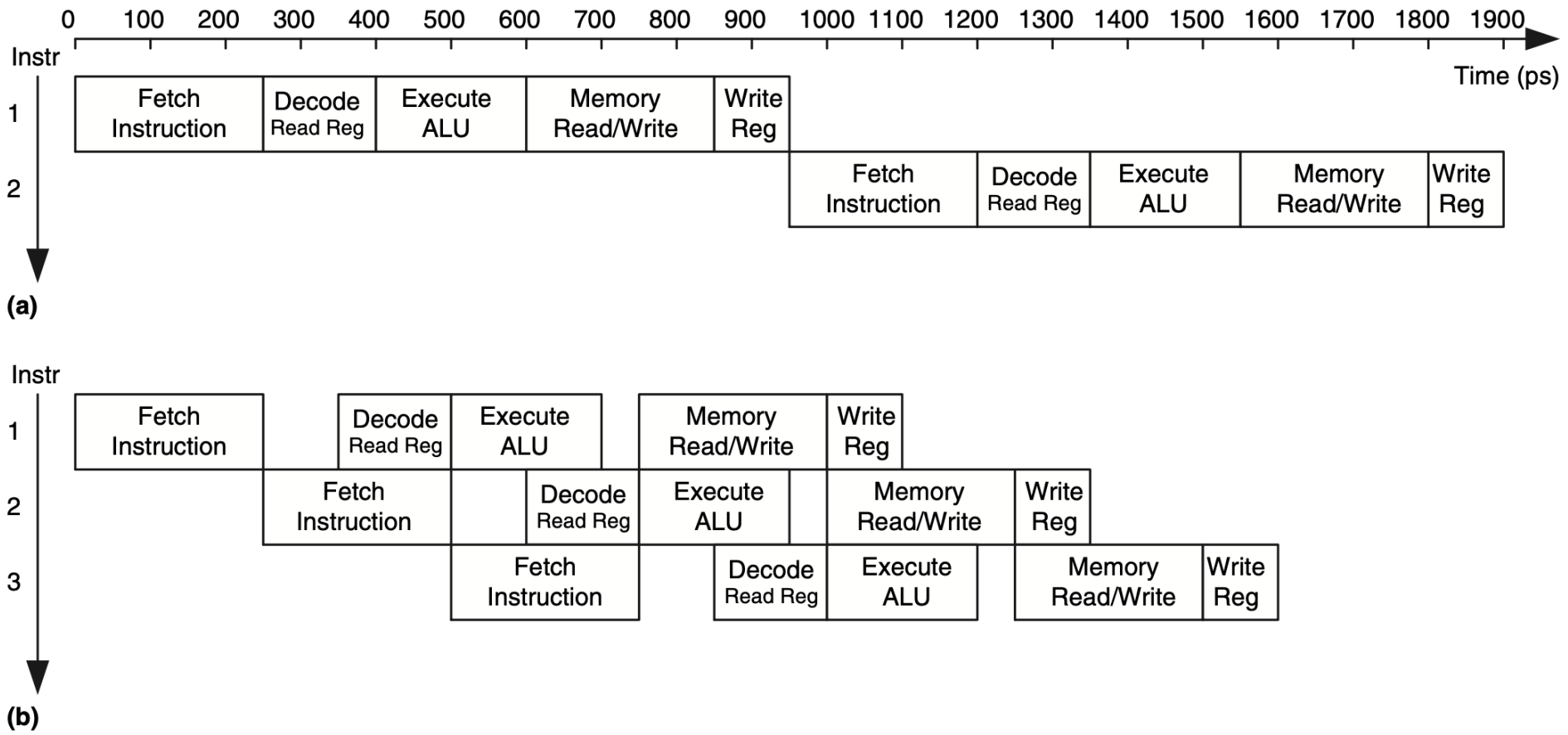


Figure 7.43 Timing diagrams: (a) single-cycle processor, (b) pipelined processor

Pipelined processor

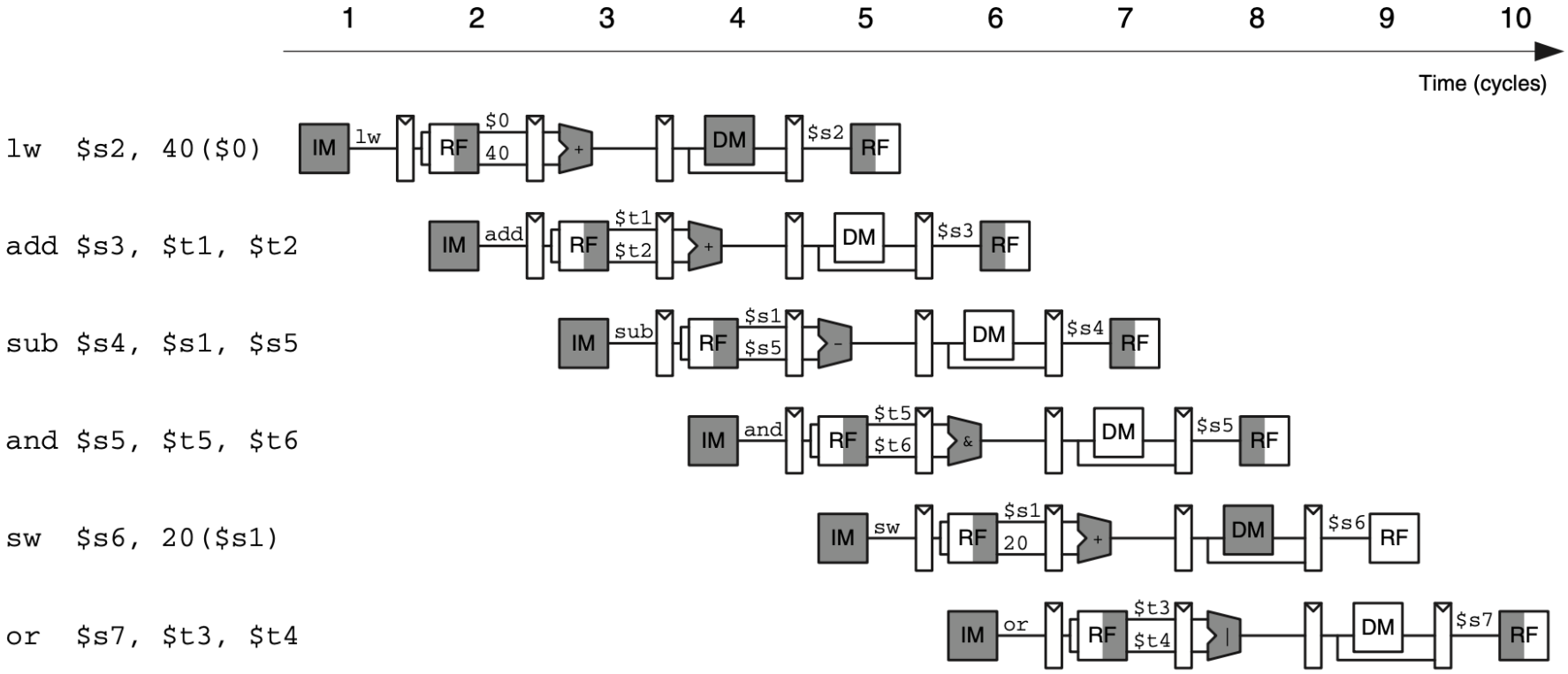


Figure 7.44 Abstract view of pipeline in operation

- **Example of what happens in Cycle 6**: OR is being fetched from IM; $s1 is being read from the register file in SW instruction; ALU computes $t5 AND $t6 in AND instruction etc.

Pipelined processor: chop single processor in 5 stages

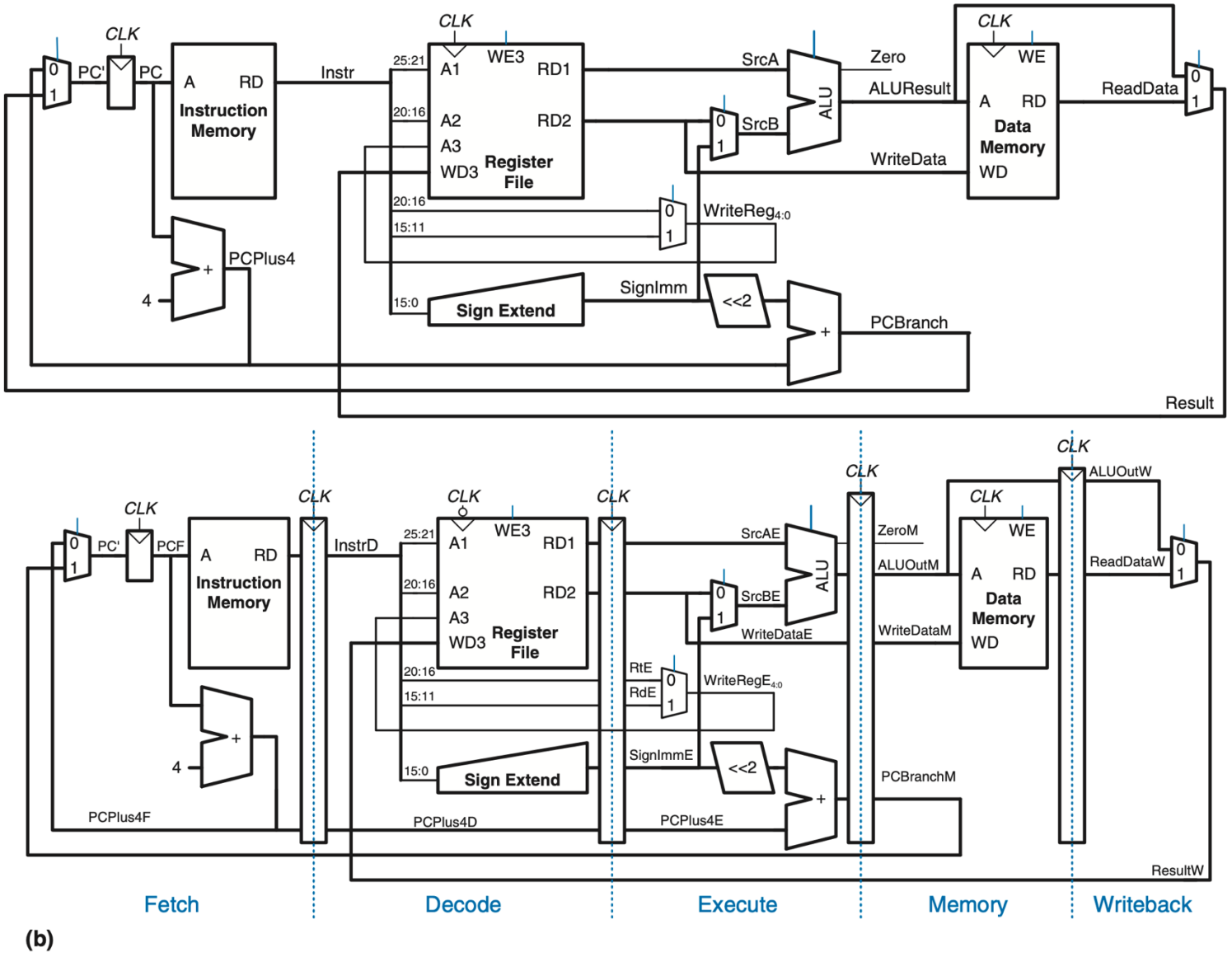


Figure 7.45 Single-cycle and pipelined datapaths

- **Bank of flip flops** hold the intermediate values

Pipelined processor

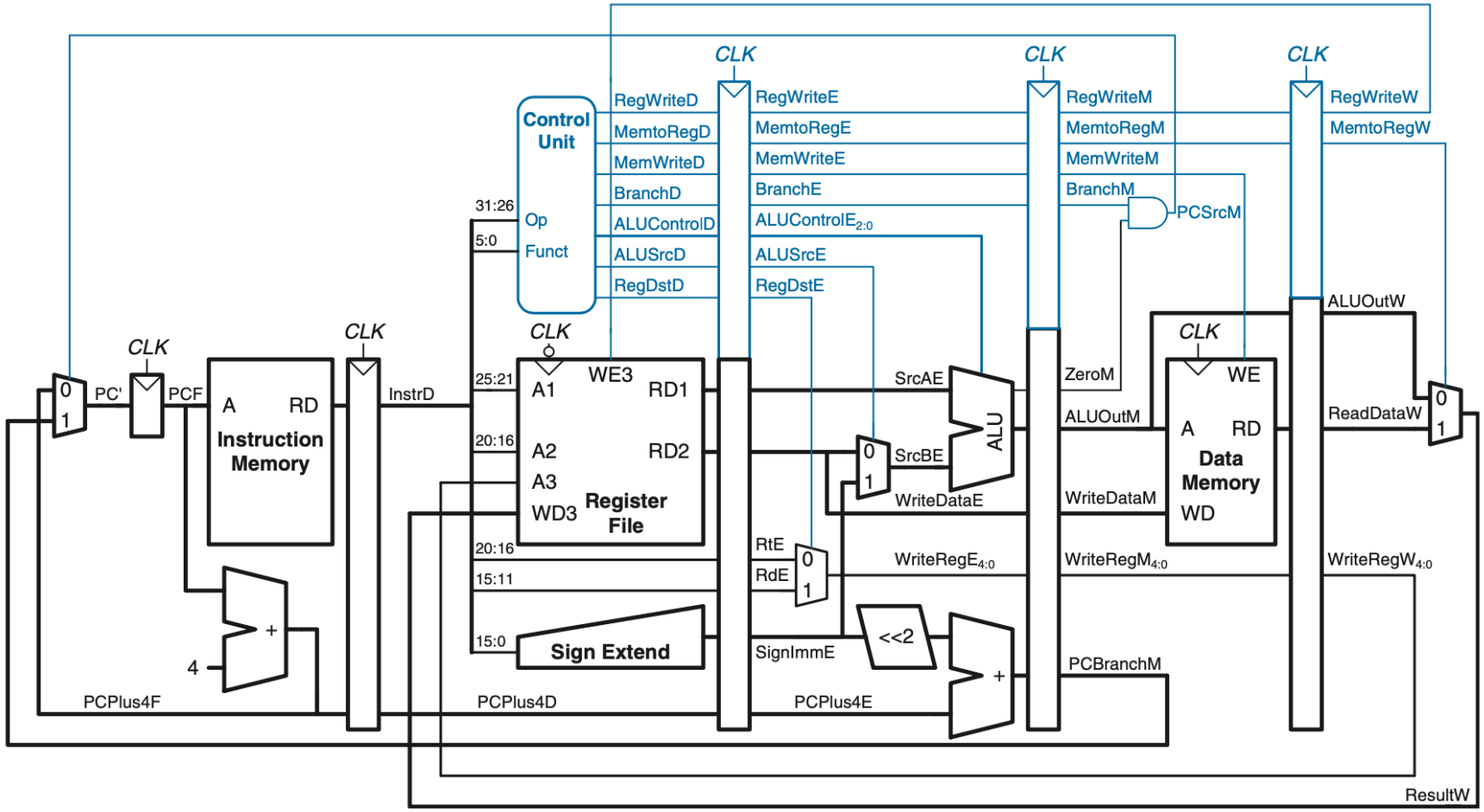


Figure 7.47 Pipelined processor with control

Pipelined processor

Systems@ETH zürich

- **Pipelined** processors have higher throughput: but there are some things to be **cautious about**: **data hazards**
- When multiple instructions are handled concurrently: if one instruction is dependent on the result of another which has not yet completed, we have a **hazard**

Pipelined processor

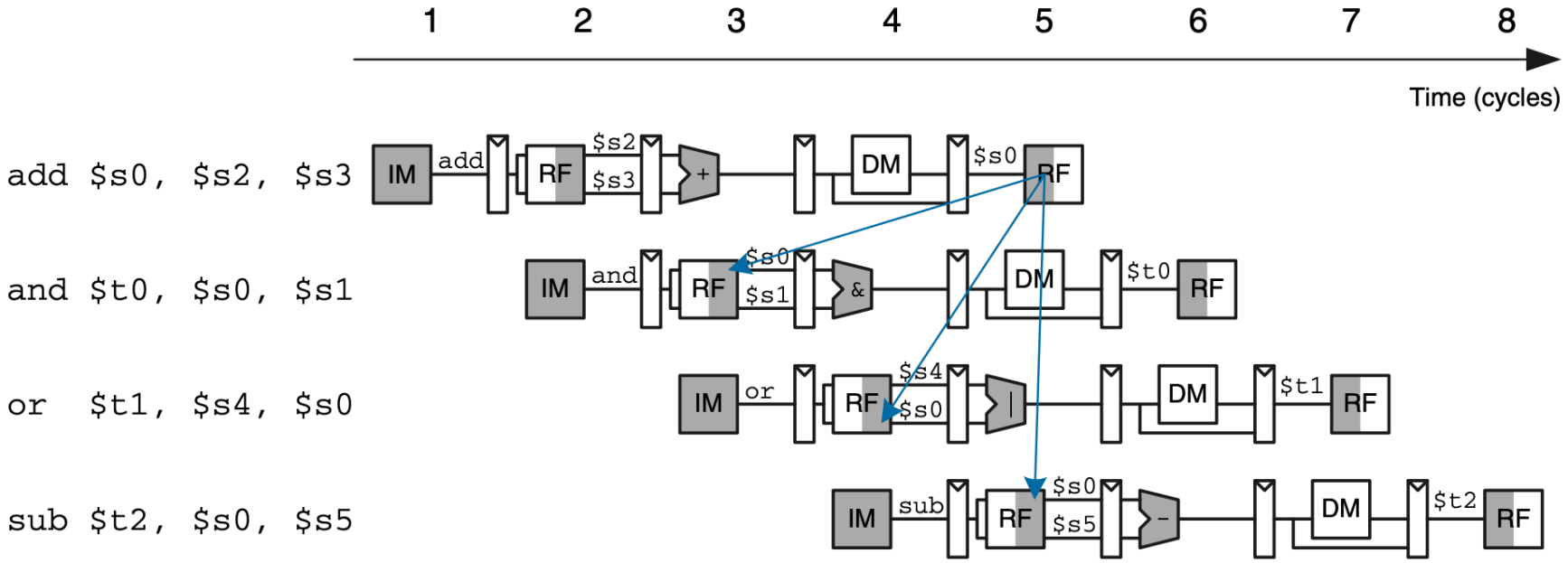


Figure 7.48 **Abstract pipeline diagram illustrating hazards**

- **RAW hazard**: add instructions writes result into $s0 at first half of cycle3, but **and** instruction reads $s0 on cycle3 obtaining the **wrong value**

