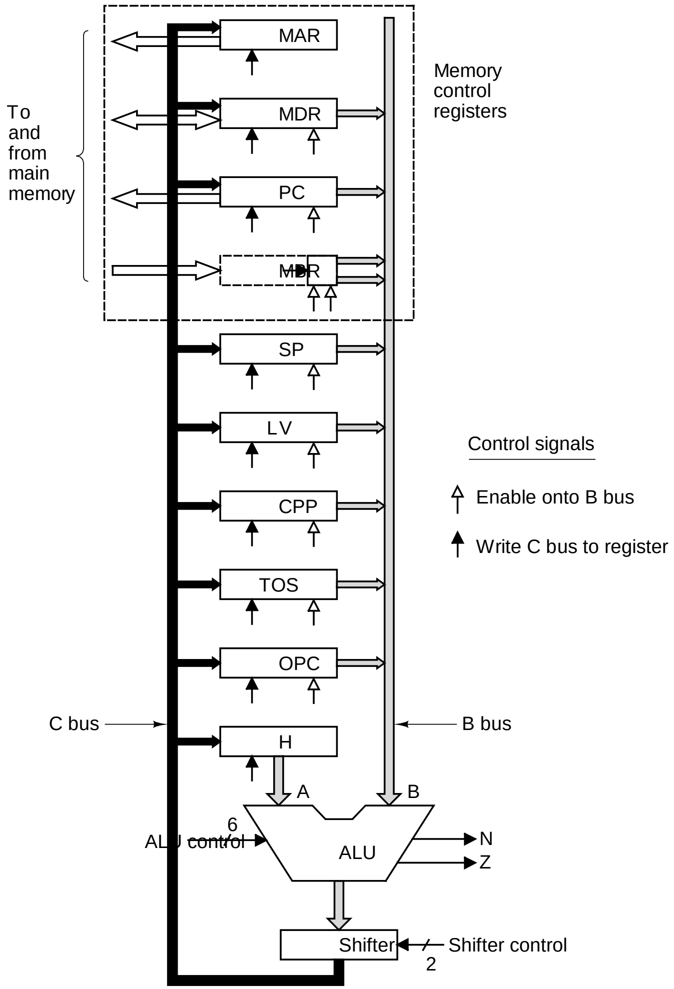
TI II: Computer Architecture Microarchitecture



Microprocessor Architecture

Microprogramming

Pipelining (superscalar, multithreaded, hazards, prediction, vector processing)

Content

1. Introduction
   - Single Processor Systems
   - Historical overview
   - Six-level computer architecture

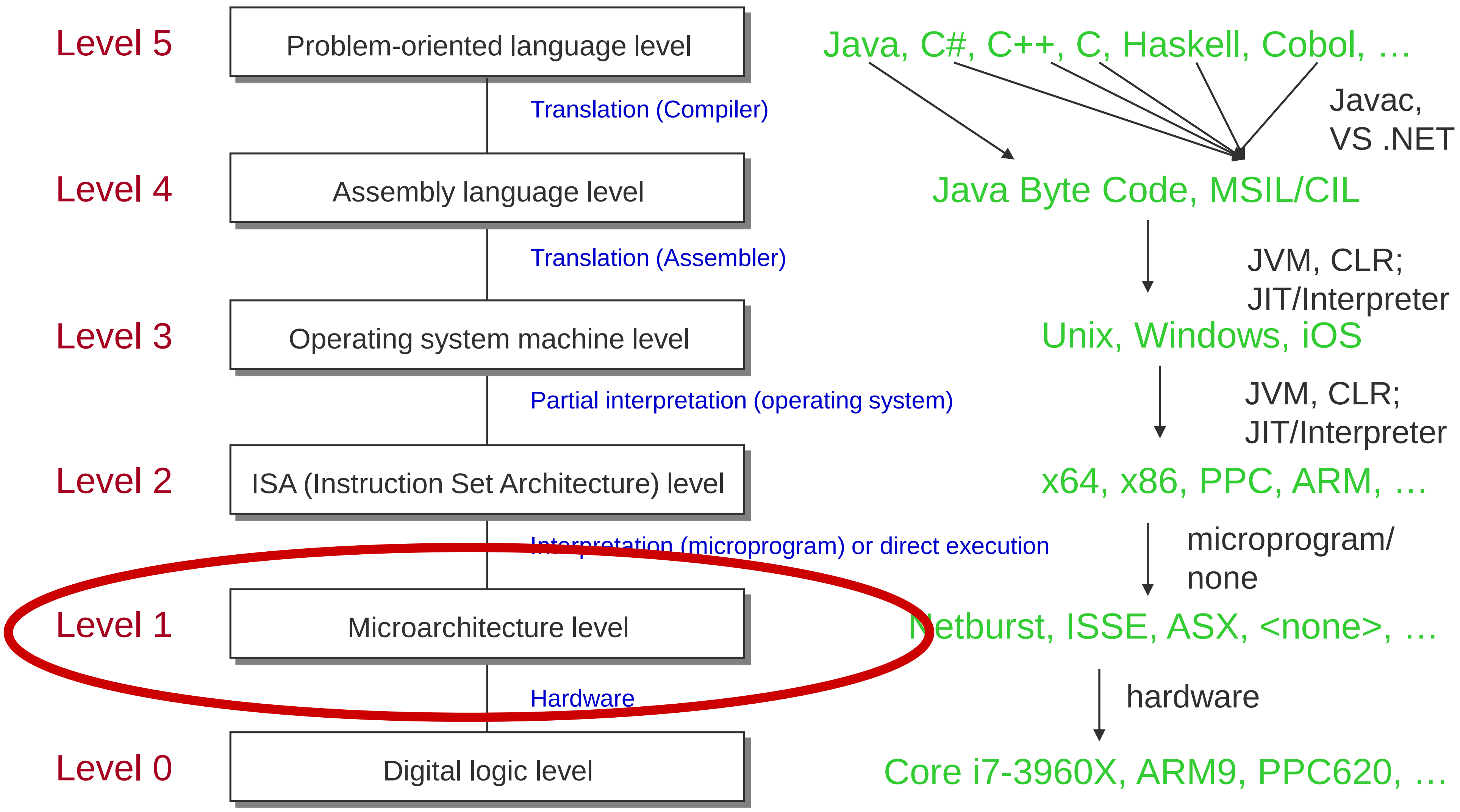
2. Data representation and Computer arithmetic
   - Data and number representation
   - Basic arithmetic

3. **Microarchitecture**
   - **Microprocessor architecture**
   - **Microprogramming**
   - **Pipelining**

4. Instruction Set Architecture
   - CISC vs. RISC
   - Data types, Addressing, Instructions
   - Assembler

5. Memories
   - Hierarchy, Types
   - Physical & Virtual Memory
   - Segmentation & Paging
   - Caches

Where are we now? - The Six-Level-Computer



Level 5 — Problem-oriented language level

Translation (Compiler)

Level 4 — Assembly language level

Translation (Assembler)

Level 3 — Operating system machine level

Partial interpretation (operating system)