Pipelined processor

Systems@ETH zürich

- **There are 2 solutions for data hazards**
- **1. Forwarding**: result from memory or writeback stage is “forwarded” to a dependent instruction in the execute stage
- **2. Stalling**: stall the pipline, i.e. hold up operation until the data is available
  - **Sometimes** we need to combine forwarding with stalling, if even with forwarding we don’t have enough time

Pipelined processor: 1. Forwarding

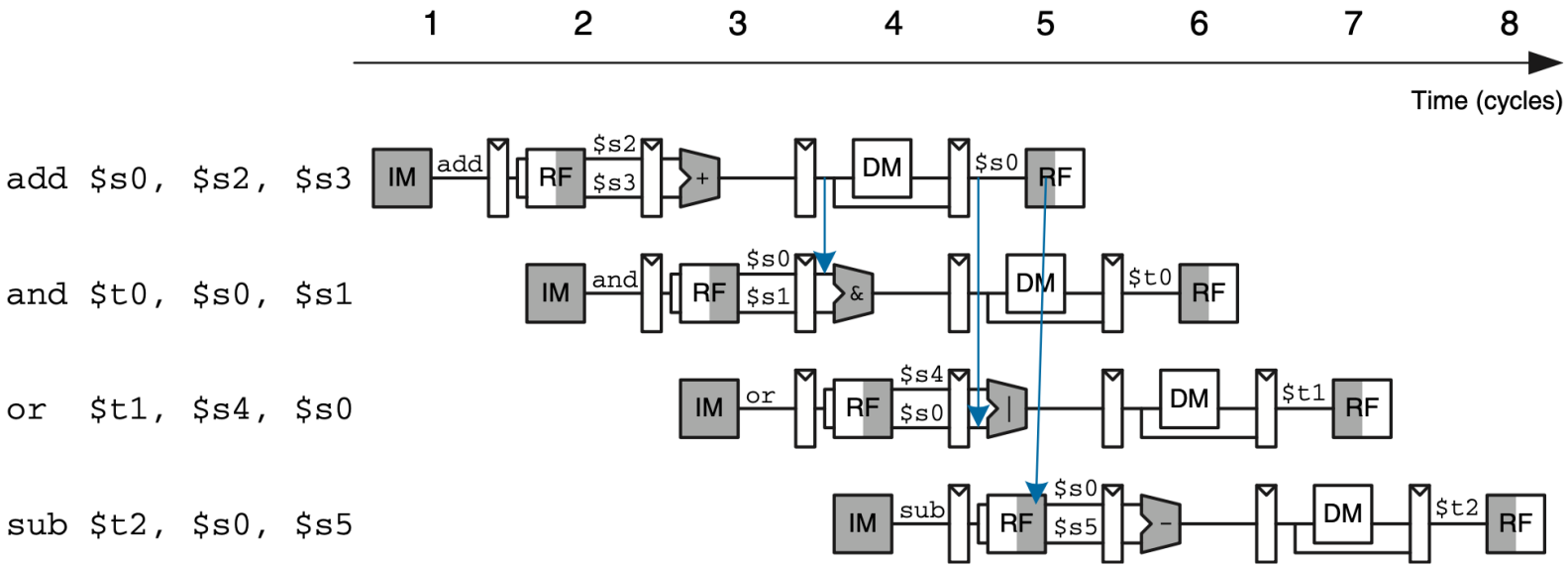


Figure 7.49 **Abstract pipeline diagram illustrating forwarding**

- **In cycle4**, $s0 is forwarded from the memory stage of the **add** instruction to the **Execute stage** of the dependent **AND** instruction

Pipelined processor

Systems@ETH zürich

- **There are 2 solutions for data hazards**
- **1. Forwarding**: result from memory or writeback stage is “forwarded” to a dependent instruction in the execute stage
- **2. Stalling**: stall the pipline, i.e. hold up operation until the data is available
  - **Sometimes** we need to combine forwarding with stalling, if even with forwarding we don’t have enough time

Pipelined processor: 2. Stalling w/ forwading

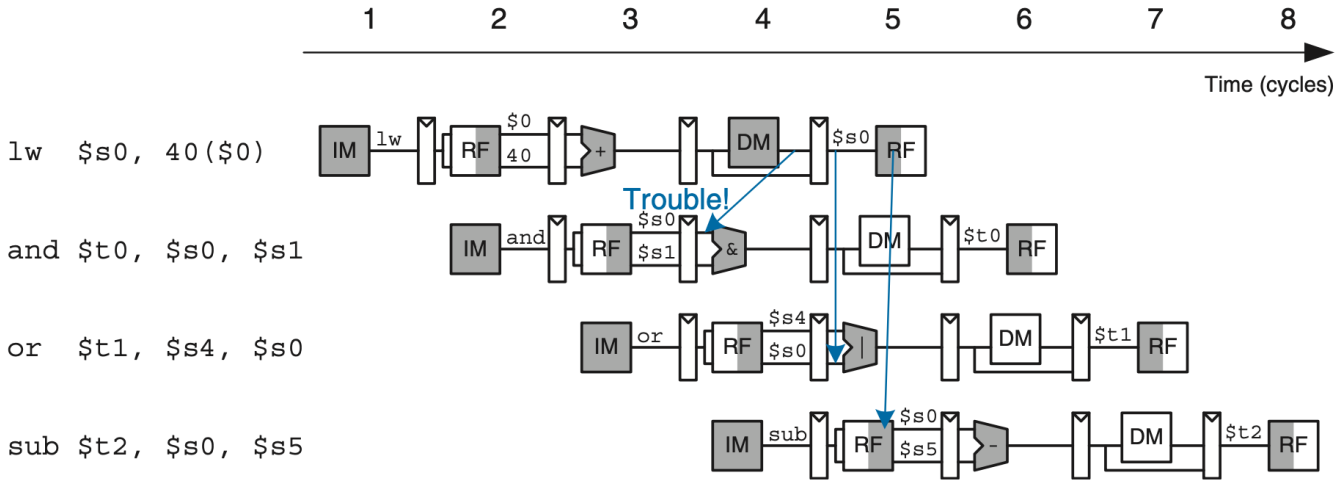


Figure 7.51 Abstract pipeline diagram illustrating trouble forwarding from `lw`

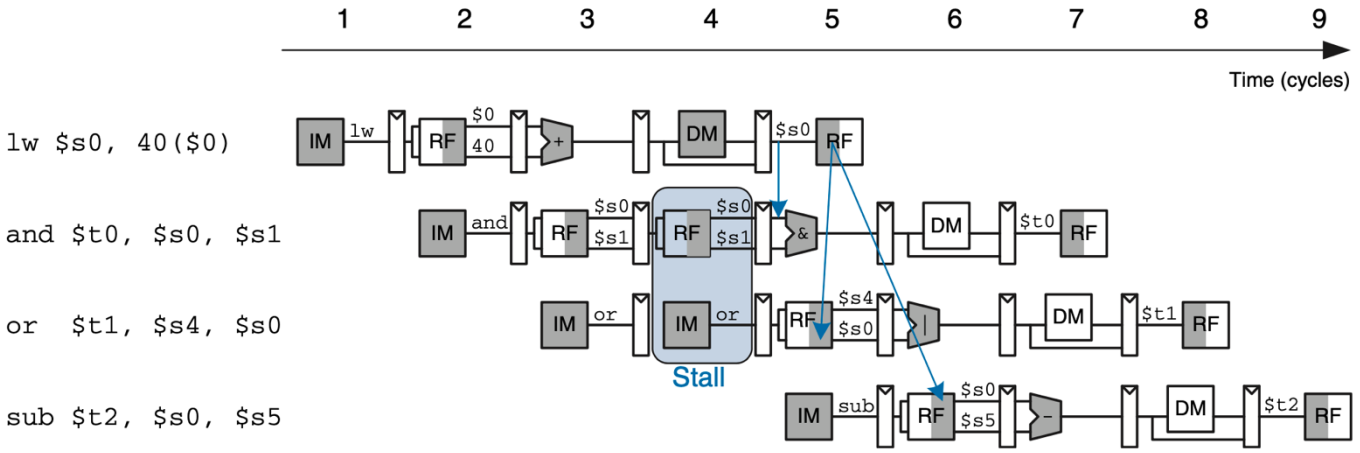


Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

- **lw** instruction received data from memory at end of cycl4, **but** AND instruction needs that data as a source operand at the **beginning of cycle4**

- **Solution**: stall pipeline and then forward

Pipelined processor

- **A piplined processor** with stalling, forwading etc. and all control units implemented then looks like the following

Pipelined processor

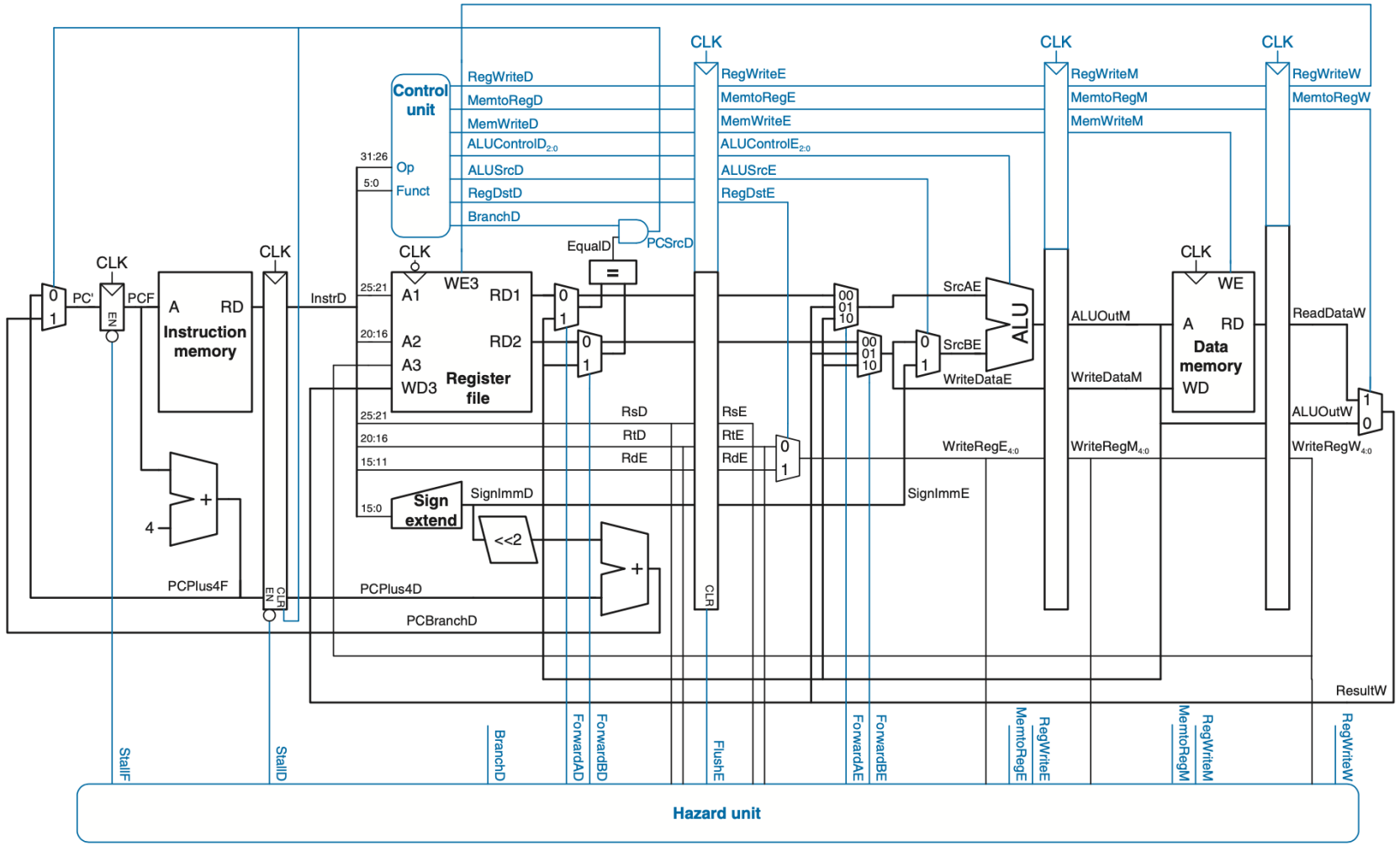


Figure 7.58 Pipelined processor with full hazard handling

Systems@ETH Zürich

Recap

Advanced Processor Architecture Paradigms

Superscalar Processors, Out of Order Execution, Multithreading, Fine Grained Multithreading, SIMD and VLIW

Superscalar Processor and OoO Execution

Systems@ETH Zürich

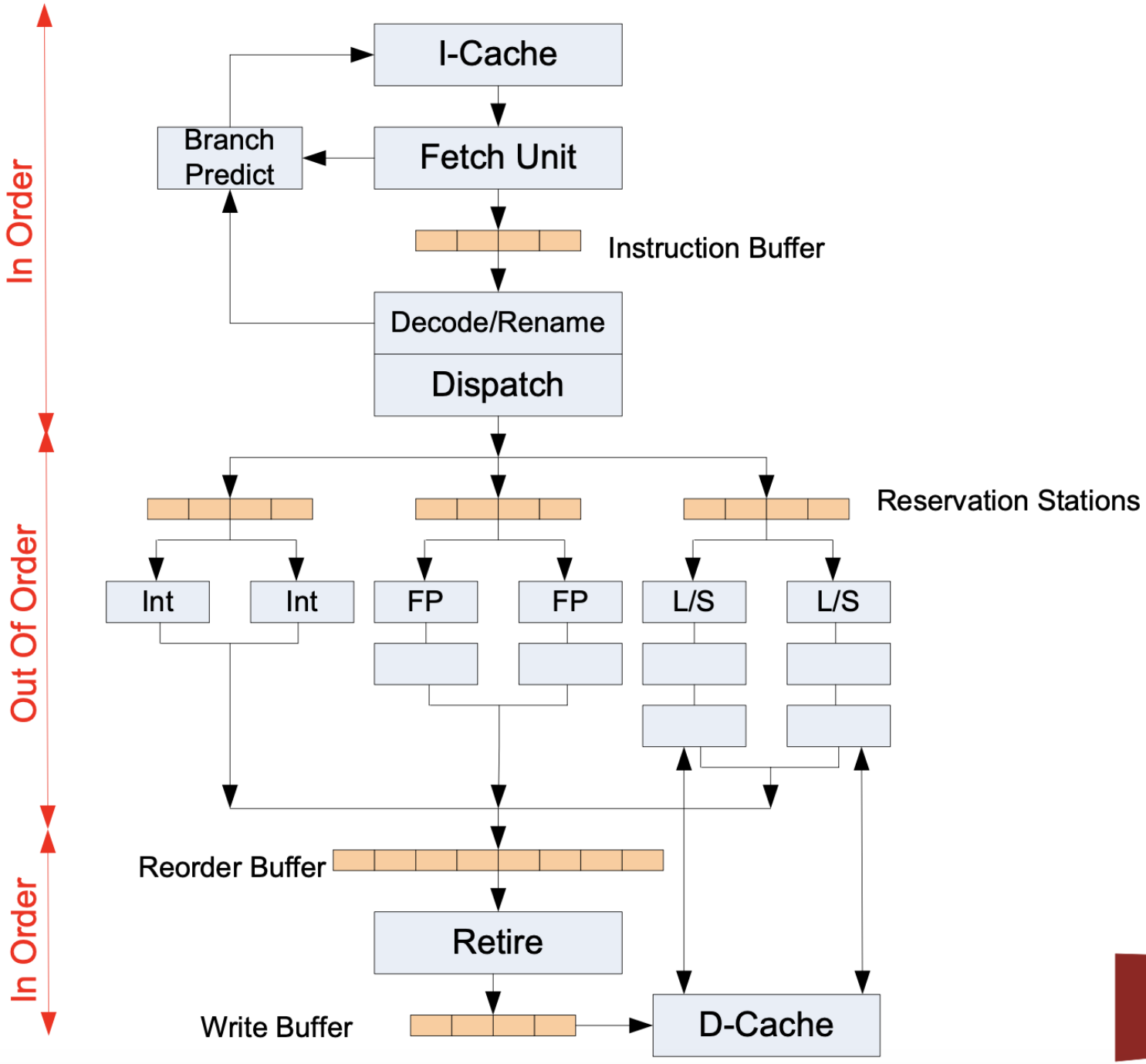
- **Superscalar architecture** refers to a processor that can issue and execute multiple instructions **simultaneously** using multiple exueciton units (multiple ALUs, FPUs tec.) to allow for **parallelism at the instruction level**
- **Note**: All processors today are pipelined and superscalar

Superscalar Processor and OoO Execution

- **Out of Order (OoO) Execution**: Instructions are being fetched and decoded in program order, but we can execute them out of order
- **Write in Order**: To allow for correct exceptions etc.

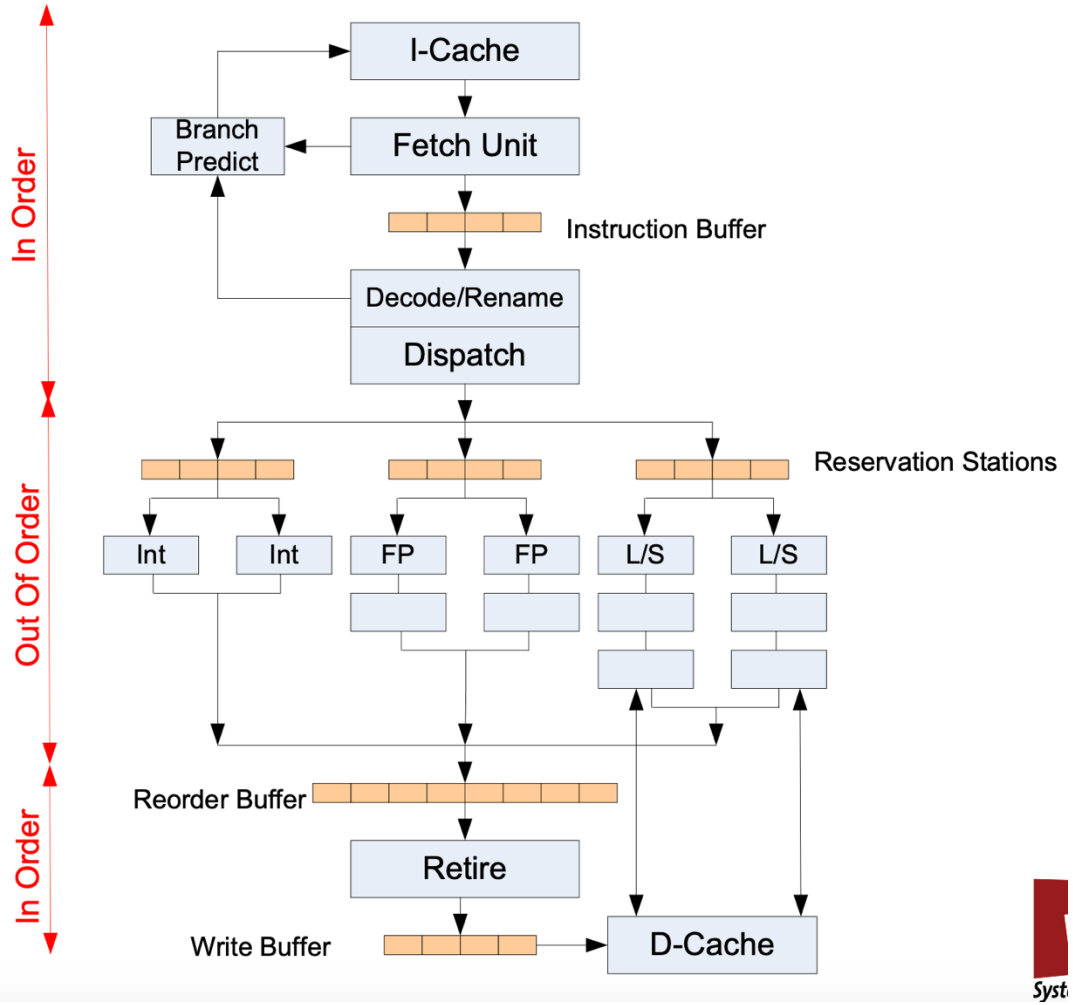
- **Example**: If ADD stalls, we can still execute MUL and SUB as they are indepdentn

```
1. ADD R1, R2, R3  ; Needs result of an earlier instruction
2. MUL R4, R5, R6  ; Independent
3. SUB R7, R8, R9  ; Independent
```

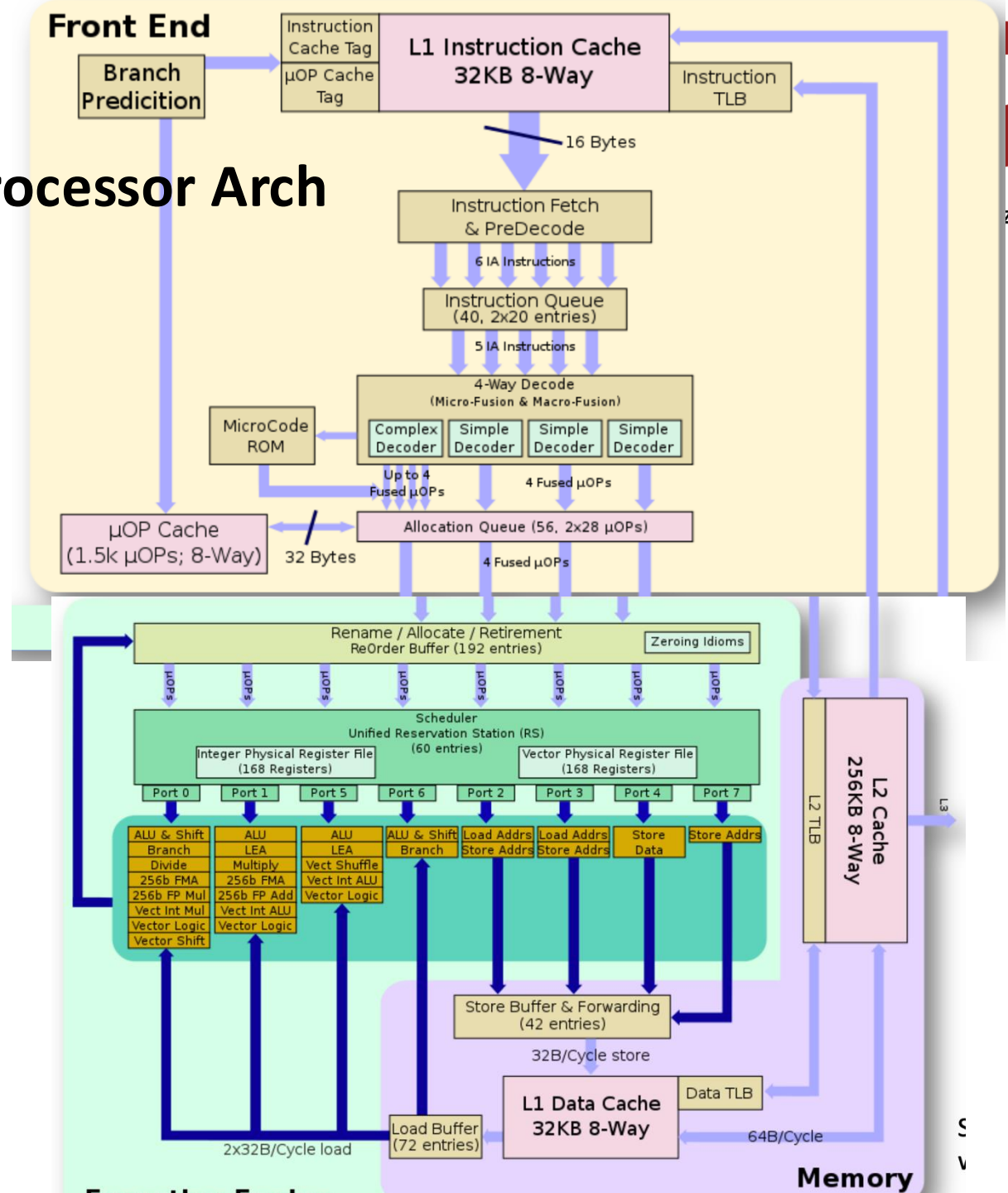


A Real Processor

- **Abstract Processor**



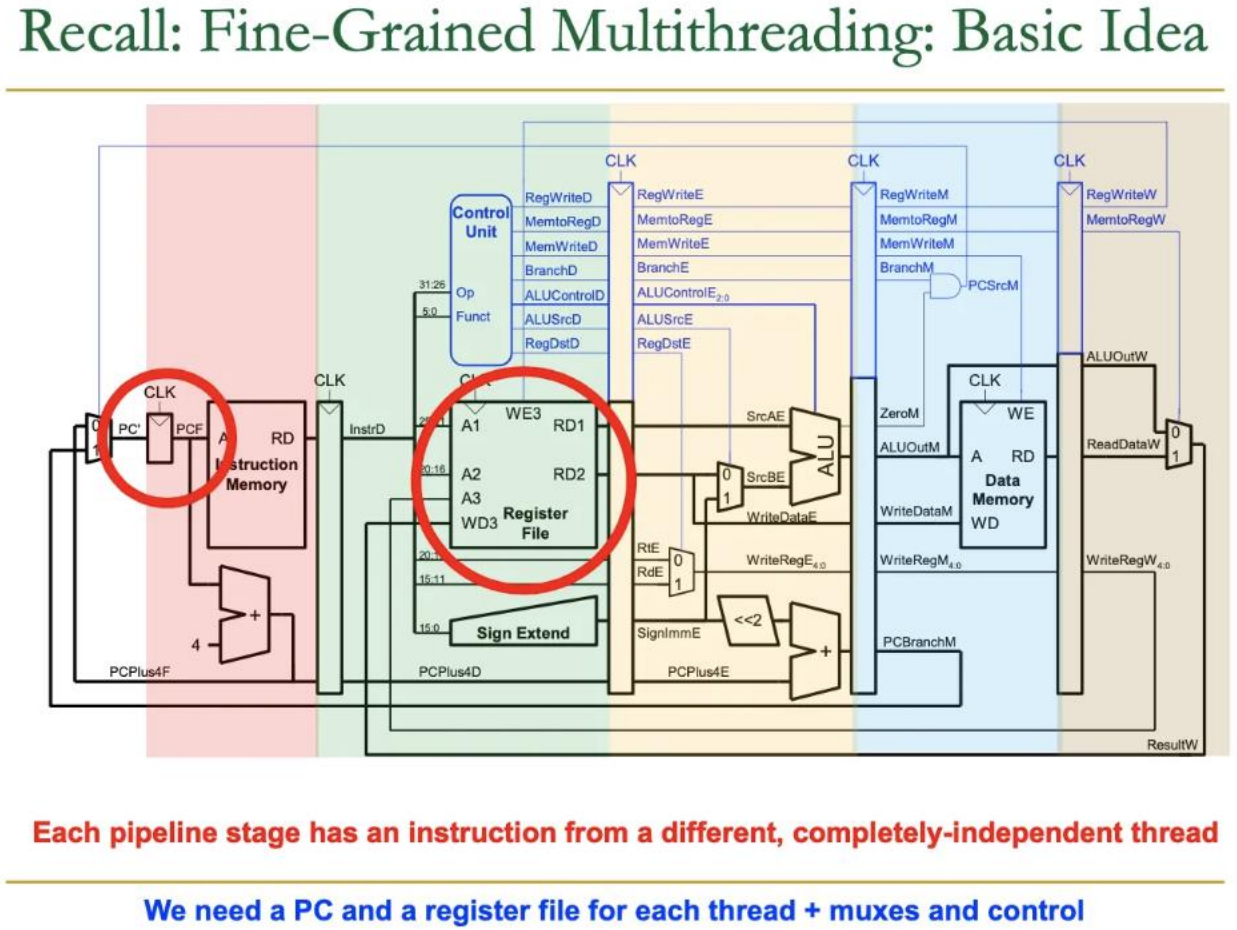
- **Real Processor Arch**



Multithreading

- **Normal multithreading**: If one process (or thread) does a memory instruction which takes time, start executing instructions of **another** process instead (to bridge the waiting time)

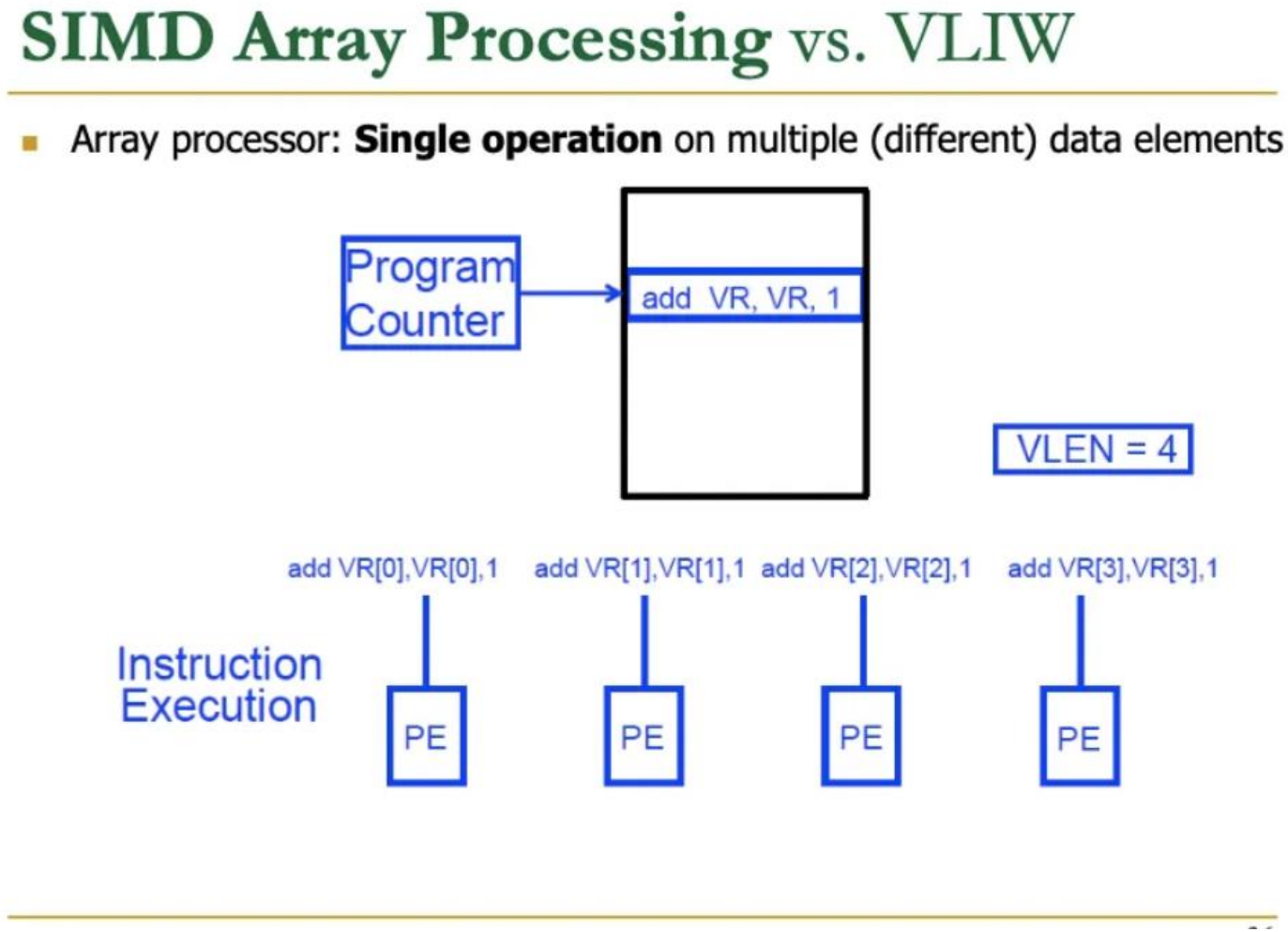
- **Fine Grained Multithreading (FGM)**: Every cycle we put a instruction from a different thread in the processor



SIMD and VLIW

- **SIMD** (Single Instruction Multiple Data): Single Instruction Multiple Data, i.e. one instruction but we operate on multiple data elements

- Called **array and vector processing**



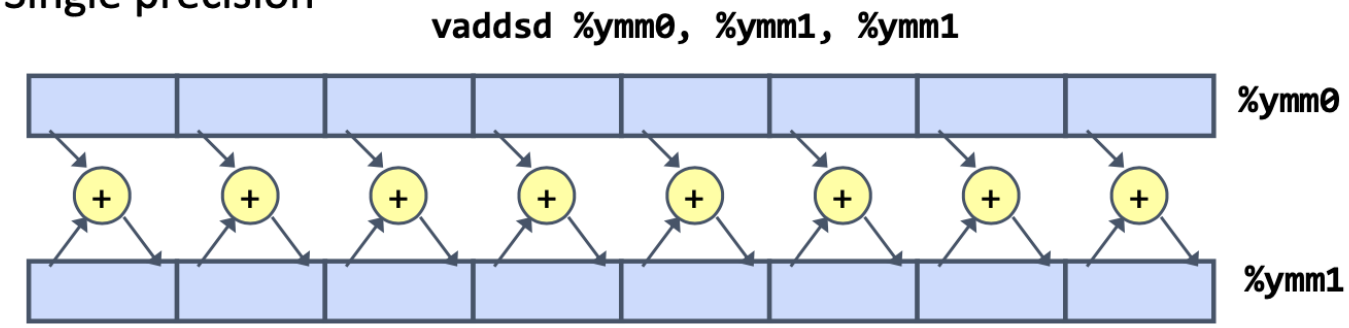
SIMD and VLIW

- **SIMD** Example: AVXV2, i.e. operation on vectors

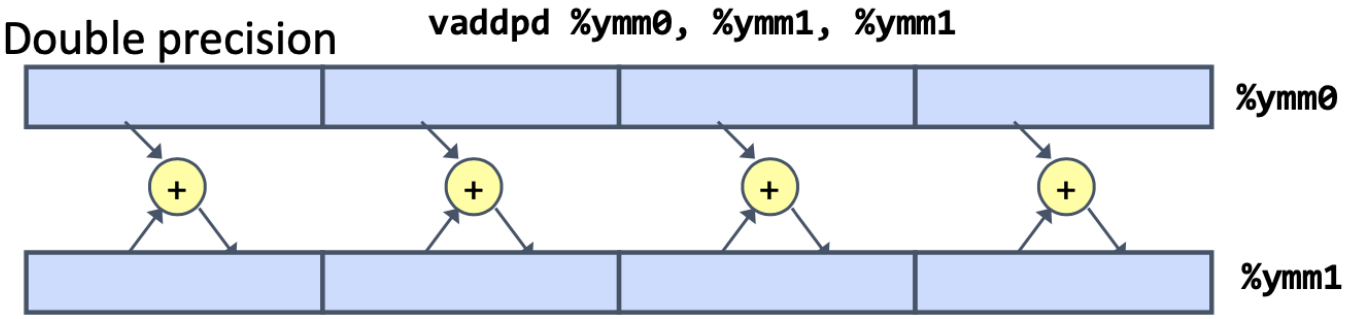
- **Single Instruction** (vaddsd) **Multiple Data** (8 ints; 4 doubles)