Level 2 — ISA (Instruction Set Architecture) level

Interpretation (microprogram) or direct execution

Level 1 — Microarchitecture level

Hardware

Level 0 — Digital logic level

Java, C#, C++, C, Haskell, Cobol, …

Javac,
VS .NET

Java Byte Code, MSIL/CIL

JVM, CLR;
JIT/Interpreter

Unix, Windows, iOS

JVM, CLR;
JIT/Interpreter

x64, x86, PPC, ARM, …

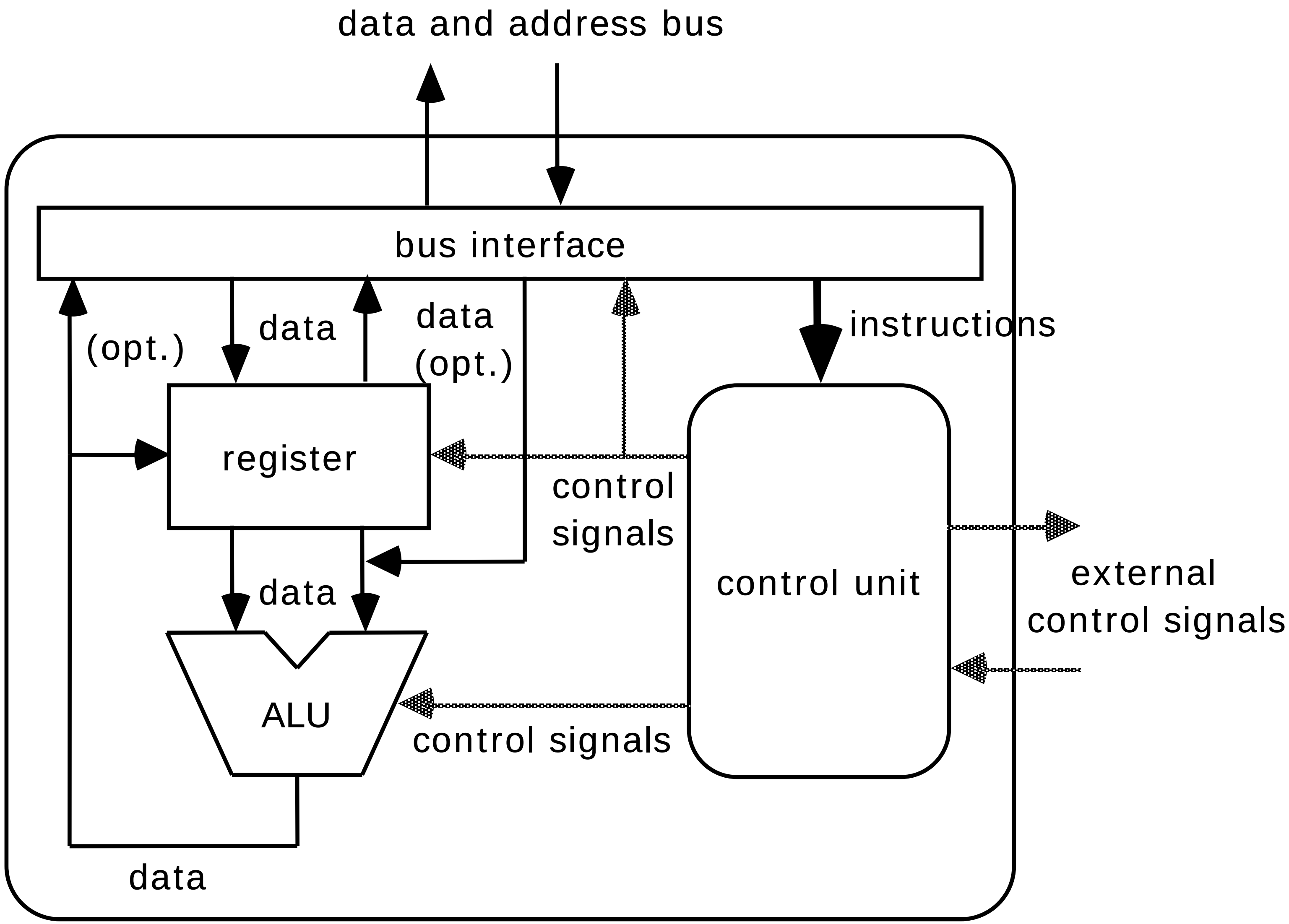
microprogram/
none

Netburst, ISSE, ASX, <none>, …

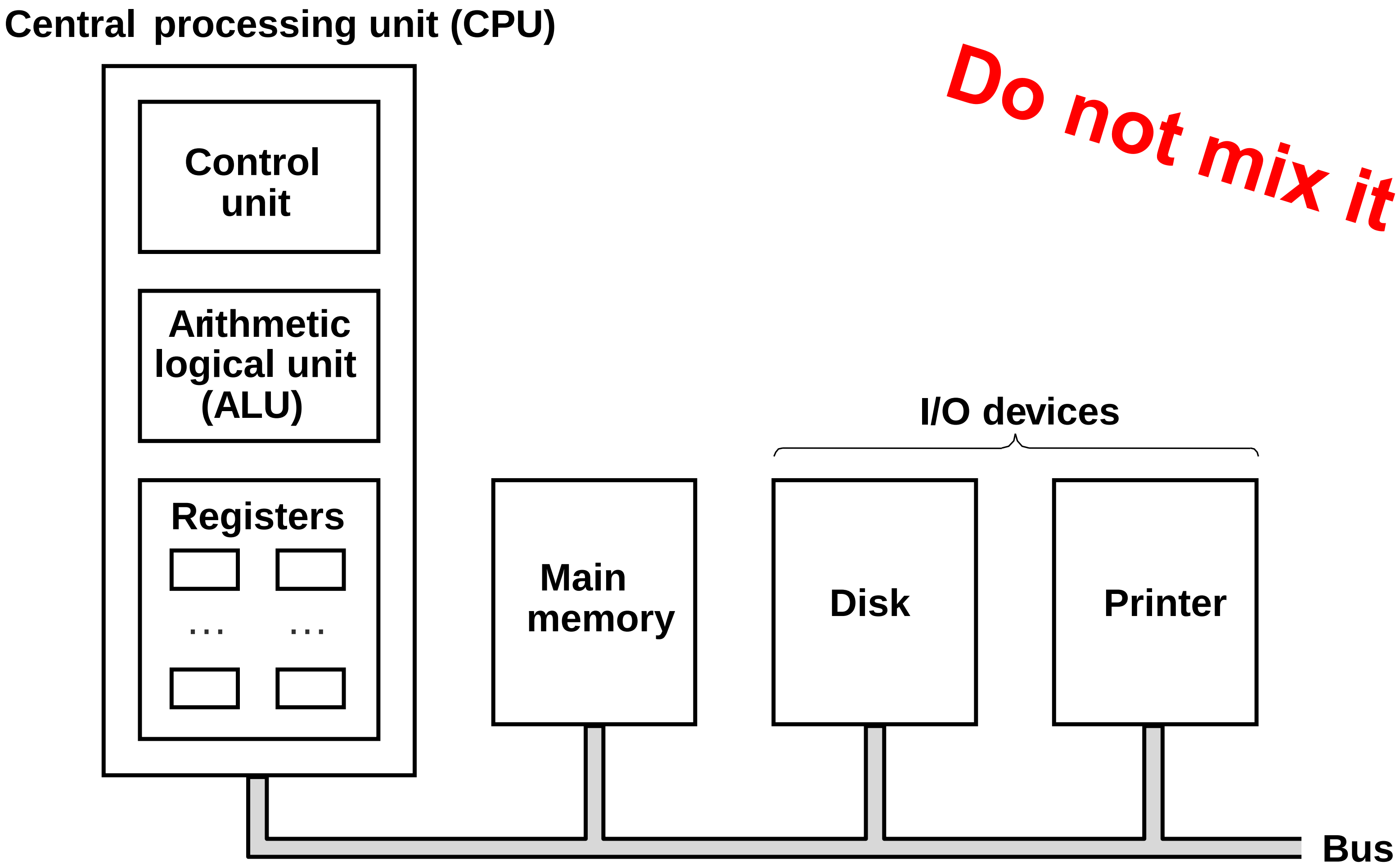
hardware

Core i7-3960X, ARM9, PPC620, …

Basic architecture of a simple micro processor

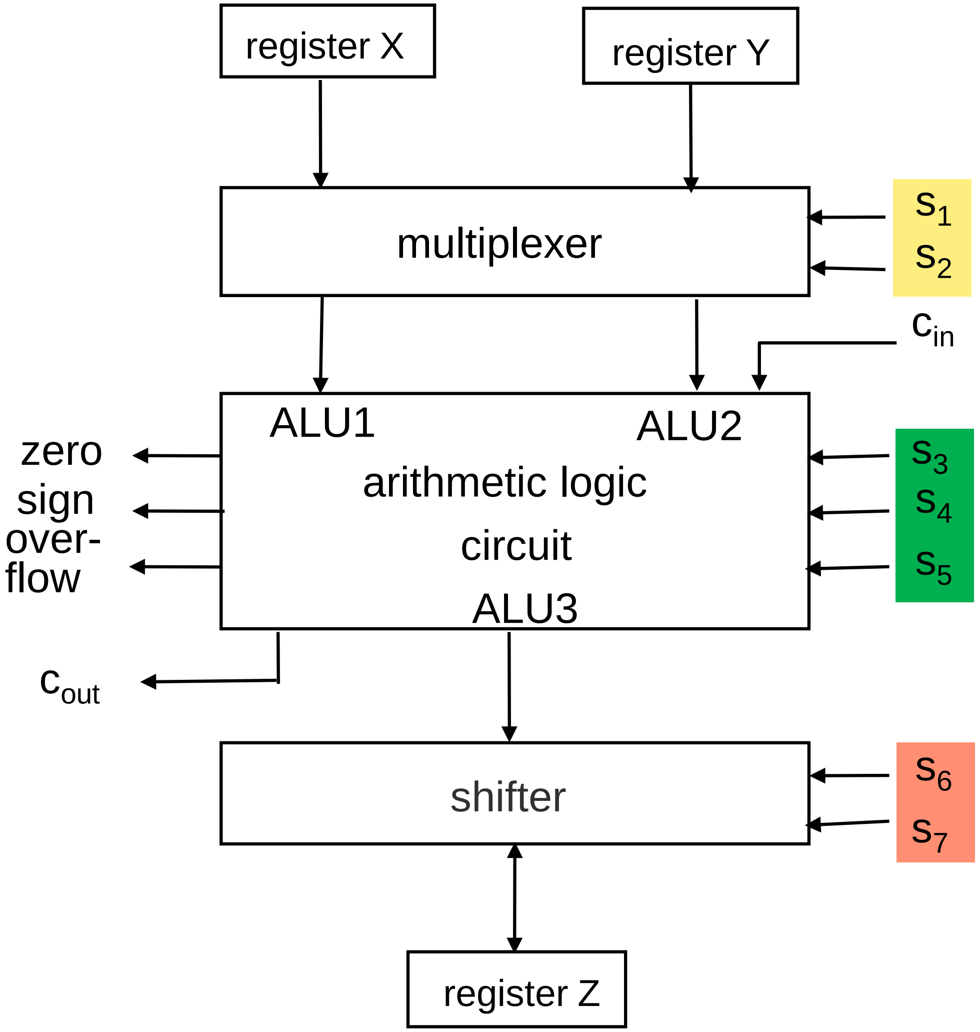


Basic architecture of a simple microcomputer



Do not mix it up!

Basic architecture of a simple ALU – see chapter 2

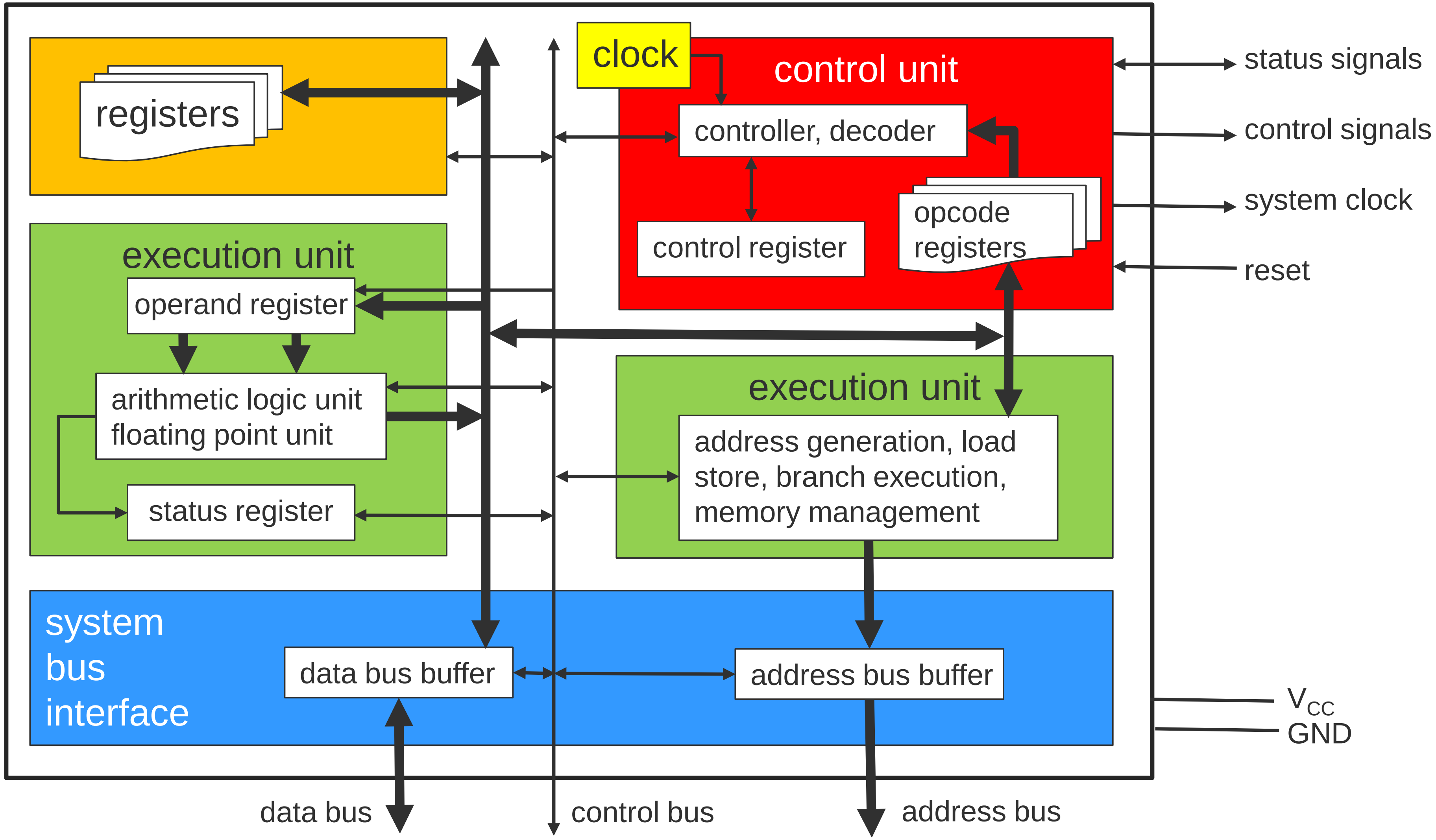


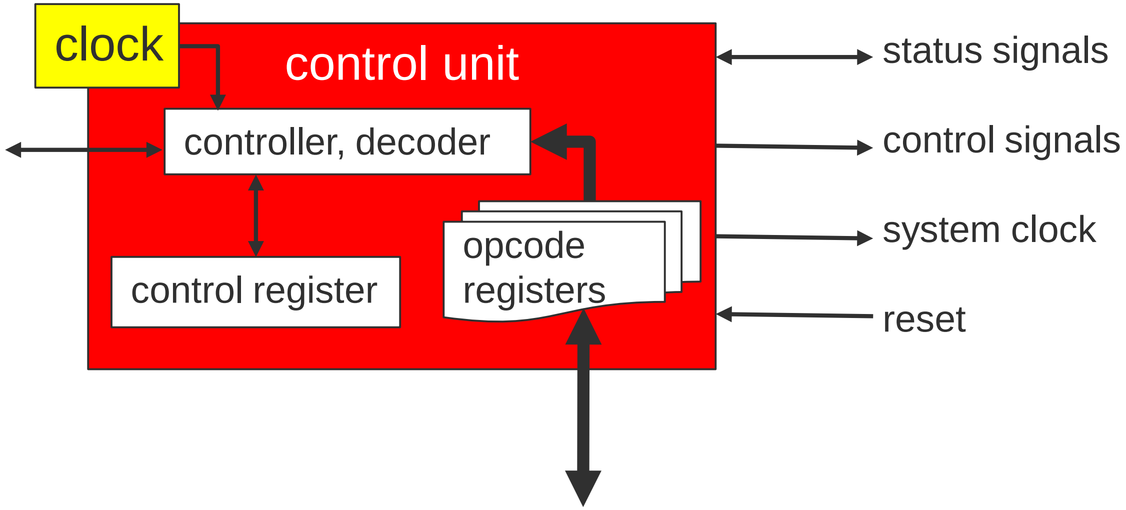
s_1	s_2	ALU1	ALU2
0	0	X	Y
0	1	X	0
1	0	Y	0
1	1	Y	X