AVX2 SIMD operations (256-bit vectors)

- Single precision



- Double precision

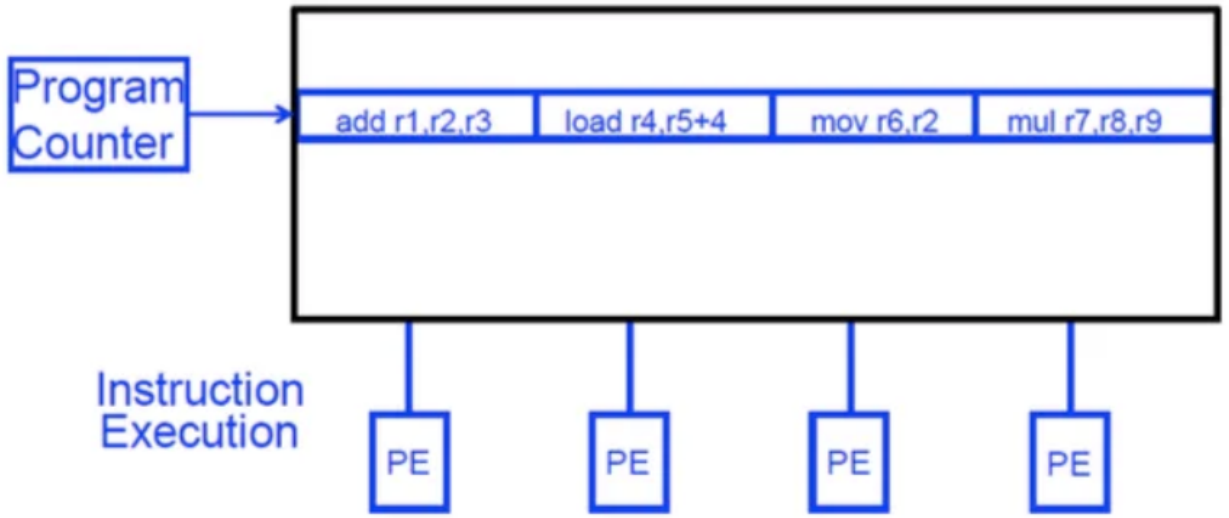


SIMD and VLIW

- **VLIW** (Very Long Instruction Word): Instead of processing just one instruction at the PC; take multiple (long instruction word)

SIMD Array Processing vs. VLIW

- VLIW: **Multiple** independent **operations** packed together into a "long inst."



Recap

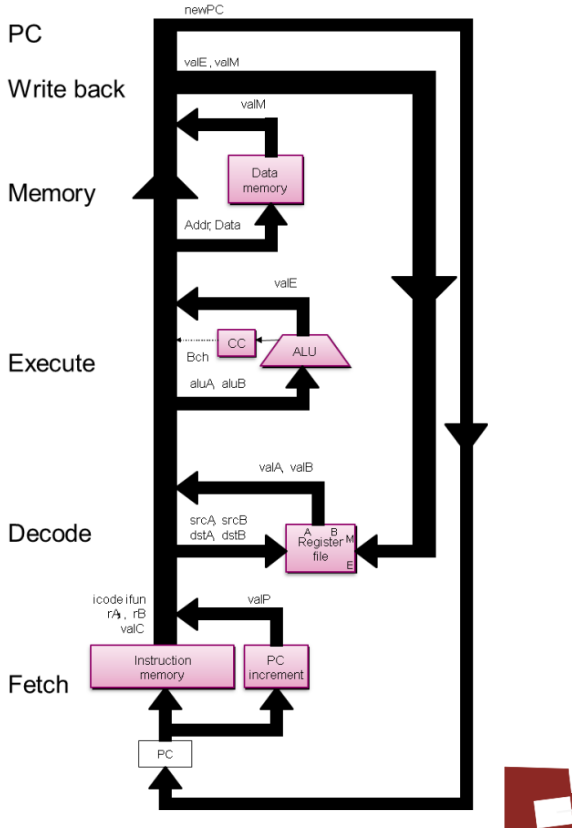
How does all this relate to what you have seen in the lecture?

What you saw in the lecture: seq proc.

Systems@ETH Zürich

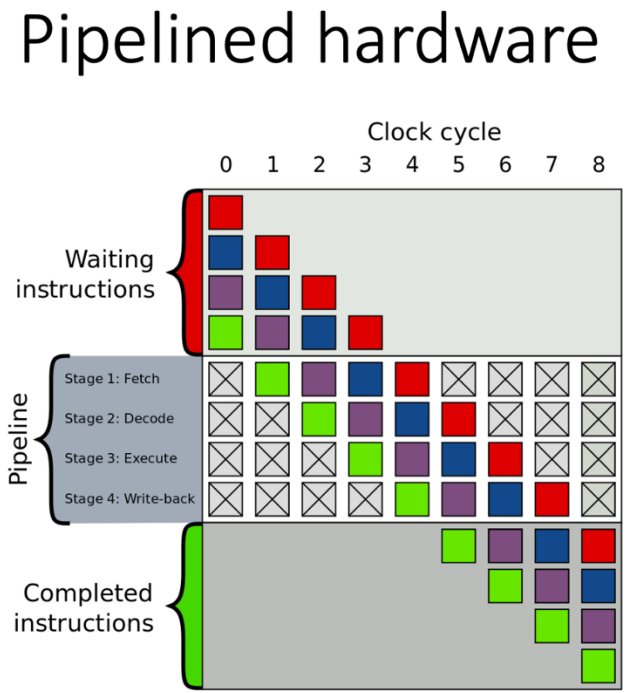
Sequential processor stages

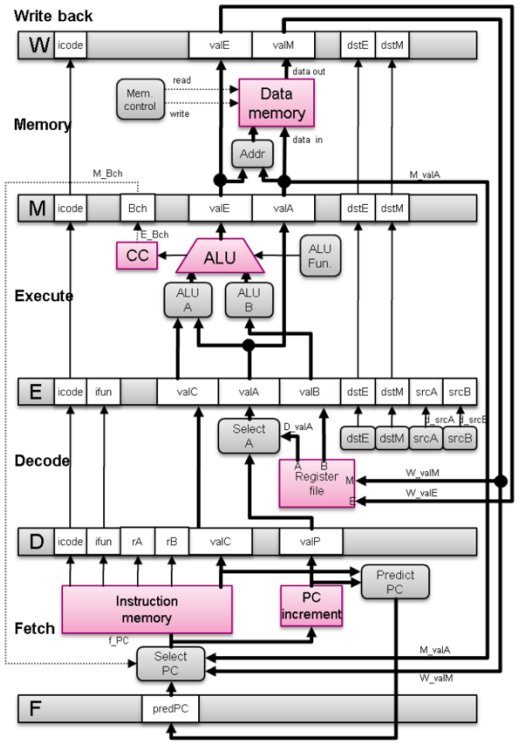
- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



- **You should now understand this processor design diagram**: the 5 stages for the sequential processor

What you saw in the lecture: pipelined proc.





- **You should also** understand what pipelining is, why its useful, and how this works inside the processor

What you saw in the lecture: data hazards

Systems@ETH Zürich

Recall: Data hazards

- Data dependencies for instruction *j* following instruction *i*
  - Read after Write (RAW) (*true dependence*)
    - Instruction *j* tries to read before instruction *i* tries to write it
  - Write after Write (WAW) (*output dependence*)
    - Instruction *j* tries to write an operand before *i* writes its value
  - Write after Read (WAR) (*anti dependence*)
    - Instruction *j* tries to write a destination before it is read by *i*
  - No such thing as a Read after Read (RAR) hazard since there is never a problem reading twice

Can avoid hazard with register renaming

- **You should also** understand what pipelining is, why its useful, and how this works inside the processor; but also what kind of **hazards** appear and why

Systems@ETH Zürich

Recap

Possible Compiler and Code Optimisations

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

Code Optimisations

Optimizing compilers

- Use optimization flags,
  default can be no optimization (-O0)!
- Good choices for gcc:
  -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
  - icc is often faster than gcc

Code Optimisations

Systems@ETH Zürich

Optimizing compilers

- Compilers are good at: mapping program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Compilers are not good at: improving asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Compilers are not good at: overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

Code Optimisations

Code motion

- Reduce *frequency* with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop
- Sometimes also called *precomputation*

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

Code Optimisations

Systems@ETH Zürich

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

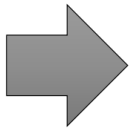
Code Optimisations

Strength reduction

- Replace costly operation with simpler one
  - Usually more specialized ("less strong")
  - Prior example: Shift/add instead of multiply or divide

16*x → x << 4

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    a[n*i + j] = b[j];
  }
}
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    a[ni + j] = b[j];
  }
  ni += n;
}
```

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

Code Optimisations

Share common subexpressions

- Reuse portions of expressions
- Compilers are not always good at exploiting arithmetic properties
  - GCC will do this with –O1

*3 mults: i*n, (i–1)*n, (i+1)*n*

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

*1 mult: i*n*

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```
imulq    %rcx, %rsi  # i*n
addq     %rdx, %rsi  # i*n+j
movq     %rsi, %rax  # i*n+j
subq     %rcx, %rax  # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

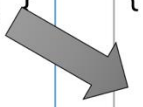
Code Optimisations

Why couldn't compiler move strlen out of inner loop?

- Procedure may have side effects
- Function might not return same value for given arguments
  - Could depend on other parts of global state
  - Procedure lower could interact with strlen

```
static int lencnt = 0;
```

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```



```
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

- Move call to strlen outside of loop
- Since result does not change from one iteration to another

Code Optimisations

- 1. Compiler Flags
- 2. Code Motion / Precomputation
- 3. Strength Reduction
- 4. Common Subexpressions
- 5. Procedure Calls
- 6. Memory Aliasing (Restrict)

Code Optimisations

Systems@ETH Zürich

Optimization Blocker: Memory Aliasing

- Two different memory references specify single location
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing

Code Optimisations

Possible aliasing

- Memory accessed
  ⇒ compiler assumes possible side effects

```
/* Sum rows of n x n matrix a and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

In memory, we have…

0x7fffbe90f860	A[0]	0	
0x7fffbe90f868	A[1]	1	
0x7fffbe90f870	A[2]	2	
0x7fffbe90f878	A[3]	**4**	B[0]
0x7fffbe90f880	A[4]	**8**	B[1]
0x7fffbe90f888	A[5]	**16**	B[2]
0x7fffbe90f890	A[6]	32	
0x7fffbe90f898	A[7]	64	
0x7fffbe90f8a0	A[8]	128	

Value of B:

```
init:  [4,  8,  16]
```


- **Issue here**: A, B might point to the same region, cant do scalar replacement (local accumulation)

Code Optimisations

How to remove aliasing

- Scalar replacement:
  - Copy array elements that are reused into temporary variables
  - Assumes no memory aliasing (otherwise possibly incorrect)

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
         double val = 0;
         for (j = 0; j < n; j++)
             val += a[i*n + j];
         b[i] = val;
    }
}
```

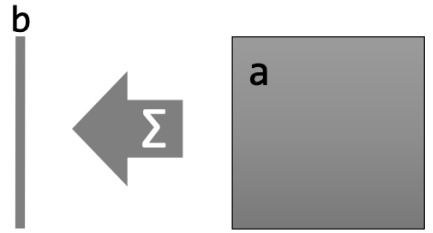
```
# sum_rows2 inner loop
.L66:
         addsd    (%rcx), %xmm0   # FP Add
         addq     $8, %rcx
         decq     %rax
         jne      .L66
```

- **Solution**: either do it explicitly yourself

Code Optimisations

`restrict` keyword: a hint to the compiler

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double restrict *a, double restrict *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
         b[i] = 0;
         for (j = 0; j < n; j++)
             b[i] += a[i*n + j];
    }
}
```



- C99 introduces `restrict` keyword for pointer declarations
- `restrict` tells the compiler that for the lifetime of this pointer, no other pointer will be used to access the "object" to which it points
- Tells the compiler not to worry about memory aliasing → enables more optimizations
- If the user violates this and passes a pointer that aliases, get undefined behavior

- **Solution**: or tell the compiler he can do it

Recap

Hardware Optimisations

Code Optimisations

- 1. Loop unrolling 2x1
- 2. Loop unrolling 2x1 with reassociation
- 3. Vector Instructions

Code Optimisations

- 1. Loop unrolling 2x1
- 2. Loop unrolling 2x1 with reassociation
- 3. Vector Instructions

Loop unrolling

Loop Unrolling (2x1)

- Perform 2x more useful work per iteration

```
void unroll2a_combine(struct vec *v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```


- **Load two values**, instead of just 1

Loop unrolling

Effect of Loop Unrolling

- Helps integer add
  - Achieves latency bound
- Others don't improve. Why?
  - Still sequential dependency

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	**1.00**	**3.00**	**3.00**	**5.00**

```
x = (x OP d[i]) OP d[i+1];
```

Can we do better?

- **Load two values**, instead of just 1

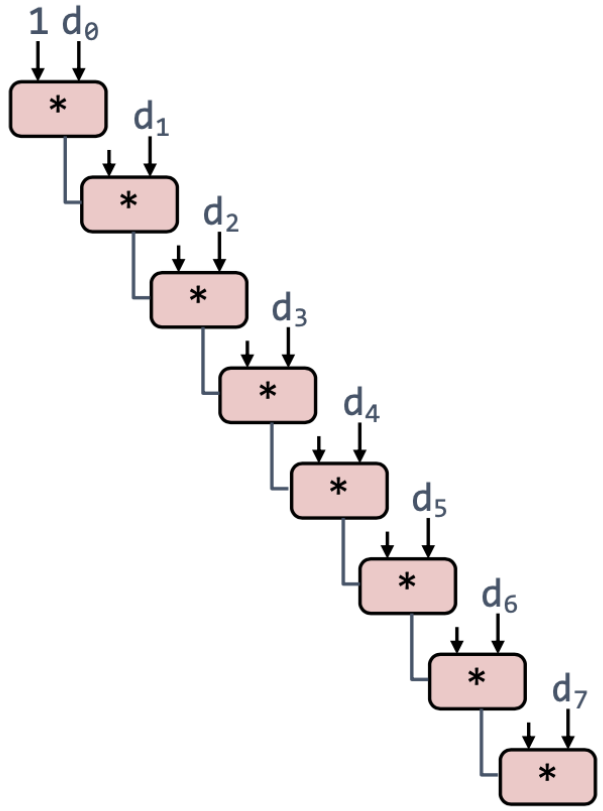
Loop unrolling

Combine4 = Serial Computation (OP = *)

- Computation (length=8)

```
(((((((1 * d[0]) * d[1]) * d[2]) * d[3])
          * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP



Code Optimisations

- 1. Loop unrolling 2x1
- 2. Loop unrolling 2x1 with reassociation
- 3. Vector Instructions

Loop unrolling with reassociation (2x1a)

Systems@ETH Zürich

- Can this change the result of the computation?
- Yes, for FP. Why?

Compare to before:

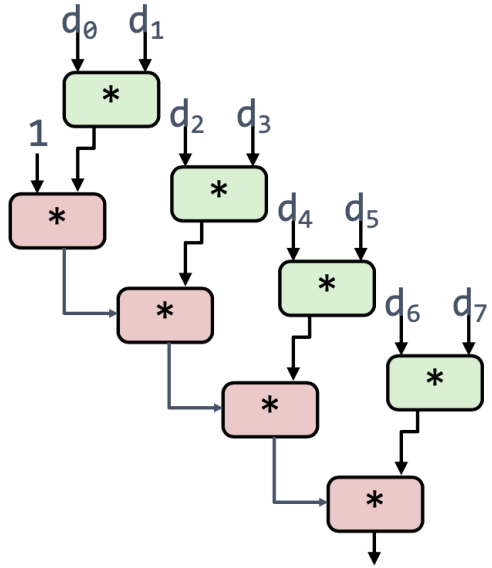
```
x = (x OP d[i]) OP d[i+1];
```

```
void unroll2aa_combine(struct vec *v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- **Why?** This breaks **sequential dependency** (operations of next iteration can directly be started after having calculated d[i] OP d[i+1]

Loop unrolling with reassociation (2x1a)

- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2




```
x = x OP (d[i] OP d[i+1]);
```

- **Why?** This breaks **sequential dependency** (operations of next iteration can directly be started after having calculated d[i] OP d[i+1]