s_3	s_4	s_5	ALU3
0	0	0	ALU1 + ALU2 +c_{in}
0	0	1	ALU1 – ALU2 – Not(c_{in})
0	1	0	ALU2 – ALU1 – Not(c_{in})
0	1	1	ALU1 ∨ ALU2
1	0	0	ALU1 ∧ ALU2
1	0	1	Not(ALU1) ∧ ALU2
1	1	0	ALU1 ⊕ ALU2
1	1	1	ALU1 ↔ ALU2

s_6	s_7	Z
0	0	ALU3
0	1	ALU3 ÷ 2
1	0	ALU3 × 2
1	1	store Z

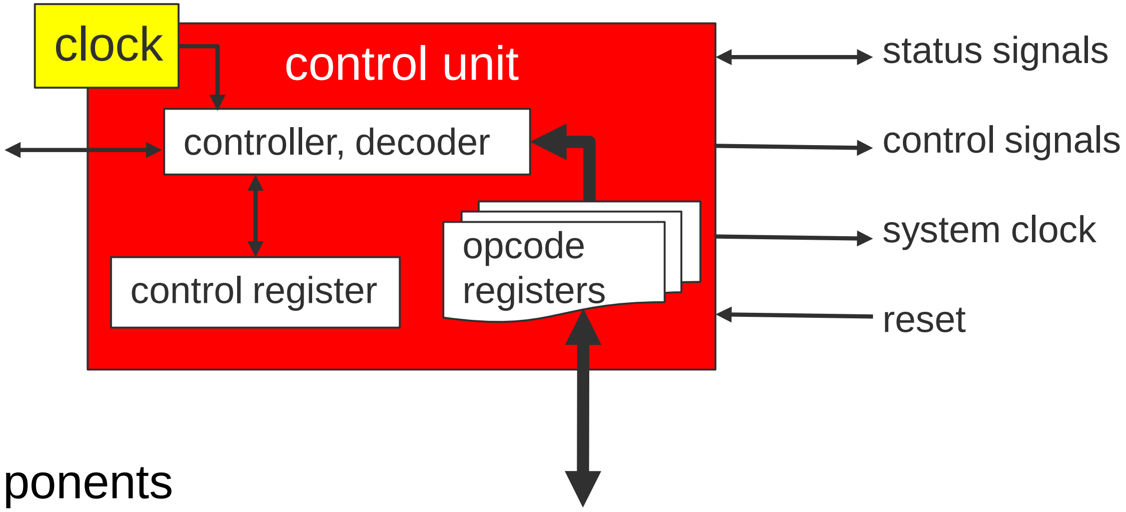
Internal architecture of a simple and simplified microprocessor





CONTROL UNIT

Overview



The control unit controls all the components

The **clock** generates the system clock for distribution to all components

Opcode registers contain the portion of the instruction that specifies the currently executed operation to be performed (and maybe some additional opcodes)

The **decoder** (often micro-programmable) generates all control signals for the components and uses status signals and opcode as input

The **control register** stores the current status of the control unit

Clocking / synchronization

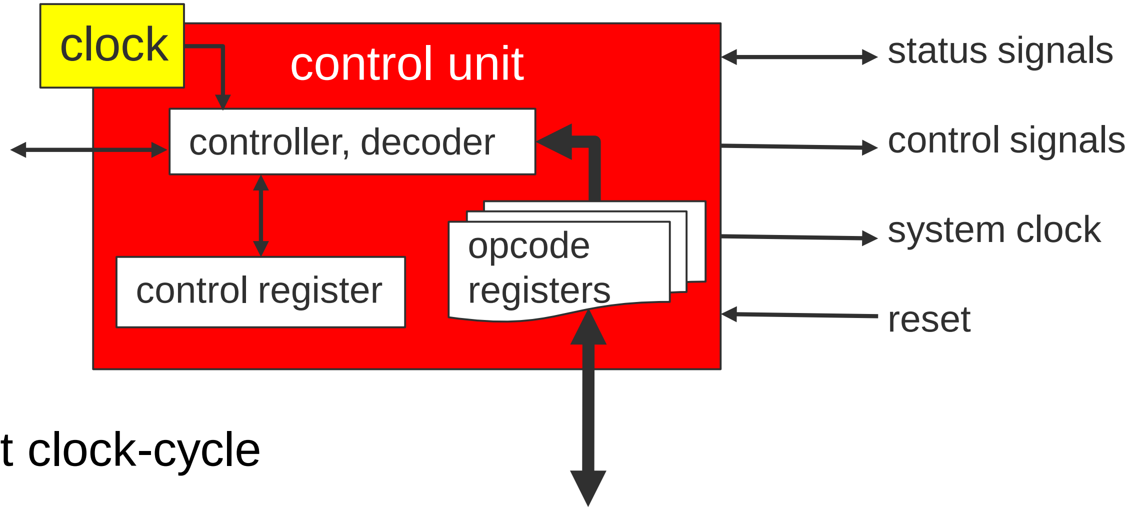
Synchronous sequential circuit

- Typically, CPUs use dynamic (clocked) logic
- State is stored in gate capacitances
- Static logic uses flip-flops instead

Minimum clock-speed required

- Otherwise, stored bits are lost due to leakage before the next clock-cycle

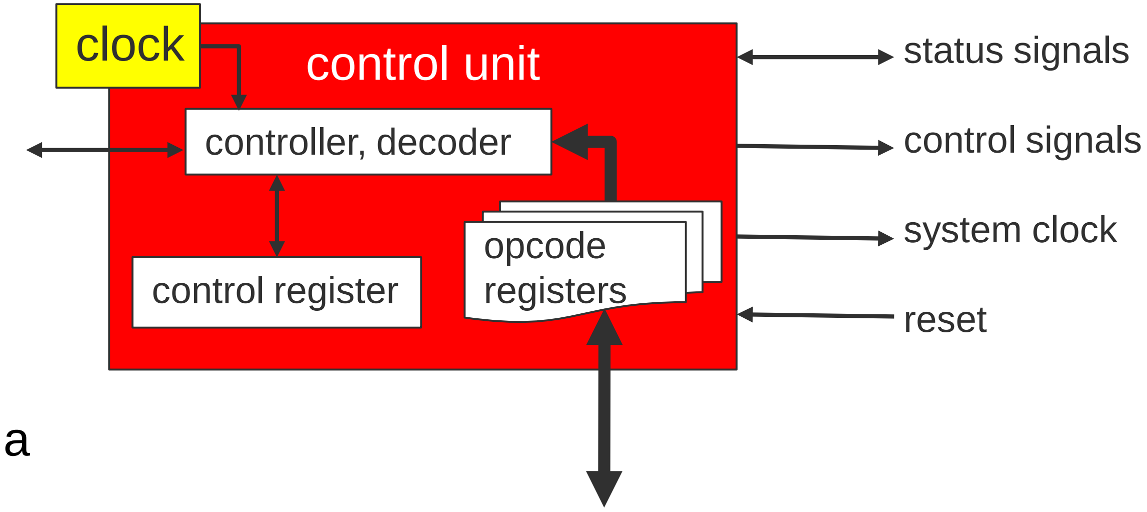
Complex clock distribution network on-chip required



Micro programmable control unit

The processor stores a microprogram for each instruction

- Microprogram: sequence of micro instructions
- Normal users cannot change the microprogram of a processor
- However, manufacturers can update the microprogram



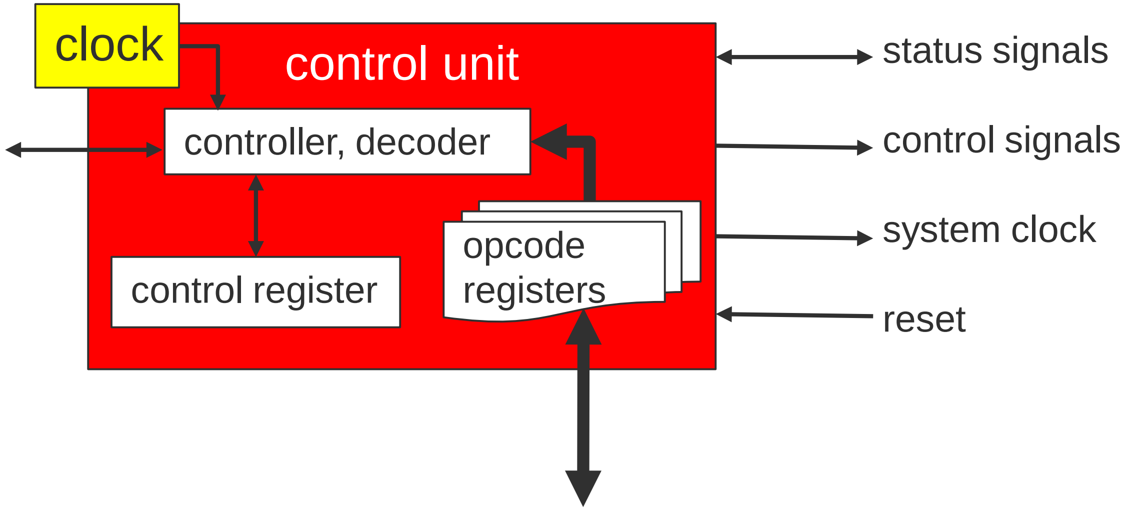
Pure RISC processors typically do not use microprograms but a fixed sequential circuit

Example micro instruction:

next address	registers	ALU operation	ALU operands	interface control	address generation	external control signals

Single bits of the micro instruction represent micro operations, thus a setting of the control signals for the components

Phases of instruction execution



Instruction fetch

- Load the next instruction into the opcode register

Instruction decode

- Get the start address of the microprogram representing the instruction

Execution

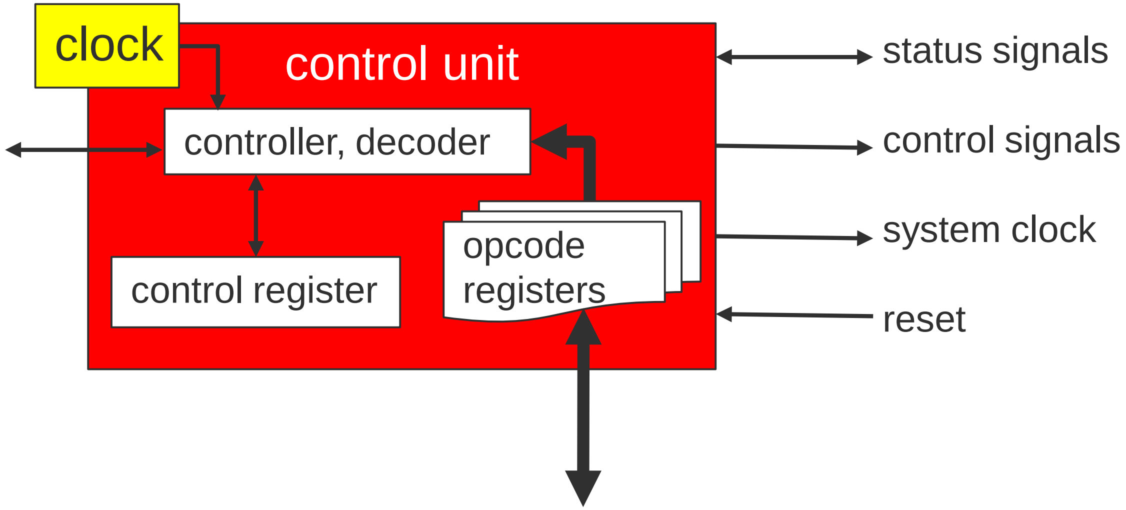
- The microprogram controls the instruction execution by sending the appropriate signals to the other components and evaluating the returned signals

Opcode register

The opcode register consists of several registers because

- different instructions may have different sizes
  (1 byte, 2 bytes, 3 bytes …)

- opcode prefetching may speed-up program execution
  - while decoding the current instruction the following instructions may be prefetched
  - this supports pipelining, branch prediction etc. (covered later)

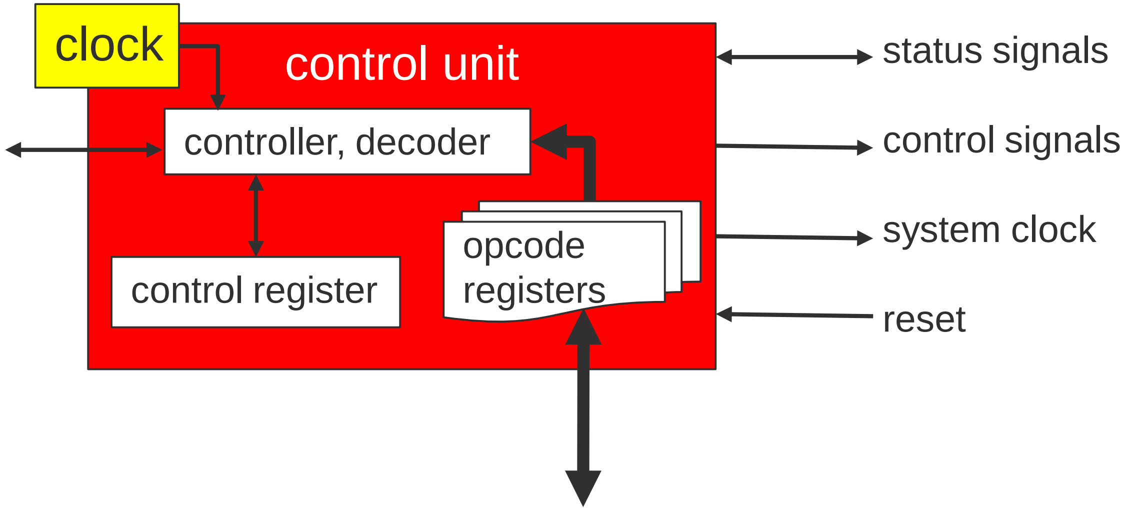


Control register

The control register stores the current state of the control unit.

This influences e.g. instruction decoding, operation mode.

The meaning of the bits depend on the processor.

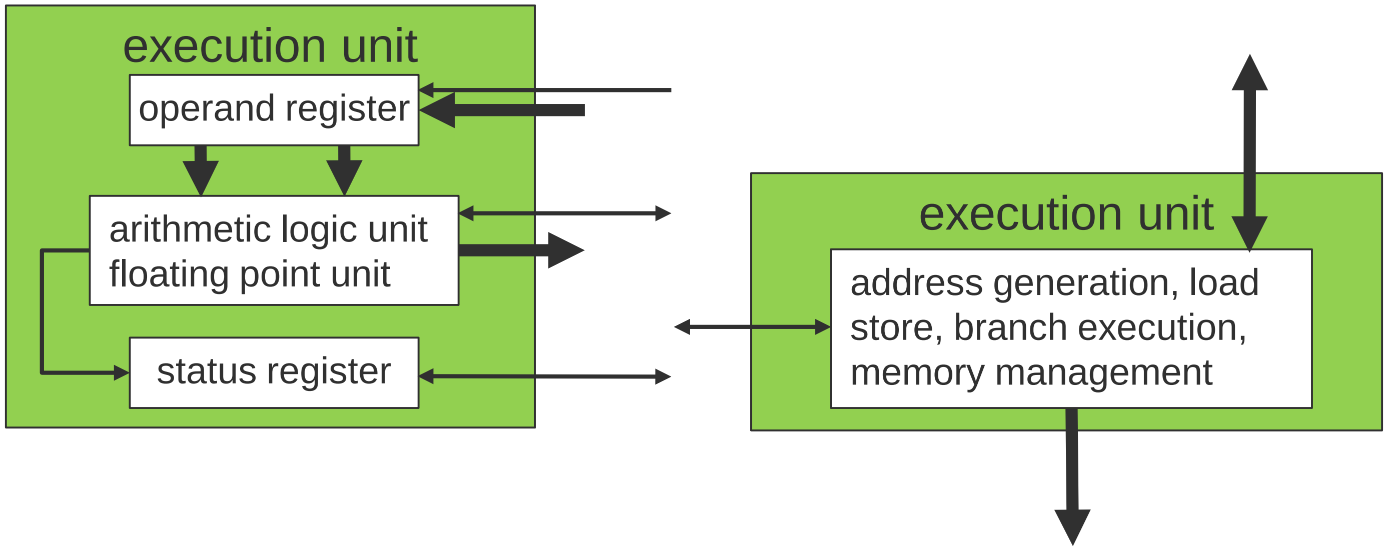


Examples:

- Interrupt enable bit
  - determines if the processor reacts to interrupts
- Virtual machine extensions enable
  - enable hardware assisted virtualization on x86 CPUs
- User mode instruction prevention
  - if set, certain instructions cannot be executed in user level
- see e.g. https://en.wikipedia.org/wiki/Control_register

Questions & Tasks

- Look at the internal architecture of a simple microprocessor. Where are potential performance bottlenecks?
- Who decides if data flows to an execution or control unit?
- What are advantages / disadvantages of micro programming?
- Name examples of control and status signals to / from the environment!
- Why do we need a reset?
- What limits the clock frequency (min/max)?



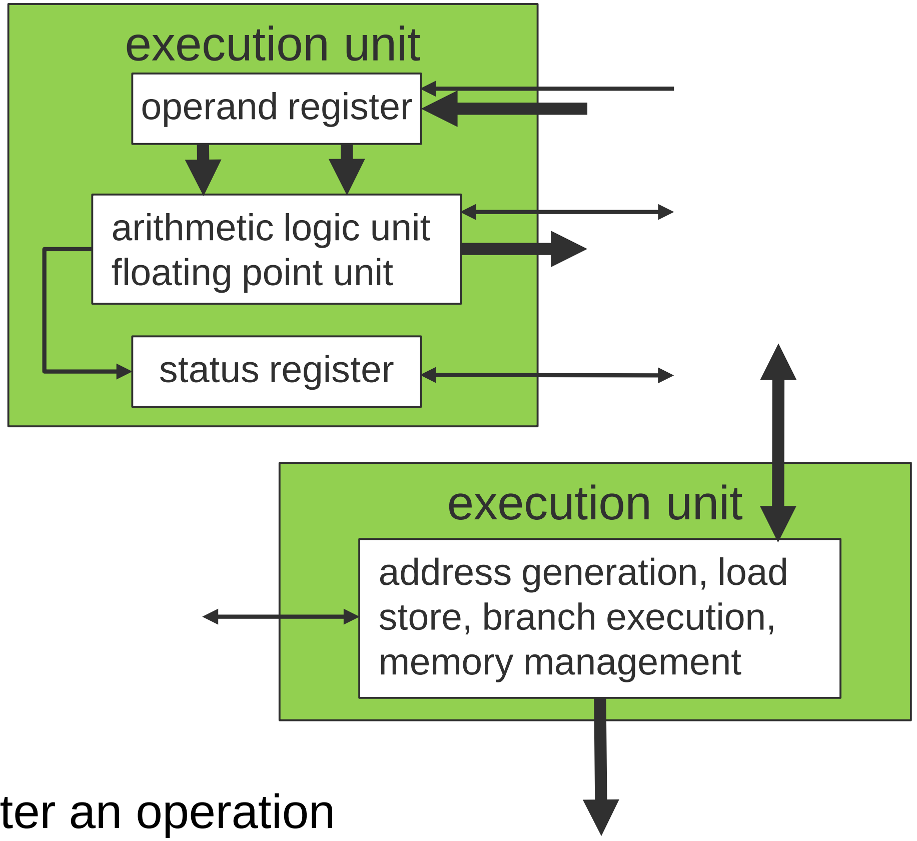
EXECUTION UNIT

Overview

The execution unit executes all logic and arithmetic operations controlled by the control unit.

Examples:

- Integer and float arithmetic operations
- Logic operations, shifting, comparisons
- All address related operations
- Speculative operations (covered later)
- Complex memory management, memory protection
- …



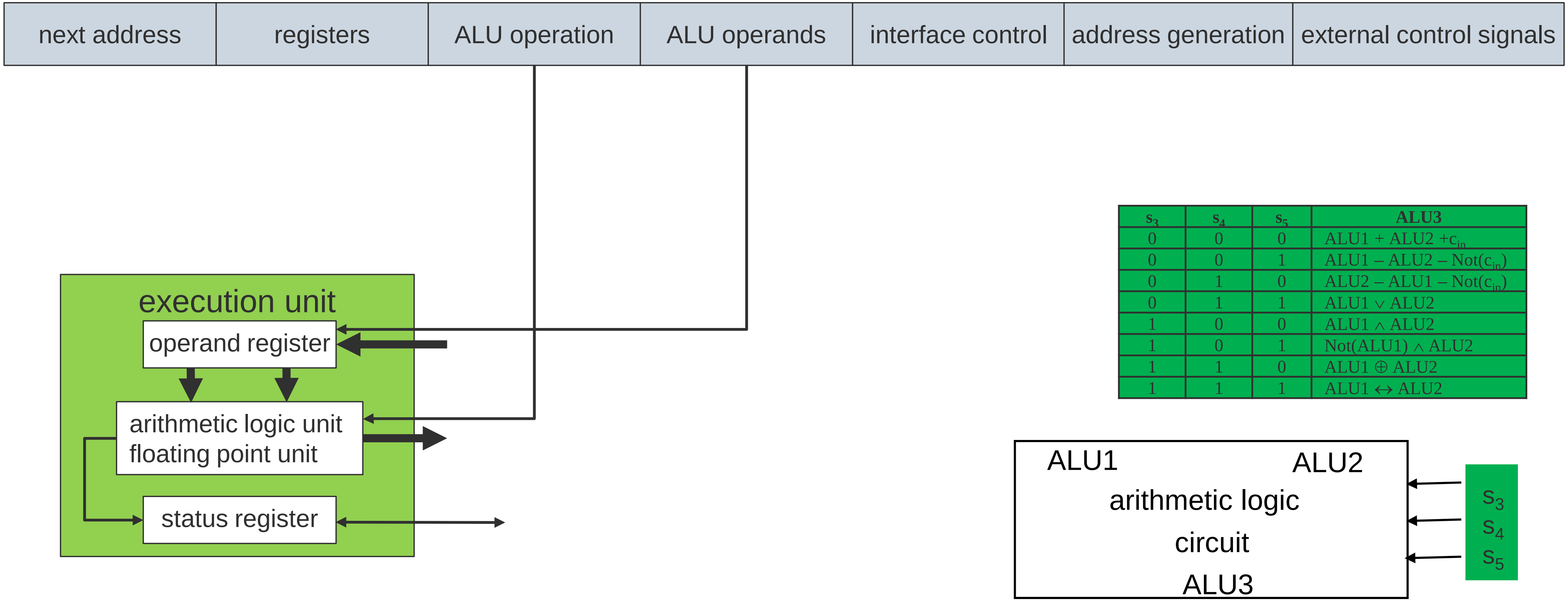
Status register informs the control unit about the state of the processor after an operation

- Examples: carry, overflow, zero, sign

Operand registers, accumulators etc.: additional registers for temporary results, fetched operators etc.