Code Optimisations

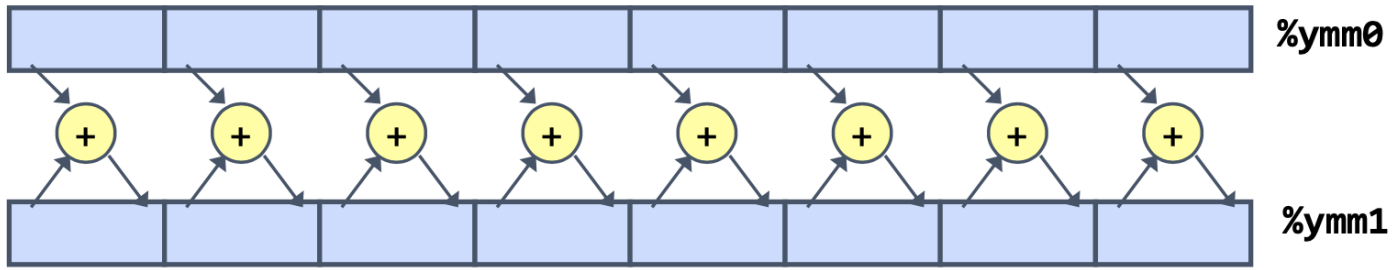
- 1. Loop unrolling 2x1
- 2. Loop unrolling 2x1 with reassociation
- 3. Vector Instructions

Vector Instructions

AVX2 SIMD operations (256-bit vectors)

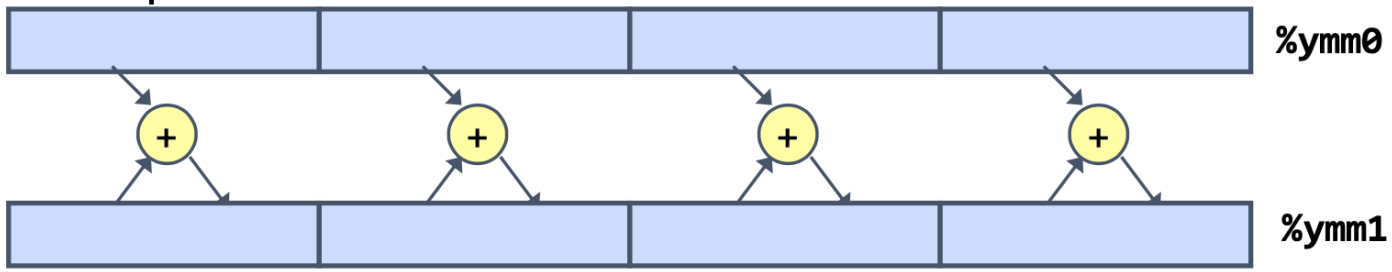
- Single precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



- Double precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



Introduction to Assignment09 and Exam Questions

HS23 CPE Calculations

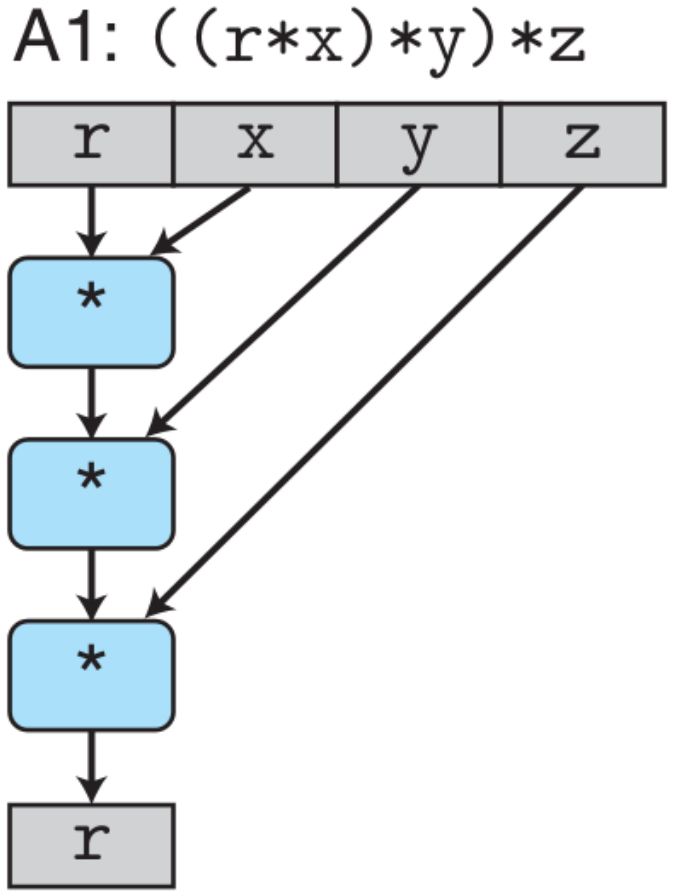
Assignment 09 Question 1

```
double aprod(double a[], int n)
{
  int i;
  double x, y, z;
  double r = 1;
  for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = r * x * y * z;
  }
  for (; i < n; i++)
    r *= a[i];
  return r;
}
```

- 3-way loop unrolling
  - 3 Execution per loop
- Find critical path
- Divide by 3

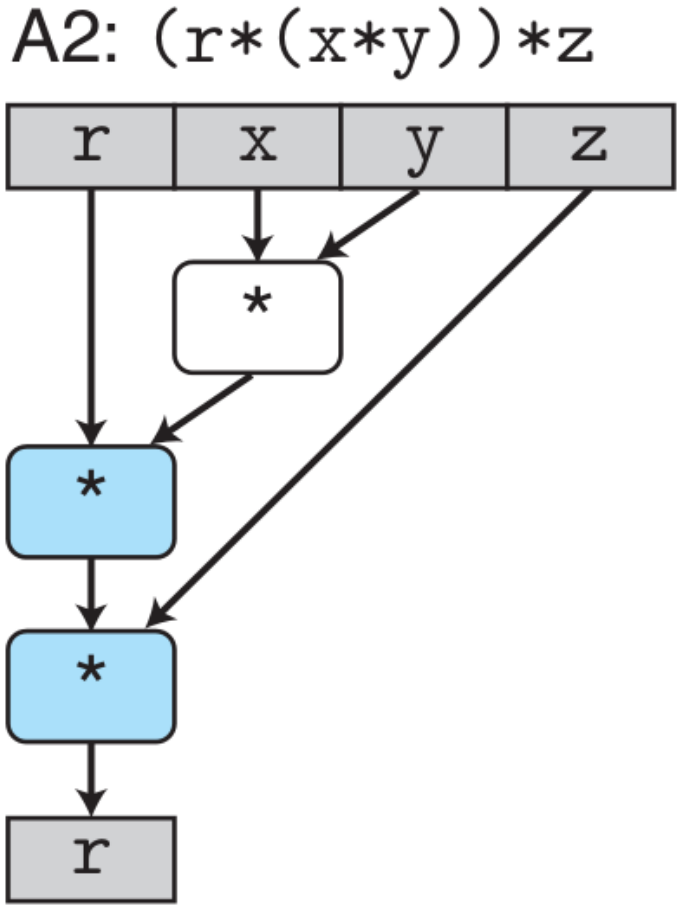
Assignment 09 Question 1

```
for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = ((r * x) * y) * z;
}
```



Assignment 09 Question 1

```
for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = (r * (x * y)) * z;
}
```



Exam Question HS2

Consider the following code:

```
int product(int* a, int n){
   int i, x, y, z;
   int p = 1;
   for (i = 0; i < n-2; i+=3) {
      x = a[i];
      y = a[i+1];
      z = a[i+2];
      p = p * x * y * z; // PRODUCT CALCULATION
   }
   for (; i < n; i++) {
      p = p * a[i];
   }
   return p;
}
```

For the line marked "PRODUCT COMPUTATION" consider 4 different re-association options:
Version a: p = ((p * x) * y) * z
Version b: p = (p * (x * y)) * z
Version c: p = p * ((x * y) * z)
Version d: p = (p * x) * (y * z)

This code executes on an Intel processor. Assume the processor has a **single integer multiplication unit** which is pipelined to support issuing **up to 1 integer multiplication per cycle** (i.e., after issuing an integer multiplication, the next integer multiplication can be issued in the next cycle). Assume the **latency of an integer multiplication is 3 cycles** for this processor.

Calculate the **theoretical cycles per element** for each PRODUCT COMPUTATION option. Recall that the cycles per element (CPE) is a measure of performance where the cycles for a computation on an array of size *n* is expressed as *Cn + K*, where *C* is the cycles per element. The theoretical CPE is the value of C assuming the only factors are the cycles per issue and the latency of the integer multiplication unit in the processor.

Hint: focus on the first for loop, you do not need to take into account the second for loop for the theoretical CPE calculation.

Fill in the blanks by specifying values with **up to 2 decimal places**.

a) With p = ((p * x) * y) * z, the theoretical CPE is: ☐

b) With p = (p * (x * y)) * z, the theoretical CPE is: ☐

c) With p = p * ((x * y) * z), the theoretical CPE is: ☐

d) With p = (p * x) * (y * z), the theoretical CPE: ☐

Exam Question HS2

Consider the following code:

```
int product(int* a, int n){
   int i, x, y, z;
   int p = 1;
   for (i = 0; i < n-2; i+=3) {
      x = a[i];
      y = a[i+1];
      z = a[i+2];
      p = p * x * y * z; // PRODUCT CALCULATION
   }
   for (; i < n; i++) {
      p = p * a[i];
   }
   return p;
}
```

a) With p = ((p * x) * y) * z, the theoretical CPE is: ☐

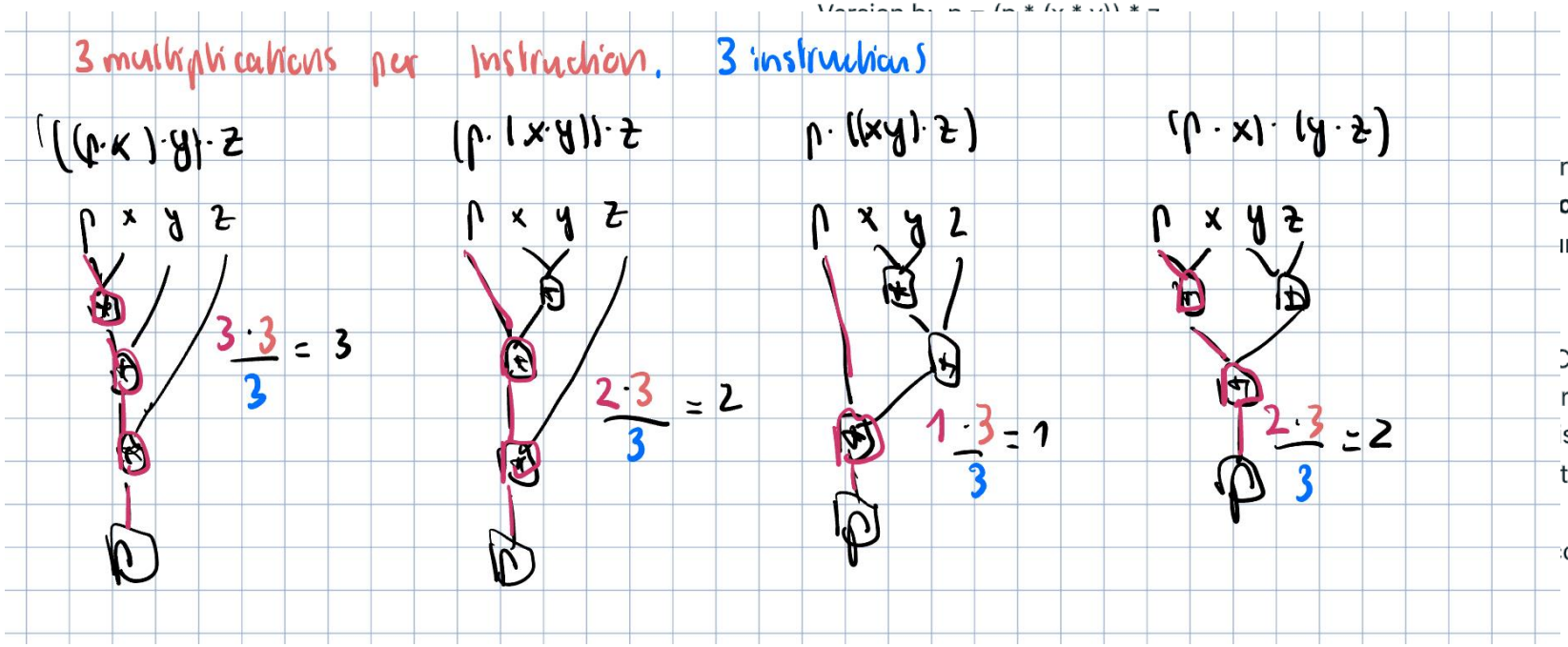
b) With p = (p * (x * y)) * z, the theoretical CPE is: ☐

c) With p = p * ((x * y) * z), the theoretical CPE is: ☐

d) With p = (p * x) * (y * z), the theoretical CPE: ☐

For the line marked "PRODUCT COMPUTATION" consider 4 different re-association options:

Version a: p = ((p * x) * y) * z



r has a **single integer multiplication unit** which is **cycle** (i.e., after issuing an integer multiplication, the ime the **latency of an integer multiplication is 3**

CT COMPUTATION option. Recall that the cycles per r a computation on an array of size *n* is expressed as s the value of C assuming the only factors are the t in the processor.

ount the second for loop for the theoretical CPE

Fill in the blanks by specifying values with **up to 2 decimal places**.

Exam Questions on Optimisation

HS22 Question 13

Question 13

[10 points]

Which of the following code optimizations can reduce the number of instructions that the processor needs to execute for a program? For each, write YES or NO and **explain your reasoning**. A correct answer with no explanation will receive no points.

i) Code inlining.

ii) Code motion.

iii) Loop unrolling.

iv) Reassociation.

v) Strength reduction.

Question 13

[10 points]

Which of the following code optimizations can reduce the number of instructions that the processor needs to execute for a program? For each, write YES or NO and **explain your reasoning**. A correct answer with no explanation will receive no points.

i) Code inlining.

ii) Code motion.

iii) Loop unrolling.

iv) Reassociation.

v) Strength reduction.

- i) Yes: no stackframe & no blackbox, i.e. allows for compiler optimisations in body
- ii) Yes: allows to precompute stuff like strlen(s)
- iii) Yes: we evaluate loop conditions less often
- iv) No: Changes only the **order** of operations
- v) No: Replaces one strong one with a weak one; may even need more for the weak ones

Systems@ETH Zürich

Exam Questions on Optimisation

HS21 Question 6

Question 6

[13 points]

a) Which of the following code optimizations can reduce the total number of instructions that will get executed in a program? For each, write YES or NO and **explain why**. A correct answer with no explanation will receive no points.

(4 points)

i) Loop unrolling.

ii) Precomputation.

iii) Vectorization (i.e., using SIMD instructions).

iv) Reassociation.

- i) Yes: we evaluate loop conditions less often
- ii) Yes: allows to precompute stuff like strlen(s)
- iii) Yes: one instruction for multiple OPs
- iv) No: Changes only the **order** of operations

b) Consider the following code:

```
void lowercase(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
        s[i] -= ('A' - 'a');
        }
    }
}
```

The execution time of this code is quadratic in relation to the length of the string. Explain why. Also describe one simple modification to the code that would make execution time linear in relation to the length of the string. What is this type of optimization called?

(3 points)

Would the `gcc` compiler apply the modification you described automatically when running with optimization flag `-O` greater than 0? Explain why or why not.

(2 points)

b) Consider the following code:

```
void lowercase(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
        s[i] -= ('A' - 'a');
        }
    }
}
```

The execution time of this code is quadratic in relation to the length of the string. Explain why. Also describe one simple modification to the code that would make execution time linear in relation to the length of the string. What is this type of optimization called?

(3 points)

Would the `gcc` compiler apply the modification you described automatically when running with optimization flag `-O` greater than 0? Explain why or why not.

(2 points)

- Precompute strlen(s); Called **precomputation/code motion**
- **Compiler would not** automatically do it: "blackbox" function, may have side effects: must not change semantics of the program

Measuring (µ-arch) performance

Expressing Program Performance

Systems@ETH Zürich

On the program level:

- Execution Time = IC • CPI • CCT
  - IC = instruction count
  - CPI = cycles per instruction (= 1/IPC)
  - CCT = clock cycle time (= 1/Frequency)

- Unable to quantify performance for individual program sections.

Expressing Program Performance

Systems@ETH Zürich

On the instruction level:

- Throughput (instructions / cycle)
- Latency (cycles / instruction)

- Each depend on the type of instruction
- What to optimize for?

Thought for later: How do these differ when considering *a single* vs. *many* instructions (on average)?