Connection to the control unit

Single bits of a micro instruction directly control e.g. ALU and operand register



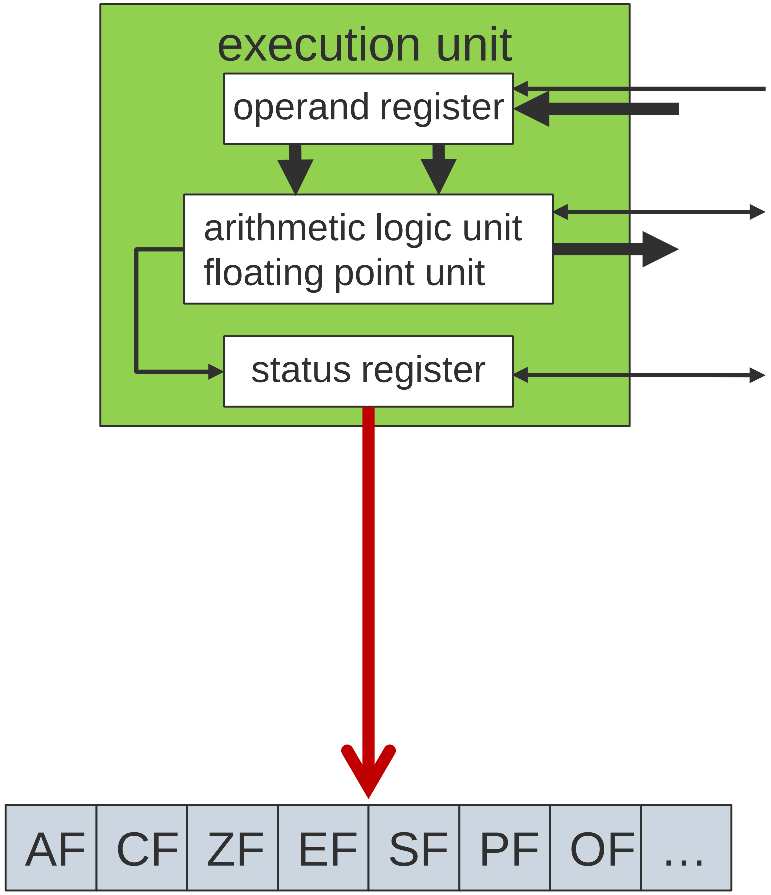
s_3	s_4	s_5	ALU3
0	0	0	ALU1 + ALU2 +c_{in}
0	0	1	ALU1 – ALU2 – Not(c_{in})
0	1	0	ALU2 – ALU1 – Not(c_{in})
0	1	1	ALU1 ∨ ALU2
1	0	0	ALU1 ∧ ALU2
1	0	1	Not(ALU1) ∧ ALU2
1	1	0	ALU1 ⊕ ALU2
1	1	1	ALU1 ↔ ALU2

Status register (flag register, Condition Code Register CCR)

Single bits representing the state of the processor after an operation are stored in the status register.

Common bits in the status register (often called flags):

- Auxiliary Carry, AF
- Carry Flag, CF
- Zero Flag, ZF
- Even Flag, EF
- Sign Flag, SF
- Parity Flag, PF
- Overflow Flag, OF
- …



AF	CF	ZF	EF	SF	PF	OF	…

Description of the status flags 1

Auxiliary Carry (AF)

- Indicates a carry between the nibbles (4 bit halves of a byte)
- Used for BCD (binary coded digit) arithmetic
- Also called half-carry flag, digit carry, decimal adjust flag

Carry Flag (CF)

- Indicates a carry produced by the MSBs
- Allows for addition/subtraction of numbers larger than a single word by sequential additions/subtractions taking the carry into account

Zero Flag (ZF)

- Indicates that the result of an operation was zero
- Used for conditional branches or loops (e.g. `if x=y then…` is translated into `SUB x,y,z; BZ…`)

Description of the status flags 2

Even Flag (EV)

- Indicates if the result is even or odd (LSB)

Sign Flag (SF)

- Indicates if the result is negative (MSB = 1) in two's complement
- Used e.g. for conditional branches (if `x > y then …` is translated into `SUB y,x,z; BNP…`)

Parity Flag (PF)

- Indicates if the number of set bits is even or odd
- Used e.g. for error detection

Overflow Flag (OF)

- Indicates that the result of an operation is too large to be represented (e.g. during addition or subtraction)

Program Status Word (PSW)

Status register plus control register determine the current state of a processor

- Result of an operation
- Privilege level
- …

Together with the program counter (address of the current or next instruction) these registers determine the state of the processor at a certain instruction of a program (or process, task, …).

The PSW combines the registers and program counter for simpler manipulation.

- Pushed to stack before context switch (e.g. switch to another process)
- Pulled from stack to continue execution of an interrupted process

Different names and semantics depending on processor architecture…

Typical (simple) operations of an ALU

Arithmetic

- Addition with/without carry
- Subtraction with/without carry
- Increment/decrement
- Multiplication with/without sign
- Division with/without sign
- Two’s complement

Logical

- NOT
- AND
- OR
- XOR

Shift and rotation

- Shift left
- Shift right
- Rotate right without carry
- Rotate right with carry
- Rotate left without carry
- Rotate left with carry

Memory

- Transfer
- Load, store

Questions & Tasks

- What do we need a status register for?
- What is the idea behind the PSW? When and why do we have to save it?
- How can we add very, very long integers (that do not fit into the operand registers of the ALU)?
- When do we typically use the flags?
- How do we know when the ALU finished its calculations?

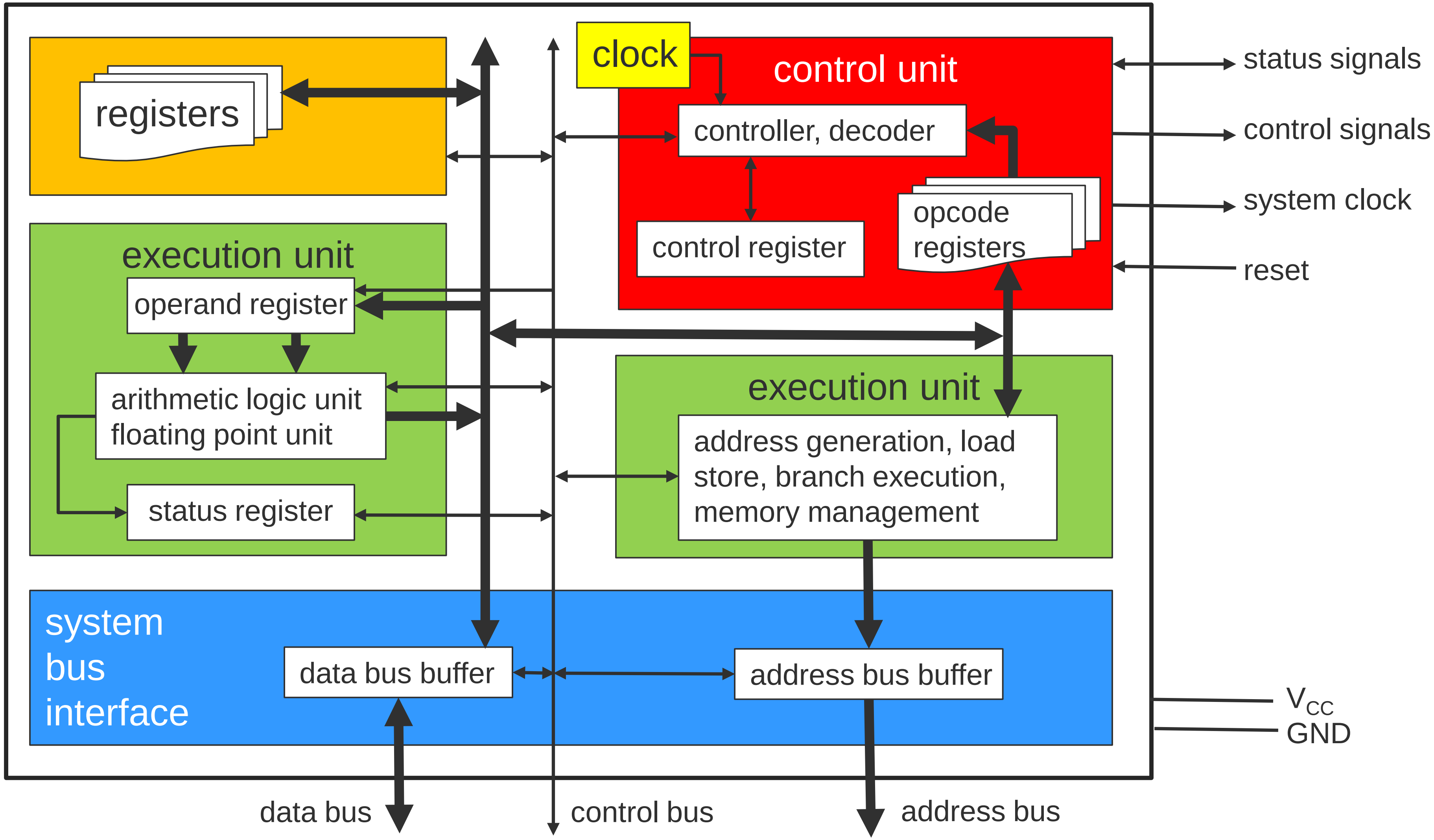
REGISTERS

Registers

Extend the registers of the execution unit

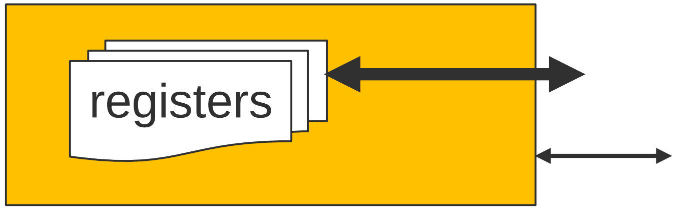
Storage of frequently used operands

Much faster than the main storage



Registers

Very fast memory with very low access time (< 1 ns)



Direct selection of single registers via dedicated control lines

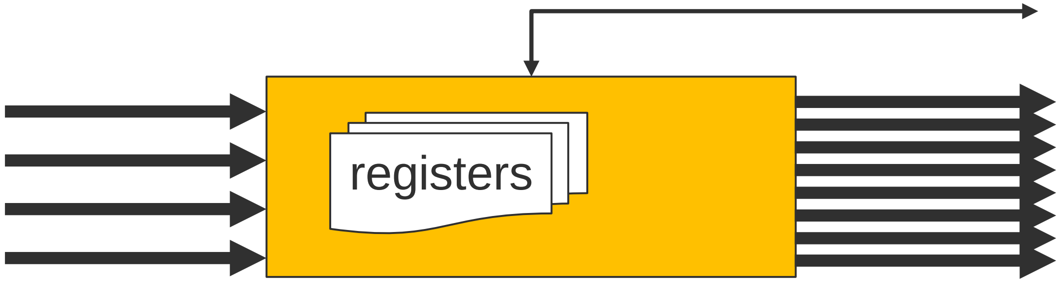
- No address decoder / decoding necessary

All registers are on-chip

- No external access necessary with delays due to run-time, multiplexing, buffering etc.