Expressing Program Performance

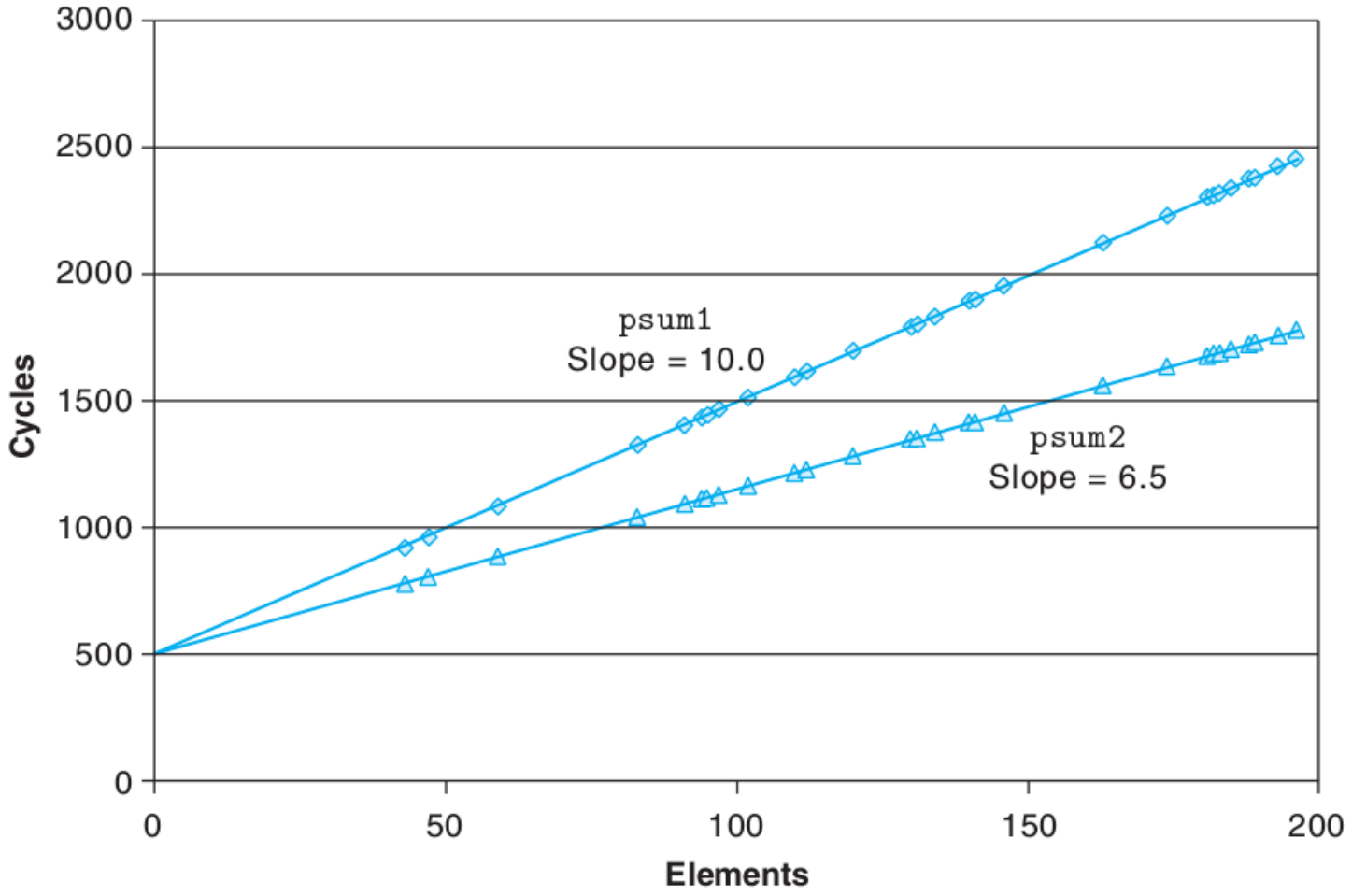
Systems@ETH Zürich

Benchmarking array-like operations:

- Cycles per Element (CPE):
  - Execution time = CPE*n + overhead
  - Independent of clock time?
  - Highly variable for different code sections
  - Hard Lower bound - why?

Measuring CPE

```
void psum1(float a[], float p[], long int n) {

    long int i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}

void psum2(float a[], float p[], long int n) {

    long int i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a[i];
        p[i] = mid_val;
        p[i+1] = mid_val + a[i+1];
    }

    if (i < n) p[i] = p[i-1] + a[i];

}
```



What do we need to know to calculate lower bound?

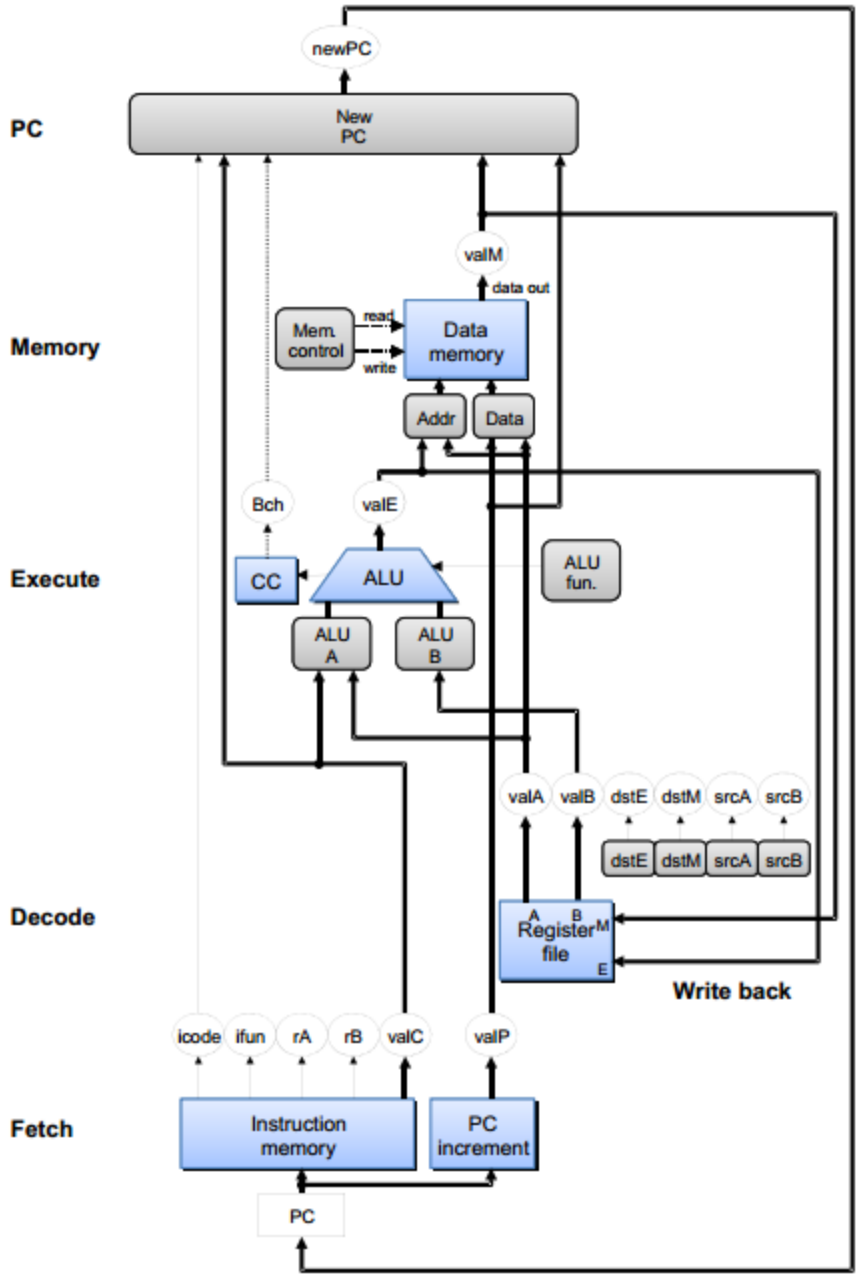
Improving µ-arch performance pt.1

Instruction parallelism (pipelining and superscalar execution)

Once upon a time…

- Sequential Processor Design (very long ago: single-cycle)
- Each instruction must be
  - Fetched
  - Decoded
  - Executed
  - Written back
- And the new PC determined

What are issues with this design?



Instruction parallelism / Pipelining

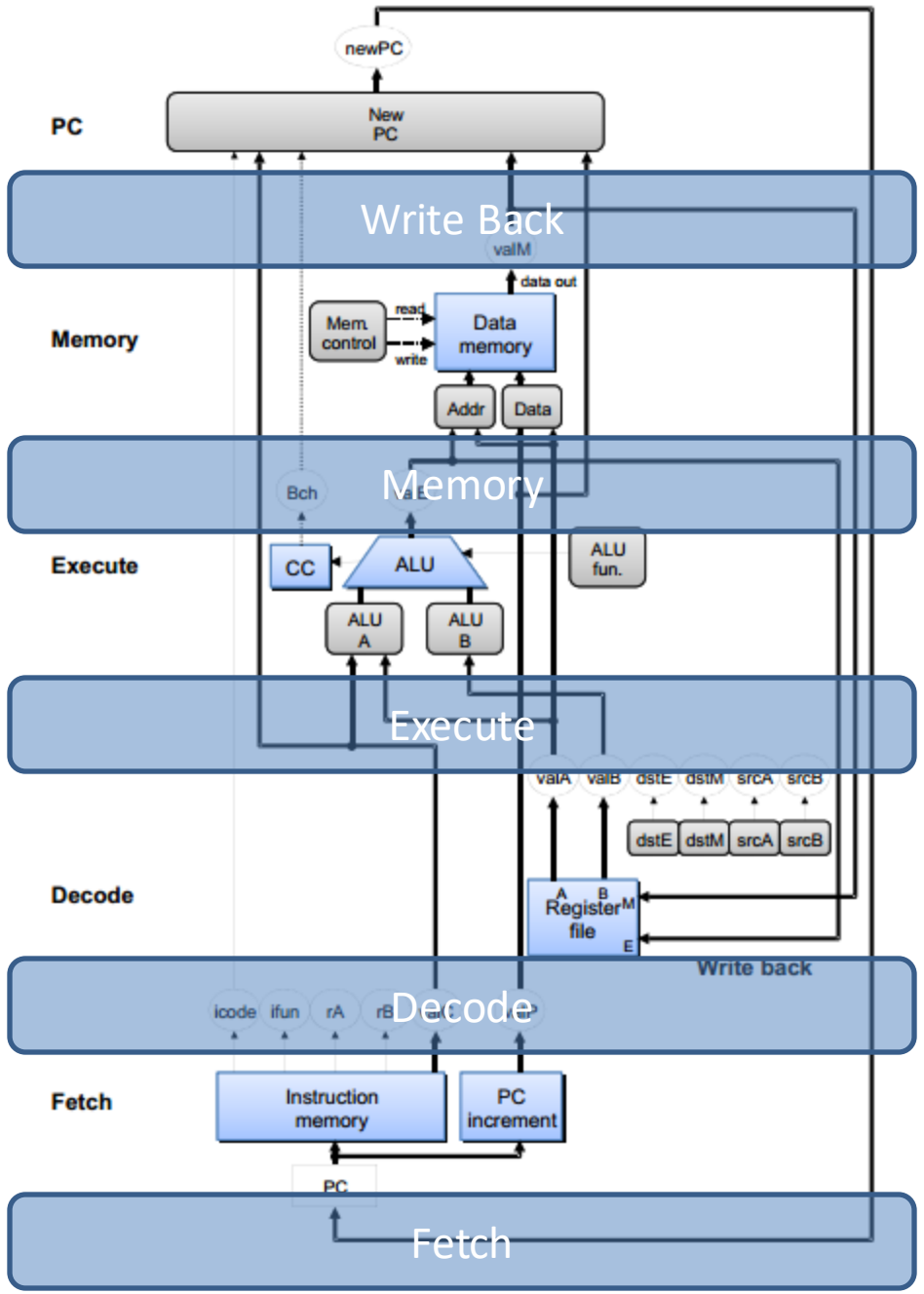
Systems@ETH Zürich

- Observation 1: higher clock frequencies as main source of performance improvement -> we reached physical limits there.
- Observation 2: Individual hardware units are idle most of the time.
- Idea: Overlap instruction execution.

Pipelining: example

How it's done in the CPU?

- Divide the Hardware into stages
- Insert registers in between that hold the intermediate values
- Add pipeline control logic



Pipelining performance

Systems@ETH Zürich

- Core Idea: Overlap instruction execution.
- Yields increase in throughput (and mean latency) proportional to #(pipeline stages) – assuming a full pipeline.
- Insight: We now have additional parameters – pipeline depth & pipeline “fullness”.

Pipeline: All Gold?

Any Problems here?

Pipeline: All Gold?

Pipelining introduces **data hazards** (read after write dependency)

Idea: Give the result value as early as possible

Applying CPE to pipelined execution

```
void combine(vec_ptr v, data_t *dest) {
  long int i;
  long int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;

  for (i = 0; i < length; i++) {
    acc = acc OP data[i];
  }
  *dest = acc;
}
```

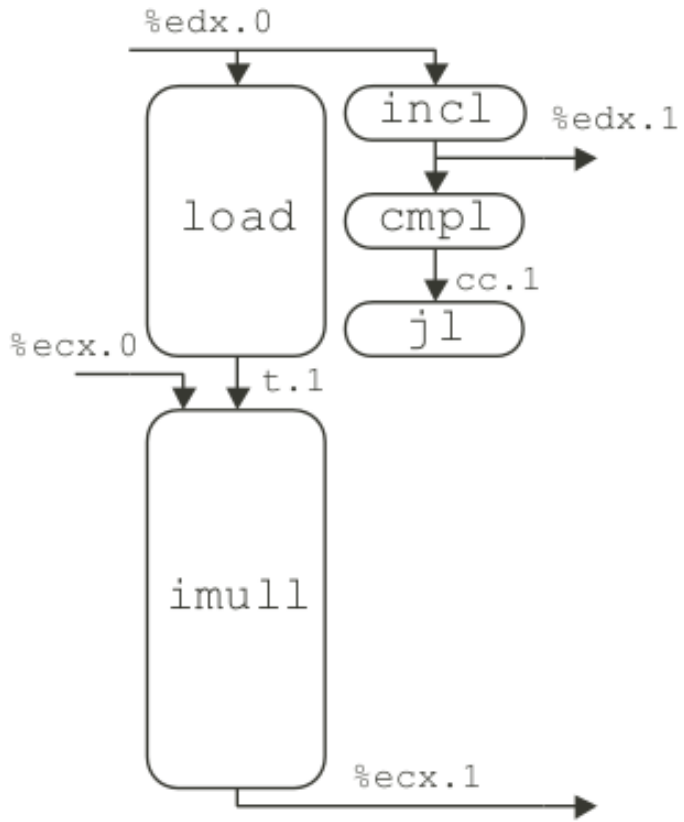
Applying CPE to pipelined execution

```
void combine(vec_ptr v, data_t *dest) {
  long int i;
  long int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = 1;

  for (i = 0; i < length; i++) {
    acc = acc * data[i];
  }
  *dest = acc;
}
```

Assembly Instructions	Execution Unit Operations
.L24:	
imull (%eax,%edx,4),%ecx	load (%eax, %edx.0, 4) → t.1
	imull t.1, %ecx.0 → %ecx.1
incl %edx	incl %edx.0 → %edx.1
cmpl %esi,%edx	cmpl %esi, %edx.1 → cc.1
jl .L24	jl-taken cc.1

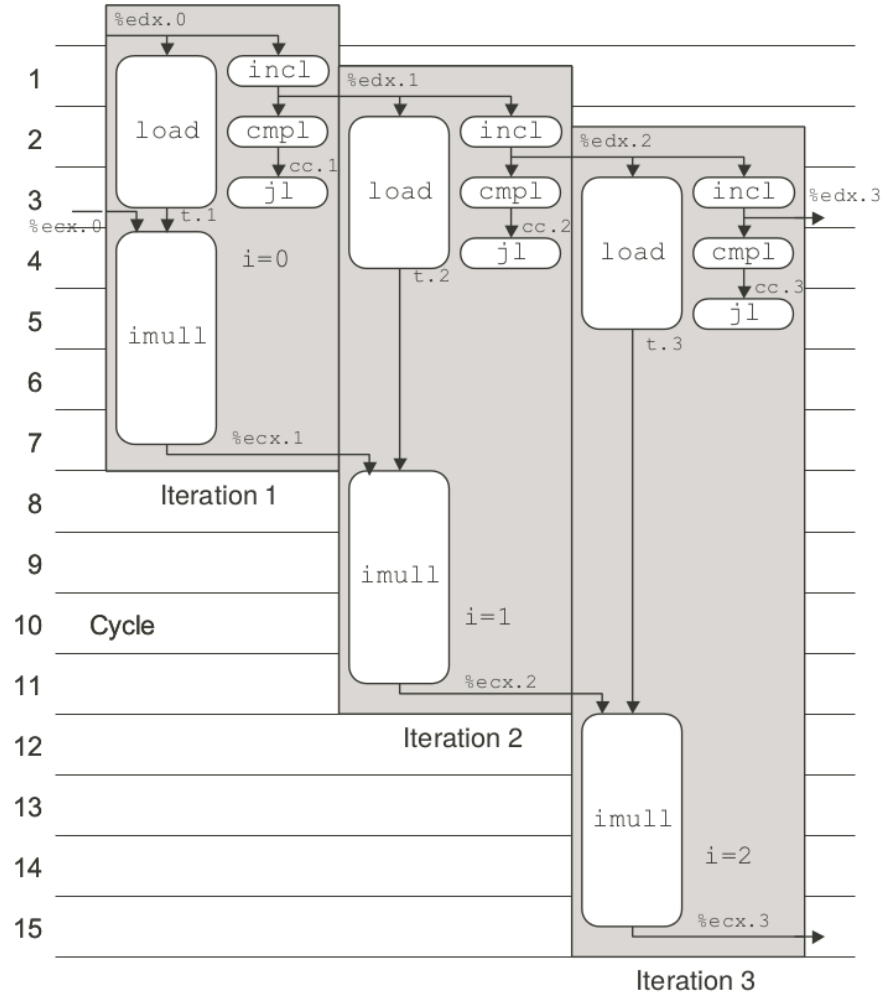
Scheduling Execution Units

Execution Unit Operations		
load (%eax, %edx.0, 4)	→	t.1
imull t.1, %ecx.0	→	%ecx.1
incl %edx.0	→	%edx.1
cmpl %esi, %edx.1	→	cc.1
jl-taken cc.1		



Lower bound CPE – with data dependencies

- Assuming perfect branch prediction & multiple load units
- CPE bound by critical path



How to improve?

- Current problems:
  - stages with different #instructions / cycle
  - data / control hazards causing stalls

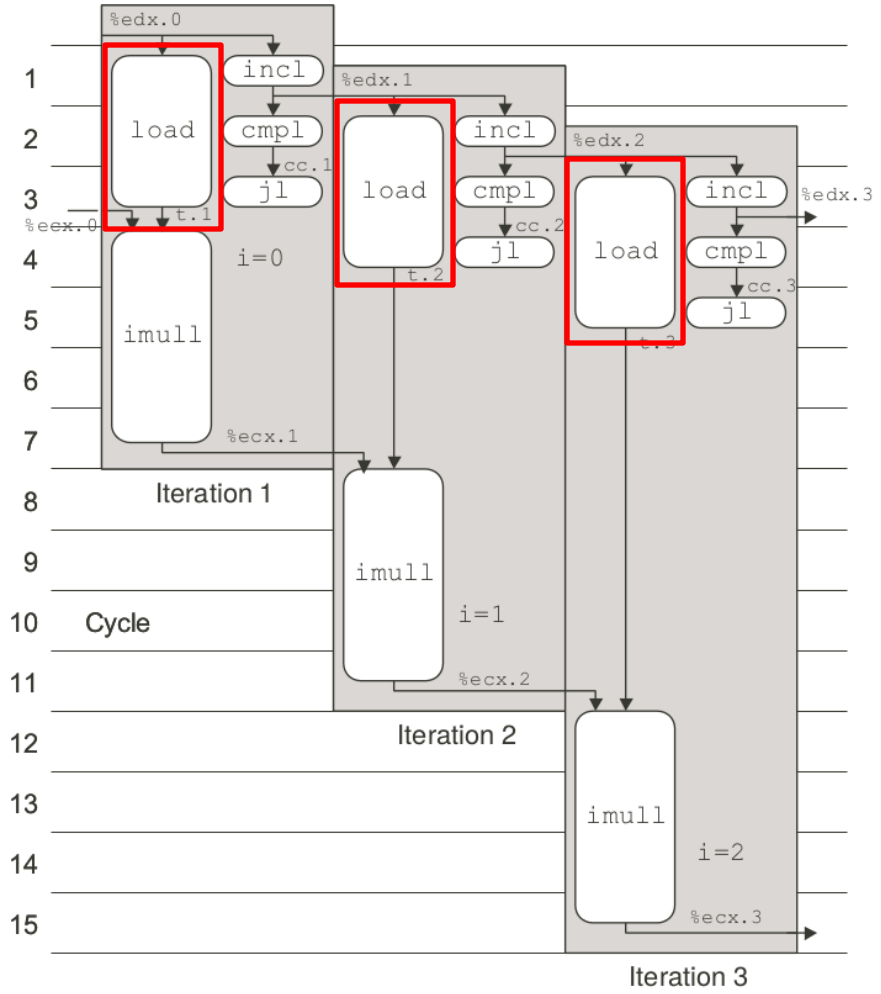
- Option 1: increase pipeline depth. How (far)? Why (not)?
- Option 2: replicate pipeline (components). How (far)? Why (not)?

Option 1: increase pipeline depth

- Theoretical speedup = #stages
- Observation: increases #instructions in pipeline at any given time
  - smart(er) scheduling needed
  - control hazards: branch mispredictions become more costly
  - data hazards: more complex to apply work-arounds (forwarding/bypasses/reordering)

Option 2: replicate pipeline (components)

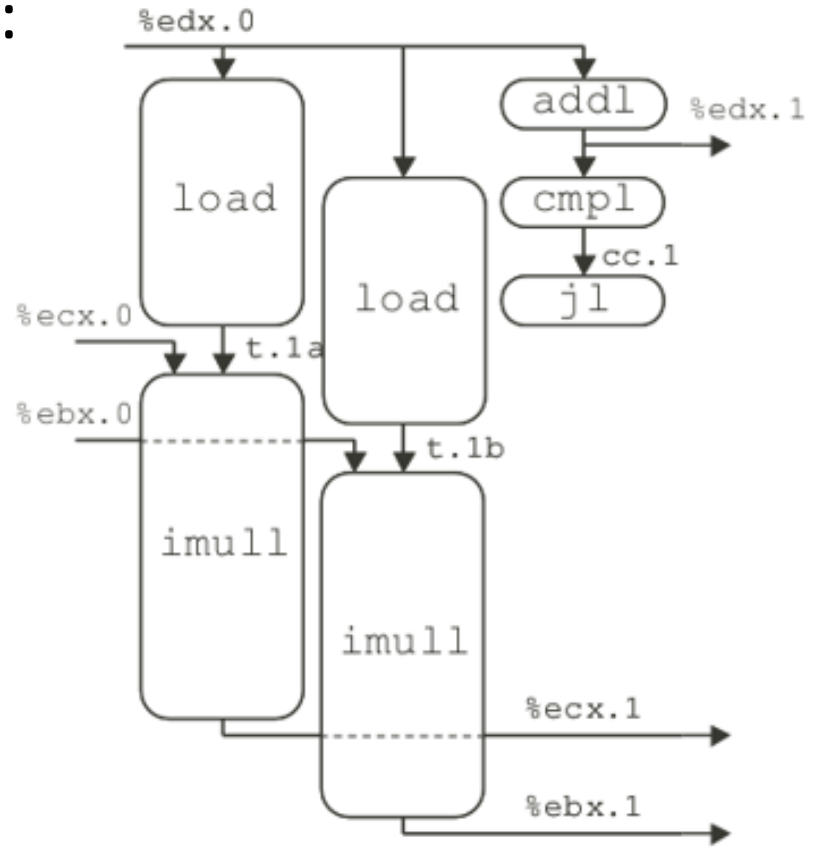
- **Step 1:** replicate components representing bottlenecks (load/store units, FPU, …)
- Ideally, all pipeline stages have the same throughput (#instructions / cycle), on average



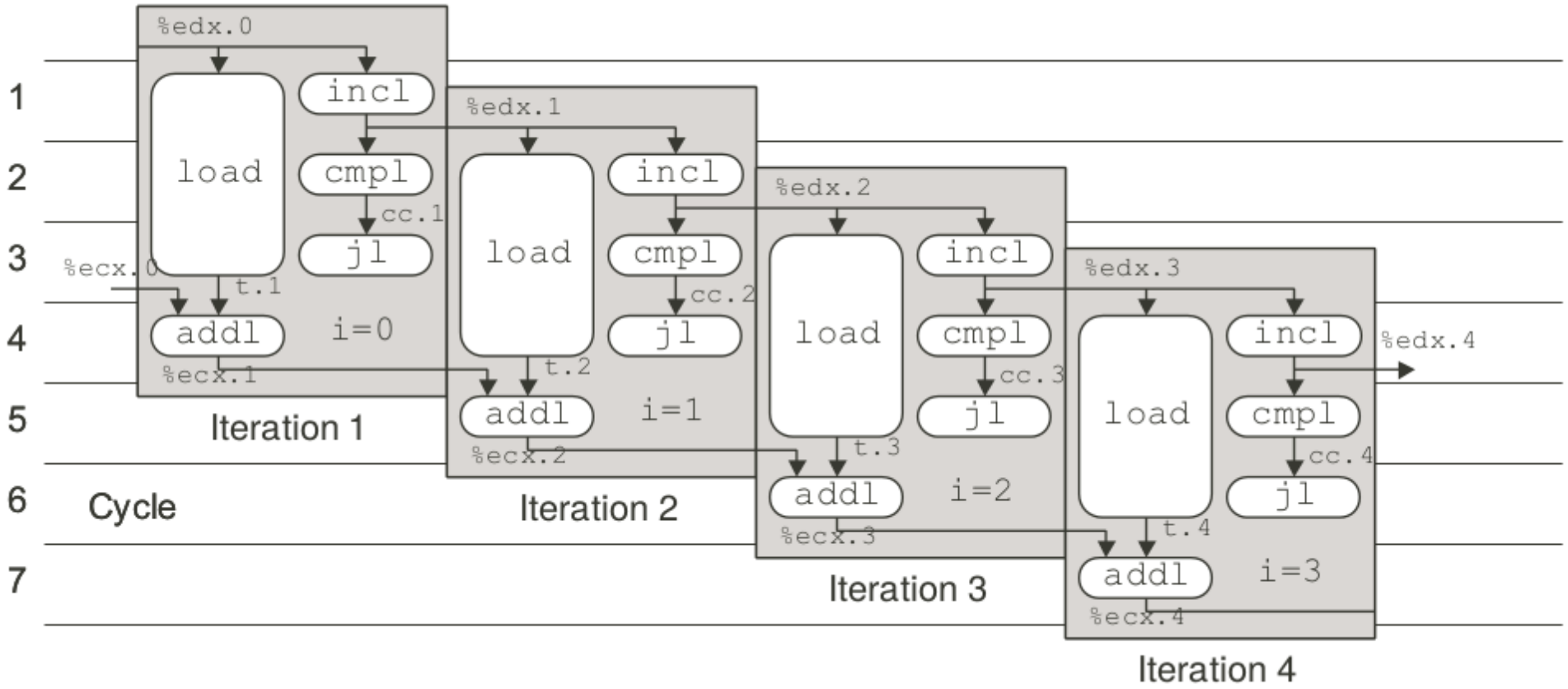
Improving CPE – multiple functional units

Combine two elements at the time:

```
for (i = 0; i < limit; i+=2) {
  x0 = x0 OP data[i];
  x1 = x1 OP data[i+1];
}
*dest = x0 OP x1;
```

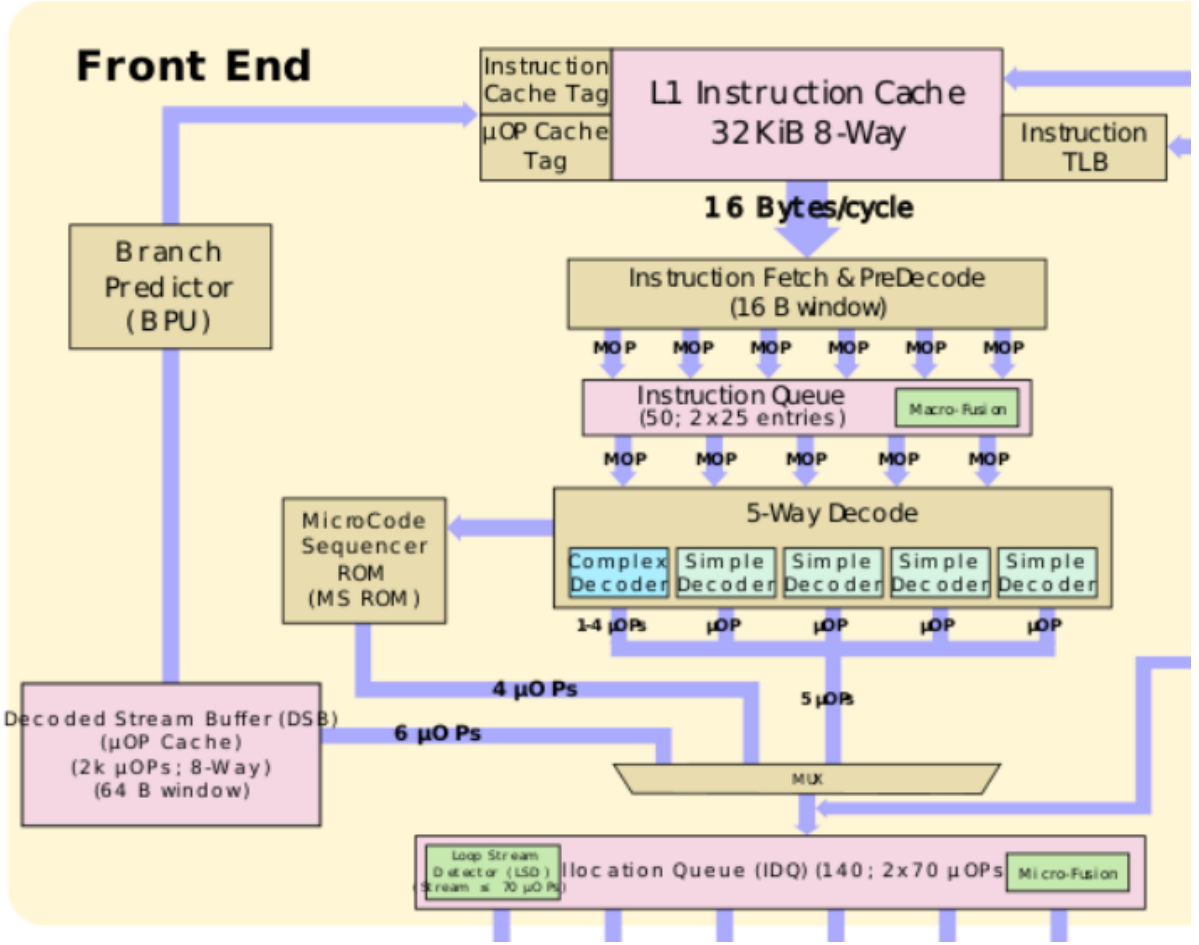


Lower bound CPE – single instruction latency



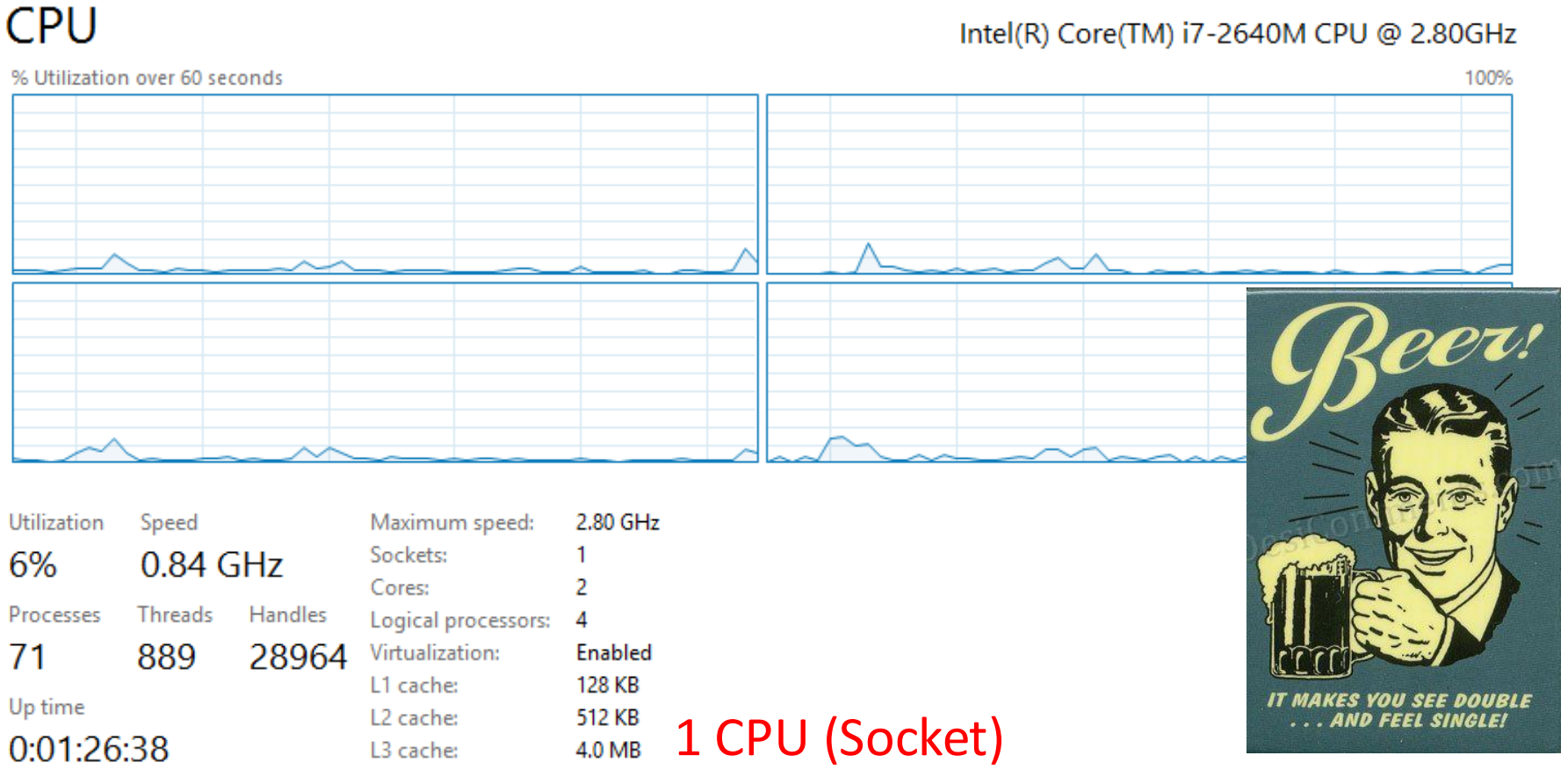
Option 2: replicate pipeline (components)

- **Step 2:** Issue multiple instructions every cycle, possibly out-of-order.
- Called superscalar execution
- Once again:
  - assume *many* instructions are available for scheduling
  - Higher penalty for mis-predictions & hazards



Intel Sunny Cove, 10nm 2019 (total: 14-19 pipeline stages, 6-way multi-issue)

Simultaneous Multithreading (SMT) from OS View



1 CPU (Socket)
2 Cores (Physical Processors)
4 Logical Processors (“Threads”)

Systems@ETH Zürich

Improving µ-arch performance pt.2

Data parallelism - **S**ingle **I**nstruction **M**ultiple **D**ata instructions

Using SIMD Instructions

Systems@ETH zürich

- Use vector-instructions instead of scalar-instructions
- Many different extensions to x86 ISA
  - MMX, SSE, SSE2, FMA3, FMA4, CLMUL, AVX, AVX2, AVX10, etc.
- Compiler can (under certain conditions) vectorize code itself. [1]
  - Enabled by `gcc -O3` or specific flags such as `-ftree-vectorize`.
  - Use `gcc -march=native` to make use of all architecture-specific optimizations on your machine.