Can offer additional functions

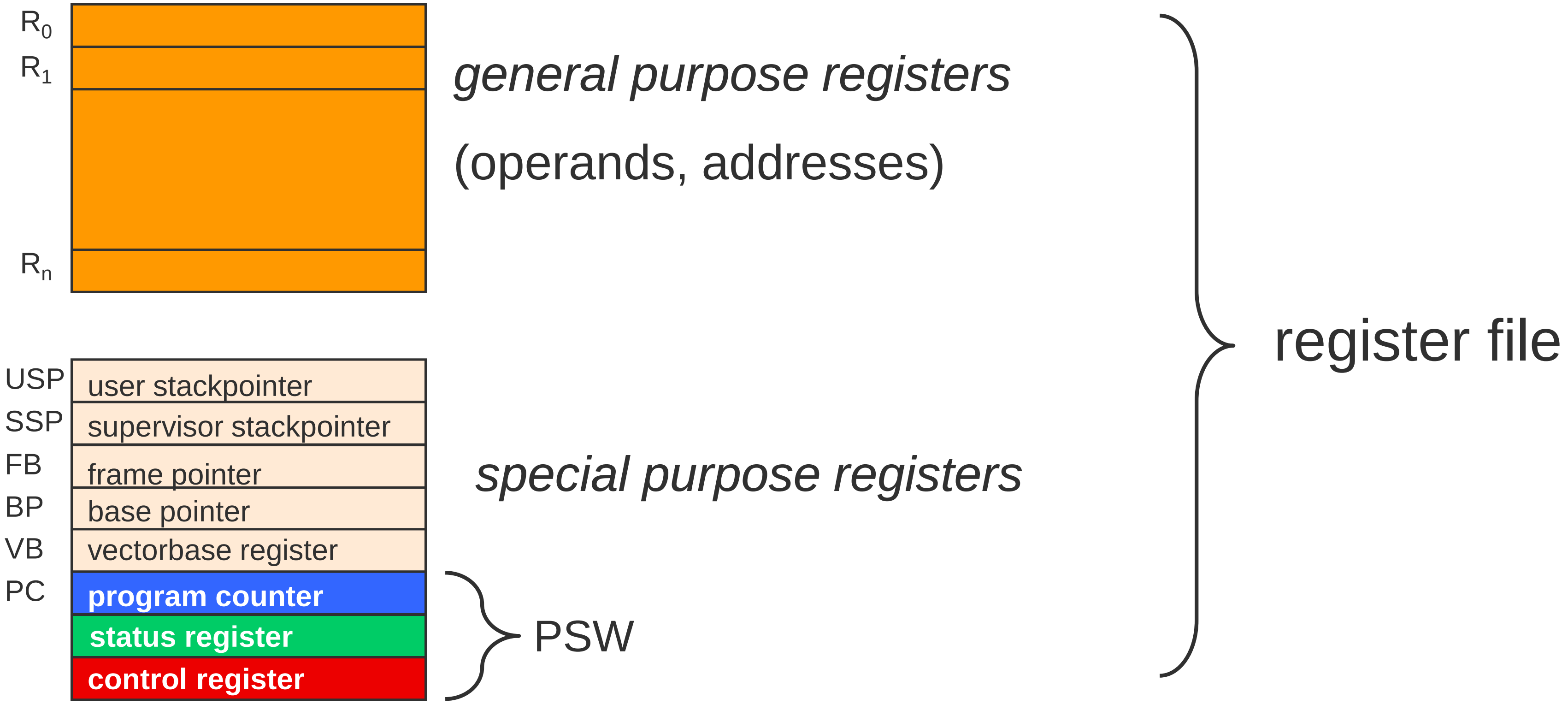
- Increment/decrement
- Shift
- Set to zero, hardwired to zero



Several independent input/output port

- Simultaneous writing and reading of several (different) registers possible
- Today’s superscalar processors are able to write 4 registers and read 8 registers in one clock cycle

Register file – example



Register file – example: Intel 8086 (first generation personal computers)

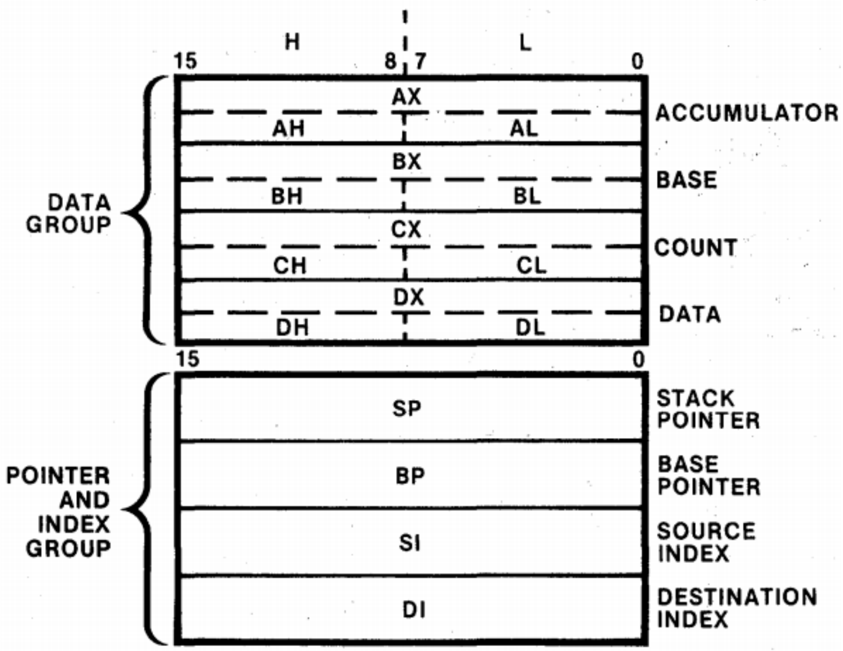


Figure 2-7. General Registers

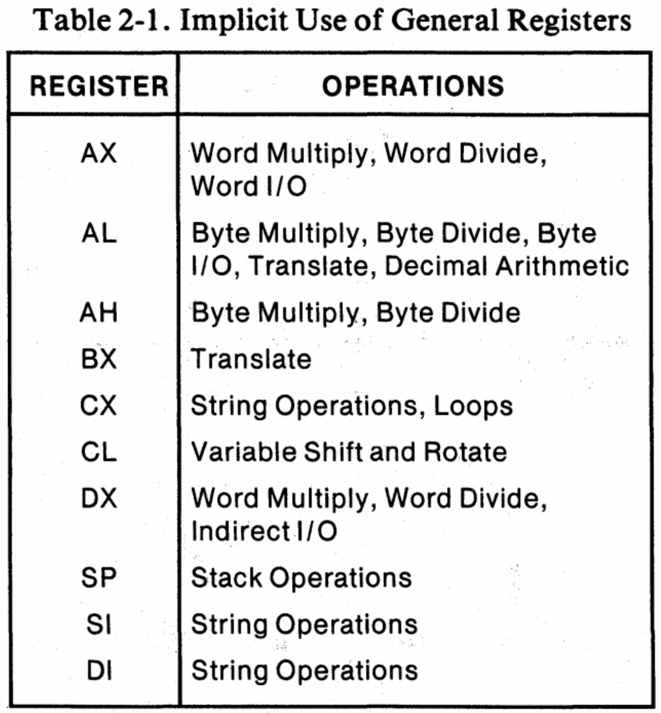
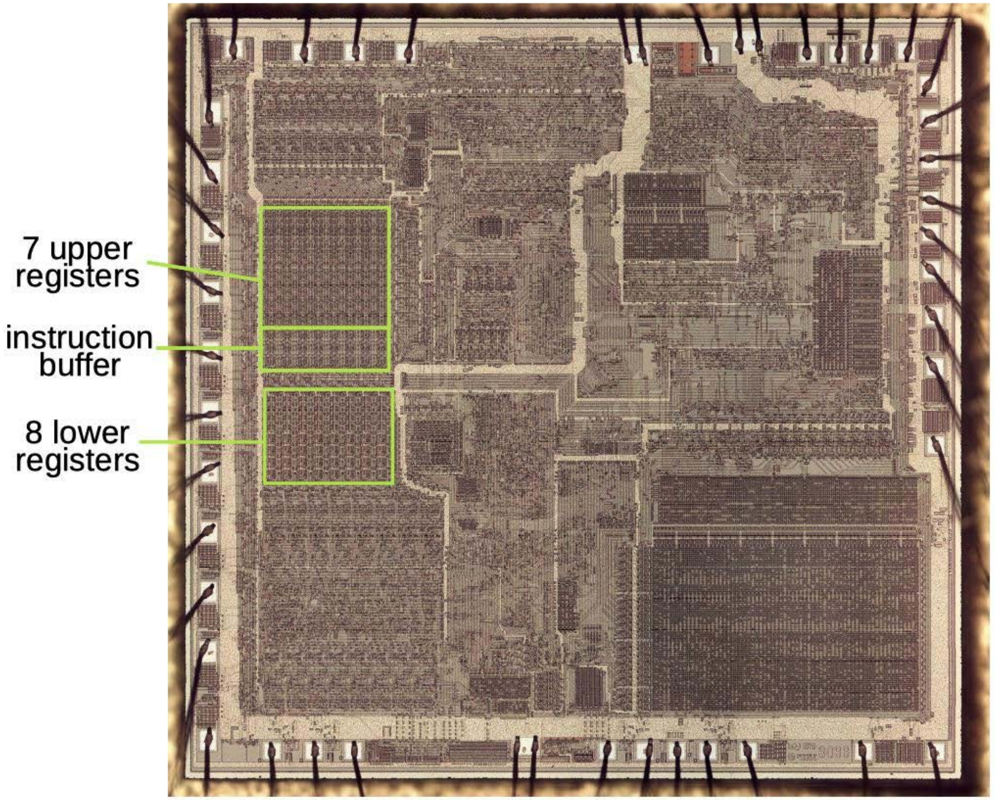


Table 2-1. Implicit Use of General Registers

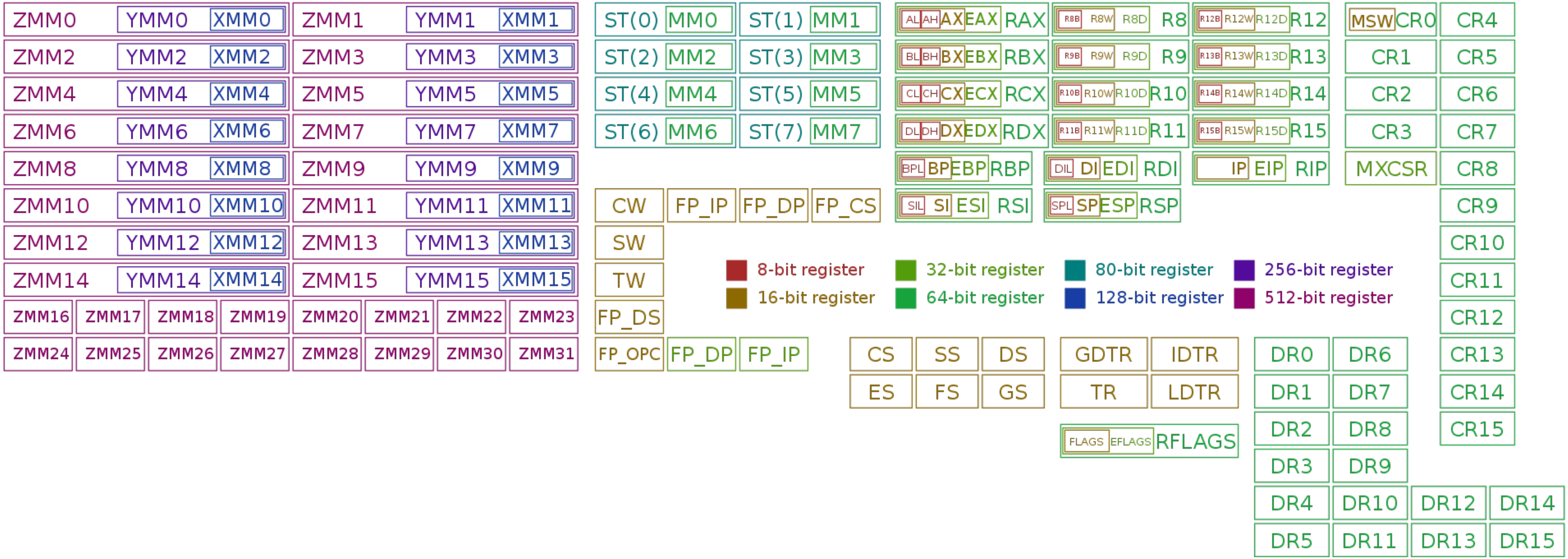
REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The 8086 Family User‘s Manual, Intel, October 1979



http://www.righto.com/2020/07/the-intel-8086-processors-registers.html

Register file – example: x86-64 (current Intel/AMD processors)