[1] https://gcc.gnu.org/projects/tree-ssa/vectorization.html

Automatic Vectorization

```c
int a[256], b[256], c[256];

int main (int argc, char* argv[]) {
  int i;

  for (i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

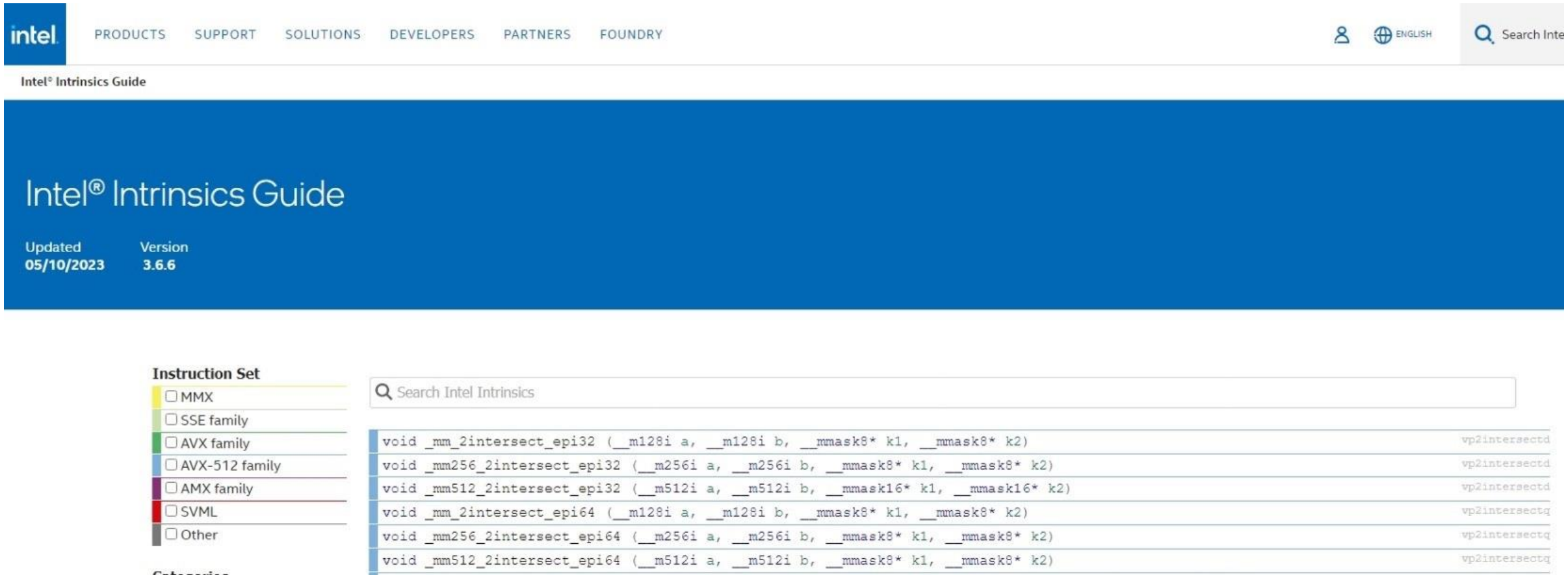
```
$ gcc -Wall -fopt-info-all -O3 -march=native -S -o autovec.S autovec.c
Unit growth for small function inlining: 13->13 (0%)
Inlined 0 calls, eliminated 0 functions
autovec.c:8:14: optimized: loop vectorized using 16 byte vectors
autovec.c:5:5: note: vectorized 1 loops in function.
autovec.c:5:5: note: ***** Analysis failed with vector mode VOID
```

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        leaq    a(%rip), %rsi
        xorl    %eax, %eax
        leaq    b(%rip), %rcx
        leaq    c(%rip), %rdx
        .p2align 4,,10
        .p2align 3
.L2:
        vmovdqa (%rcx,%rax), %xmm1
        vpaddd  (%rdx,%rax), %xmm1, %xmm0
        vmovdqa %xmm0, (%rsi,%rax)
        addq    $16, %rax
        cmpq    $1024, %rax
        jne     .L2
        xorl    %eax, %eax
        ret
        .cfi_endproc
```

Explicit Vectorization

- Can also write explicitly vectorized code.
- Offers better control and more advanced optimizations
  - Use intrinsics or built-ins
  - Need to take care of portability!

Explicit Vectorization



https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

```c
#include <immintrin.h>
#include <stdio.h>

void addArraysAVXAligned(float *a, float *b, float *result, int size) {
    int avxSize = 8;  // AVX supports 8 floats / instruction
    int avxOperations = size / avxSize;
    for (int i = 0; i < avxOperations; ++i) {
        // Load 256 bits (8 floats) into AVX registers
        __m256 avx_a = _mm256_load_ps(&a[i * avxSize]);
        __m256 avx_b = _mm256_load_ps(&b[i * avxSize]);
        // Perform addition using AVX
        __m256 avx_result = _mm256_add_ps(avx_a, avx_b);
        _mm256_store_ps(&result[i * avxSize], avx_result);
    }
    for (int i = avxOperations * avxSize; i < size; ++i) {
        result[i] = a[i] + b[i];
    }
}

int main() {
    // Size of the arrays
    int size = 18;
    // Ensure proper alignment (32 bytes for AVX)
    size_t alignment = 32;
    // Allocate aligned memory for arrays using _mm_malloc
    float *a = (float*)_mm_malloc(size * sizeof(float), alignment);
    float *b = (float*)_mm_malloc(size * sizeof(float), alignment);
    float *result = (float*)_mm_malloc(size * sizeof(float), alignment);
```

```c
// Initialize arrays with some values
    for (int i = 0; i < size; ++i) {
        a[i] = i;
        b[i] = 2 * i;
    }
    addArraysAVXAligned(a, b, result, size);
    printf("Result: ");
    for (int i = 0; i < size; ++i) {
        printf("%.1f ", result[i]);
    }
    printf("\n");
    _mm_free(a);
    _mm_free(b);
    _mm_free(result);
    return 0;
}
```

Support your local µ-arch

Compile-time optimizations

Compile-time optimization

Systems@ETH Zürich

- Key rationale: Hardware is more *expensive* to improve than software.
- But: Compiler must preserve program semantics.
- So-called *optimization blockers* make it hard for the compiler to check whether a given optimization strategy can be applied.
- Examples
  - Procedure Calls
  - Memory Aliasing

Procedure Calls

```c
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

?=

```c
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

Optimization idea: Move the call to `strlen` out of the for loop.

Procedure Calls

```c
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

```c
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++) {
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] -= ('A' - 'a');
    }
  }
}
```

- Procedures can have side-effects
- Compiler treats them as black-boxes
- Solutions:
  - Inline the function where possible or
  - manually move the call out of the loop.

Memory Aliasing

- Especially problematic in C (allows address arithmetic)

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

?

=

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

Optimization idea: Can we accumulate `b[i]` in a processor register, instead of going to memory?

Memory Aliasing

Systems@ETH Zürich

What will happen when running the following code?

```
int main(int argc, char* argv[]) {

double A1[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double A2[9];

memcpy(A2, A1, sizeof(A1));

double *B1 = A1+3;
double *B2 = A2+3;

sum_rows1(A1, B1, 3);
sum_rows2(A2, B2, 3);

for (int i = 0; i < 9; i++)
    assert(A1[i] == A2[i]);

}
```

```
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

Memory Aliasing

```
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

- Functions are only equal under certain assumptions.
- Compiler cannot tell whether a and b are aliases for each other.
- To enable the compiler to generate optimized code
  - explicitly rewrite using local accumulator or
  - use the `restrict` keyword.

Using the `restrict` keyword (since C99)

```
void sum_rows1(double * restrict a, double * restrict b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

- Tells the compiler that, for the lifetime of the `restrict`-annotated pointer, it will be the only pointer used to access the underlying memory.
- Programmer responsible for making sure this is the case.

Now we can discuss:

Systems@ETH zürich

- How to…
  - … improve processor *performance** past Moore's Law?
  - … improve the improvement?
  - … deal with practical challenges of performance hacks?

    * how to define and measure performance?
- Up next: your turn!

Assignment 9: Compiler Optimizations

Assignment 9

Systems@ETH Zürich

Some options for optimizing matrix multiplication:

- Loop unrolling
- Cache optimization (Blocking, Locality…)
- Compiler optimization
- Vectorization
- …

Assignment 9

Systems@ETH Zürich

Understanding what your program does:

- Use perf (Good)
  https://perf.wiki.kernel.org/index.php/Tutorial
- Use Intel V-Tune (Good)
  Order it from IDES or download trial version online

Curious what the compiler optimized?

Systems@ETH Zürich

- Use `gcc -fopt-info -all` (optimized + missed + note)
  - `-optimized` (applied optimizations)
  - `-missed` (missed optimizations)
  - `-note` (print verbose info about optimizations)

```
$ gcc -Wall -mavx -fopt-info-all -O3 -o main main.c
main.c:22:9: optimized:   Inlining _mm256_store_ps/917 into addArraysAVXAligned/5512 (always_inline).
main.c:19:29: optimized:   Inlining _mm256_add_ps/833 into addArraysAVXAligned/5512 (always_inline).
main.c:26:45: optimized: loop vectorized using 16 byte vectors
main.c:13:23: missed: couldn't vectorize loop
```

https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html

Cycle Counter

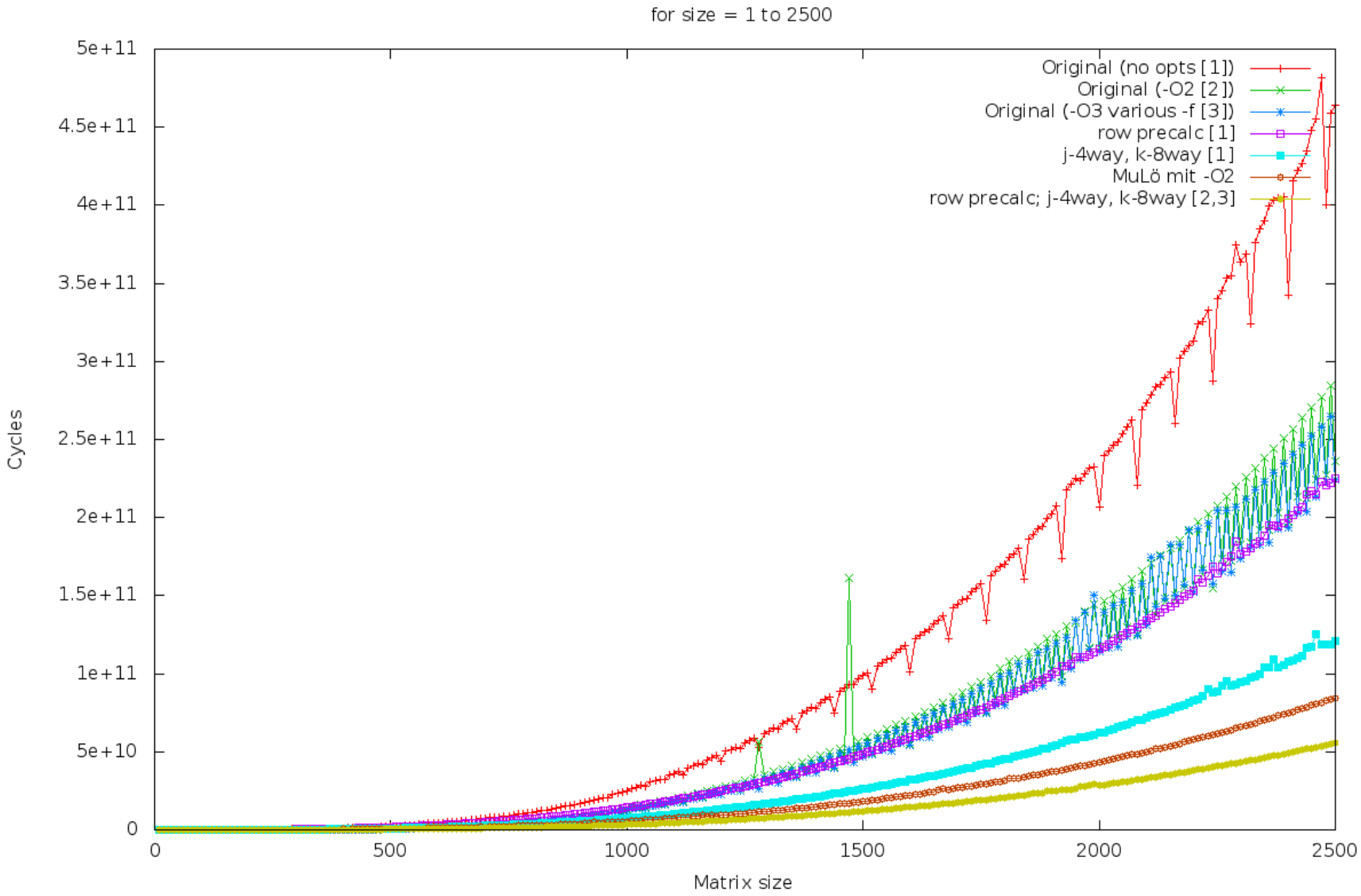
- You will need to access the cycle counter of your CPU!
  - One function to start the counter
  - One function to get the counter value

```
1. void start_counter();
2.
3. unsigned long get_counter();
```

Cycle Counter

- The value you need is located in the rdtsc register
  - Use inline assembly to load the value into your variables (see slides of the first lecture)

Example Solution



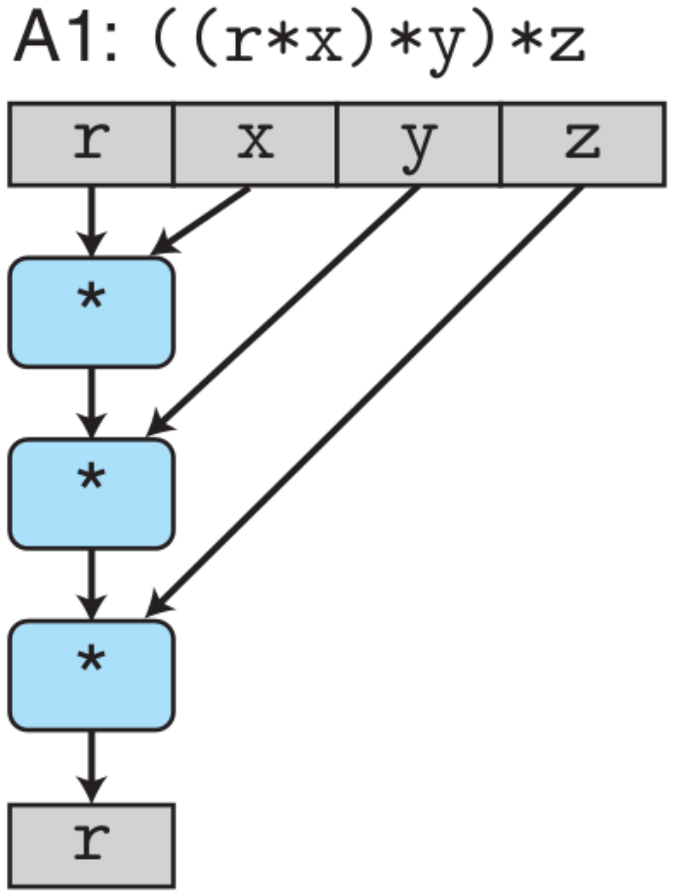
Assignment 09 Question 1

```
double aprod(double a[], int n)
{
  int i;
  double x, y, z;
  double r = 1;
  for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = r * x * y * z;
  }
  for (; i < n; i++)
    r *= a[i];
  return r;
}
```

- 3-way loop unrolling
  - 3 Execution per loop
- Find critical path
- Divide by 3

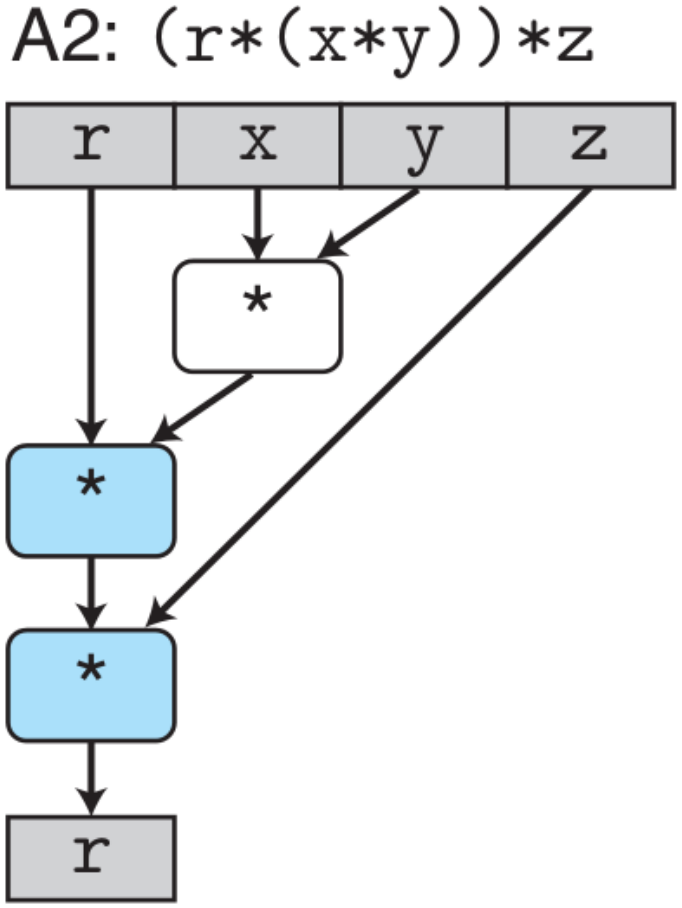
Assignment 09 Question 1

```
for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = ((r * x) * y) * z;
}
```



Assignment 09 Question 1

```
for (i = 0; i < n-2; i+= 3) {
    x = a[i];
    y = a[i+1];
    z = a[i+2];
    r = (r * (x * y)) * z;
}
```



Have a nice week!