ZMM0 YMM0 XMM0 | ZMM1 YMM1 XMM1 | ST(0) MM0 | ST(1) MM1 | AL AH AX EAX RAX | R8B R8W R8D R8 | R12B R12W R12D R12 | MSW CR0 | CR4

ZMM2 YMM2 XMM2 | ZMM3 YMM3 XMM3 | ST(2) MM2 | ST(3) MM3 | BL BH BX EBX RBX | R9B R9W R9D R9 | R13B R13W R13D R13 | CR1 | CR5

ZMM4 YMM4 XMM4 | ZMM5 YMM5 XMM5 | ST(4) MM4 | ST(5) MM5 | CL CH CX ECX RCX | R10B R10W R10D R10 | R14B R14W R14D R14 | CR2 | CR6

ZMM6 YMM6 XMM6 | ZMM7 YMM7 XMM7 | ST(6) MM6 | ST(7) MM7 | DL DH DX EDX RDX | R11B R11W R11D R11 | R15B R15W R15D R15 | CR3 | CR7

ZMM8 YMM8 XMM8 | ZMM9 YMM9 XMM9 | BPL BP EBP RBP | DIL DI EDI RDI | IP EIP RIP | MXCSR | CR8

ZMM10 YMM10 XMM10 | ZMM11 YMM11 XMM11 | CW | FP_IP | FP_DP | FP_CS | SIL SI ESI RSI | SPL SP ESP RSP | CR9

ZMM12 YMM12 XMM12 | ZMM13 YMM13 XMM13 | SW | CR10

ZMM14 YMM14 XMM14 | ZMM15 YMM15 XMM15 | TW | CR11

ZMM16 | ZMM17 | ZMM18 | ZMM19 | ZMM20 | ZMM21 | ZMM22 | ZMM23 | FP_DS | CR12

ZMM24 | ZMM25 | ZMM26 | ZMM27 | ZMM28 | ZMM29 | ZMM30 | ZMM31 | FP_OPC | FP_DP | FP_IP | CS | SS | DS | GDTR | IDTR | DR0 | DR6 | CR13

ES | FS | GS | TR | LDTR | DR1 | DR7 | CR14

DR2 | DR8 | CR15

FLAGS EFLAGS RFLAGS

DR3 | DR9

DR4 | DR10 | DR12 | DR14

DR5 | DR11 | DR13 | DR15

🟥 8-bit register 🟩 32-bit register 🟦 80-bit register 🟪 256-bit register

🟫 16-bit register 🟩 64-bit register 🟦 128-bit register 🟪 512-bit register

https://en.wikipedia.org/wiki/X86

Special registers: base pointer and index

The base pointer contains the start address of a memory region

- The memory could e.g. represent an array

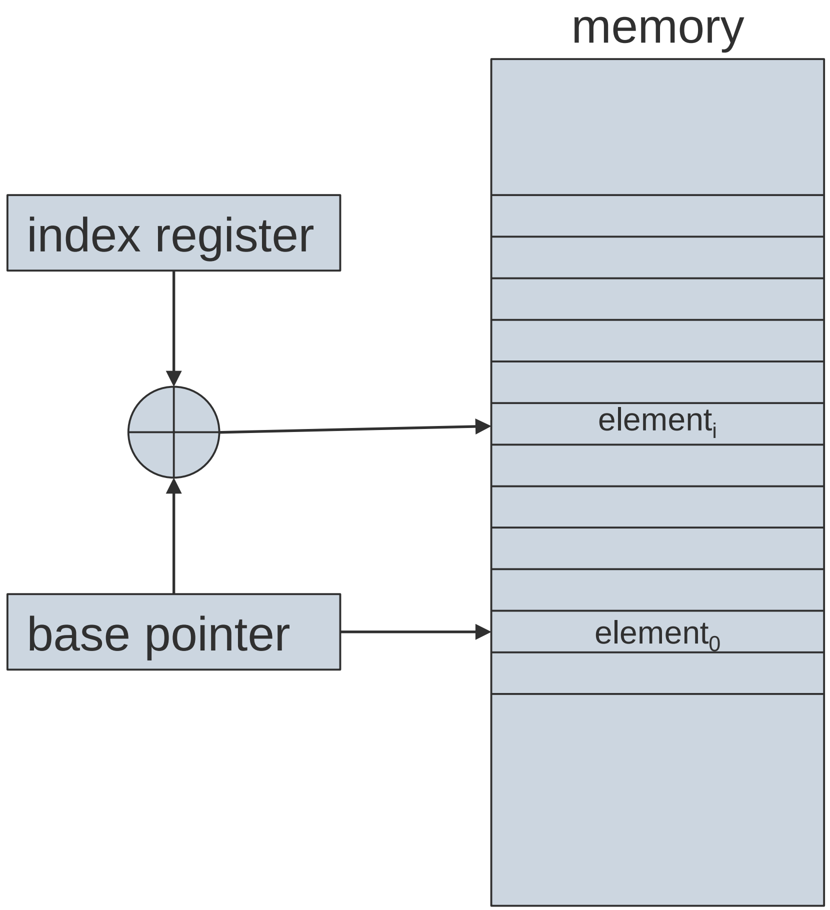
The index represents the offset relative to the base pointer

- Selection of a single element e.g. of an array

Base pointer + index gives the absolute address of an element

Programming can be done relative to the base pointer

- This allows for e.g. moving or copying the array without changing the relative addressing



Special functions of index registers

Post-increment

- Automatic increment of the register by n after addressing the memory

Pre-decrement

- Automatic decrement of the register by n before addressing the memory

Auto-increment / auto-decrement

Auto-scaling by factor n (1, 2, 4, 8 etc.)

- Used to access memory in bytes/words/…

Saves time as the ALU is freed from this additional operation when using the index!

The (runtime-) stack

Part of the memory (typically in the main memory) that is organized as stack following the LIFO (Last-In-First-Out) principle.

Purpose

- Stores the PSW (status of processor, program counter) during subroutine call / interrupt processing
- Parameter passing
- Storage of temporary results

Processors often come with several stacks for different purposes: system stack, user stack, data stack, …

Hardware support

- special register: stack pointer – address of the newest data on the stack

Special instructions to transfer data to/from the stack

- PUSH: transfer the value of a register on top of the stack
- POP (PULL): load a register with the value on top of the stack and remove this element

Management of a stack pointer

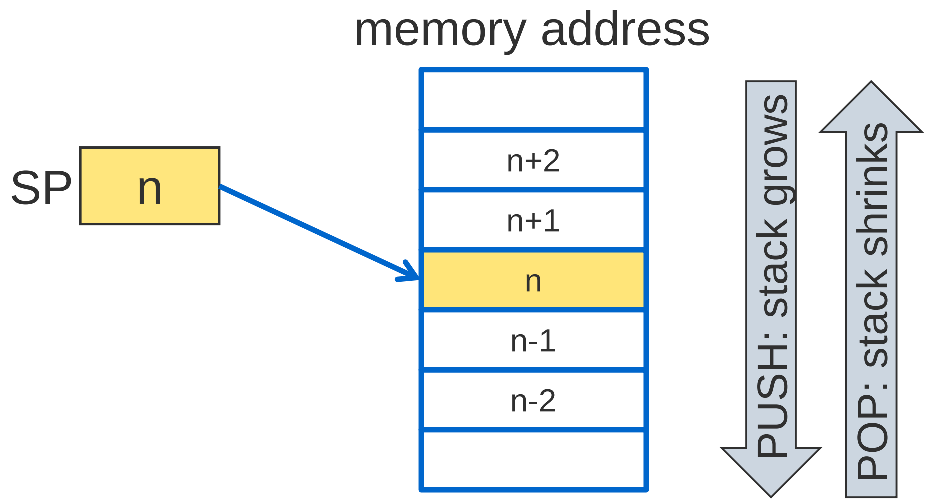
The stack pointer always points to the address of the element on top of the stack.

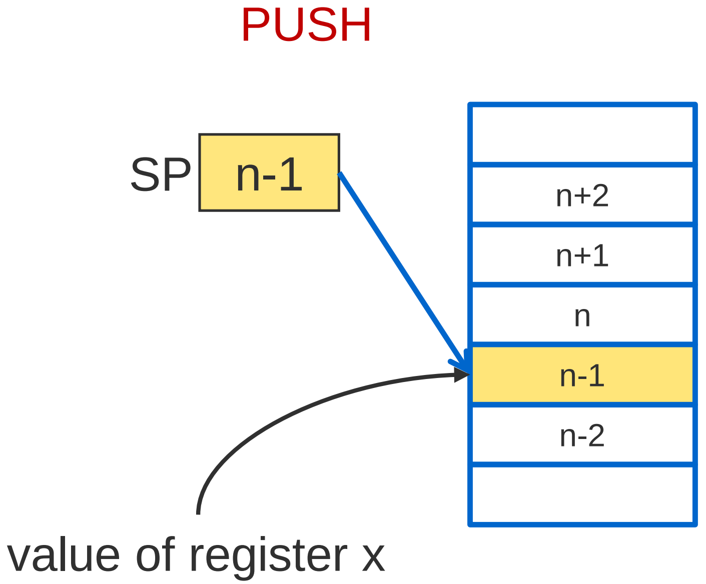
Stacks quite often grow from “top-to-bottom”, i.e., from higher to lower addresses.

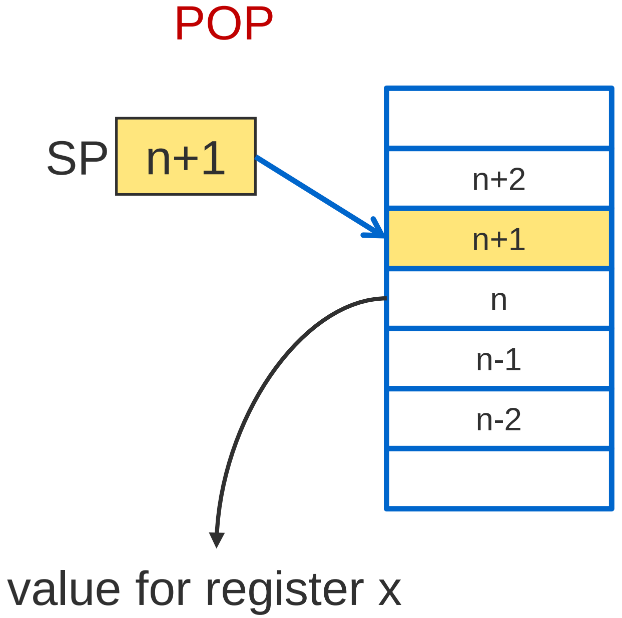
PUSH x: decrement the stack pointer, then load the value of register x on top of the stack (i.e. into the memory at the address the stack pointer points to)

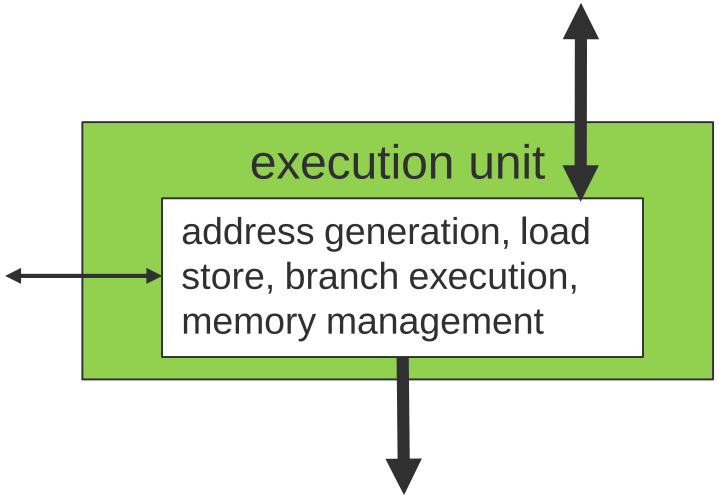
POP x: load the value stored at the address the stack pointer points to into register x, then increment the stack pointer (i.e. reduce the size of the stack by one element)

Here we can use index registers with pre-decrement and post-increment.








execution unit
address generation, load store, branch execution, memory management

ADDRESS GENERATION UNIT

Address generation unit

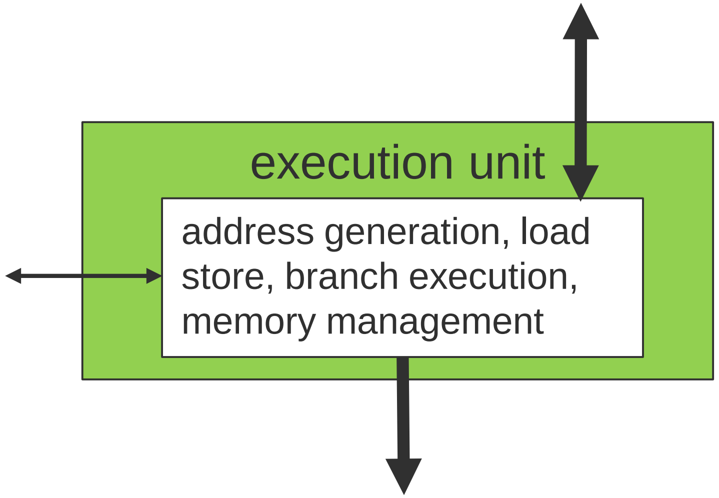
Specialized part of the execution unit

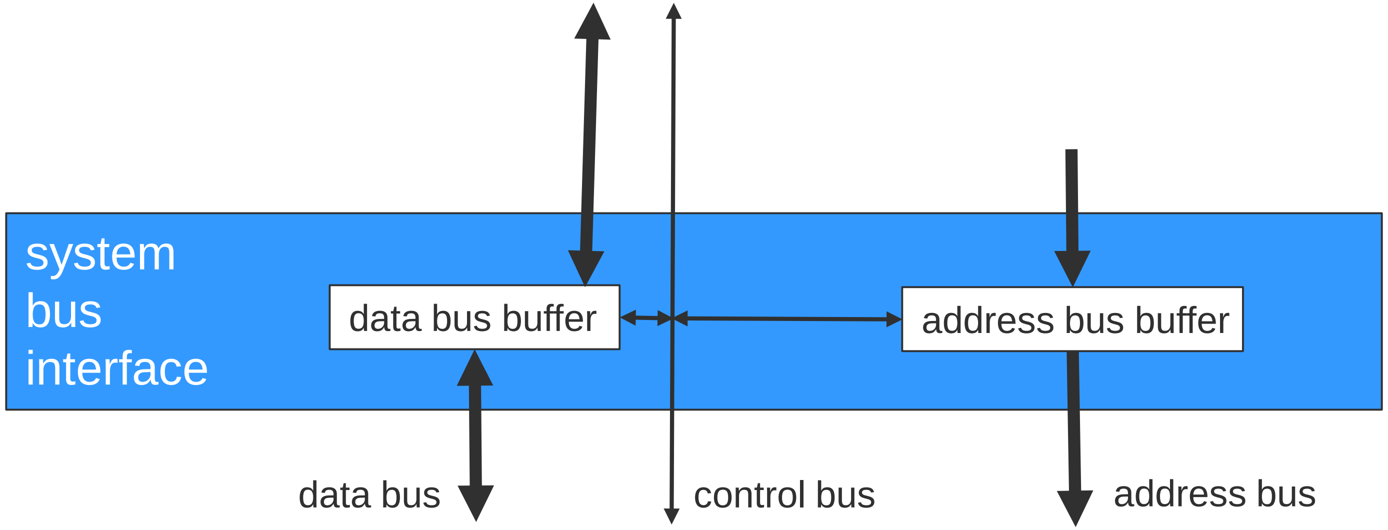
Basic operation: calculate an address based on control signals from the control unit and possibly additional content of registers

- e.g. base pointer + index = address

Can be very simple, but also very complex

- MMU (memory management unit)
- many different modes, memory protection, virtual address space
- cache optimization, branch prediction, speculative loading etc.
- covered later!


execution unit
address generation, load store, branch execution, memory management



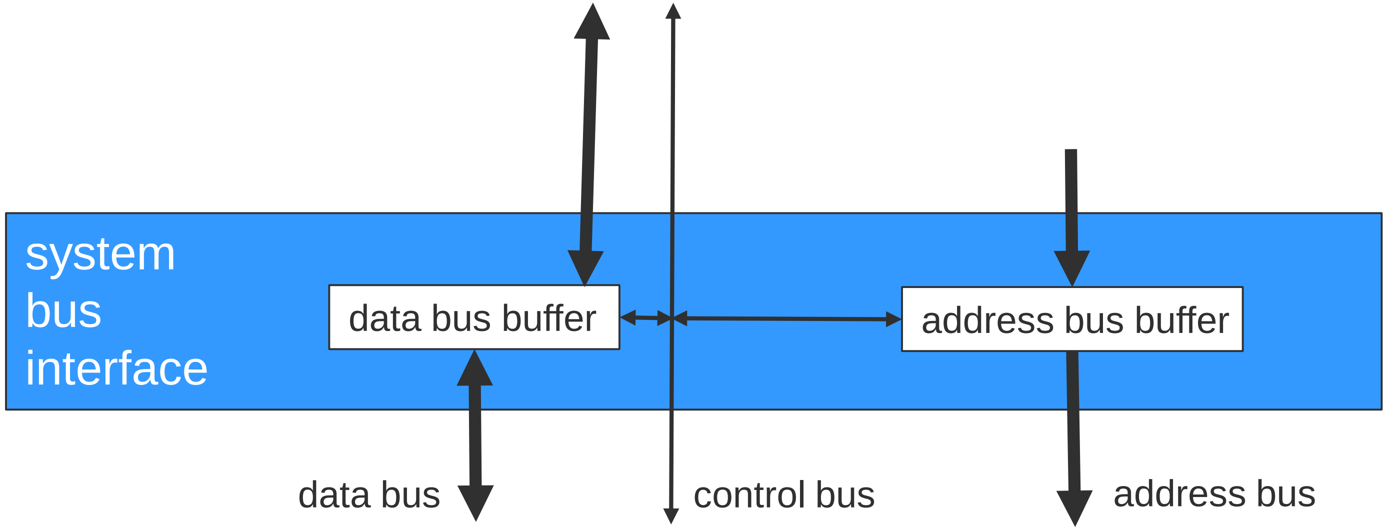
SYSTEM BUS INTERFACE

System bus interface

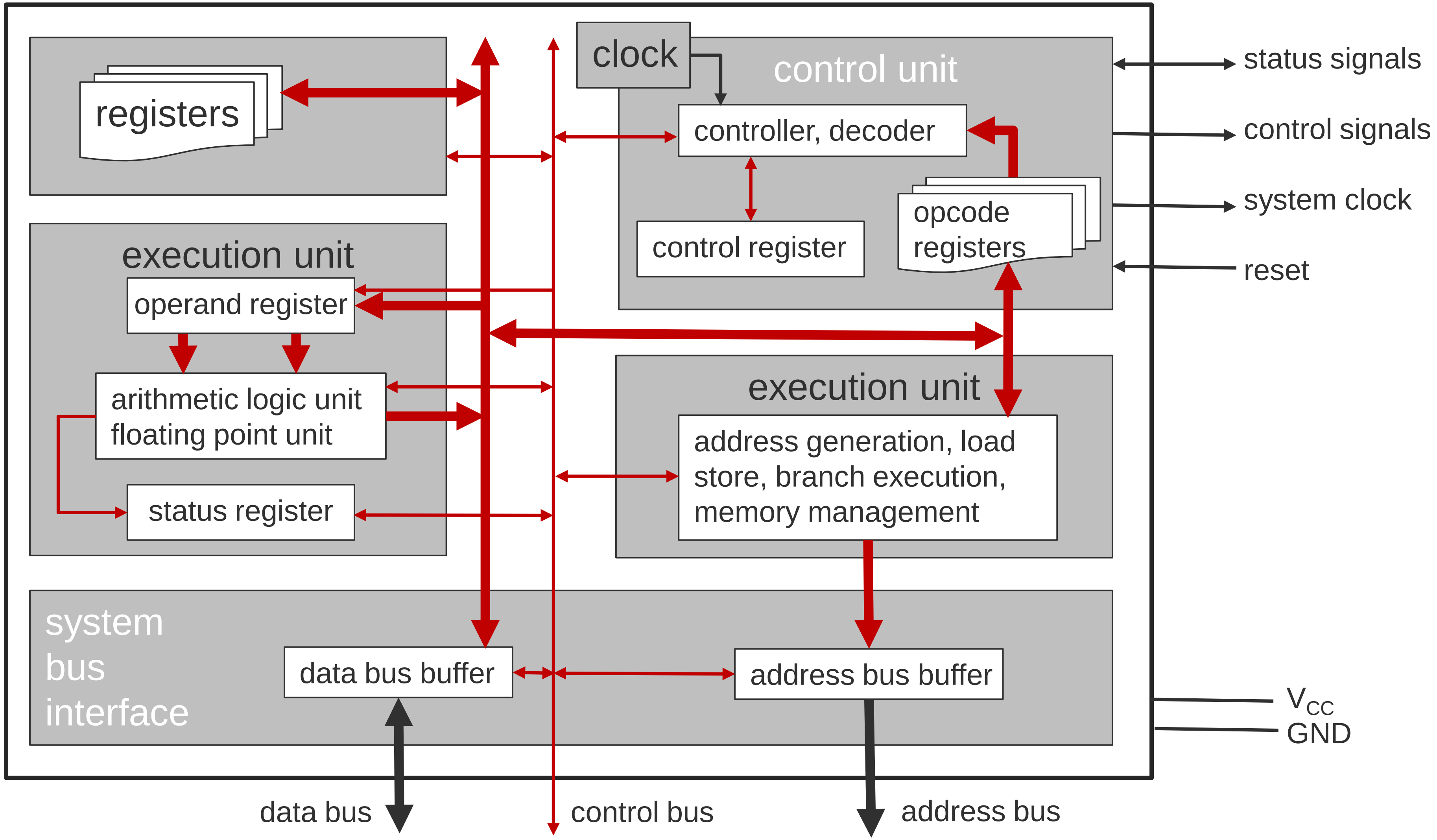
The system bus interface (Bus Interface Unit, BIU) is the connection of the microprocessor to its environment (all the other components of a micro computer)

Purpose

- Buffering of addresses and data (operands and instructions)
- Adaptation of clock cycles, bus width, voltages
- Tristate: detaching the processor from the external bus



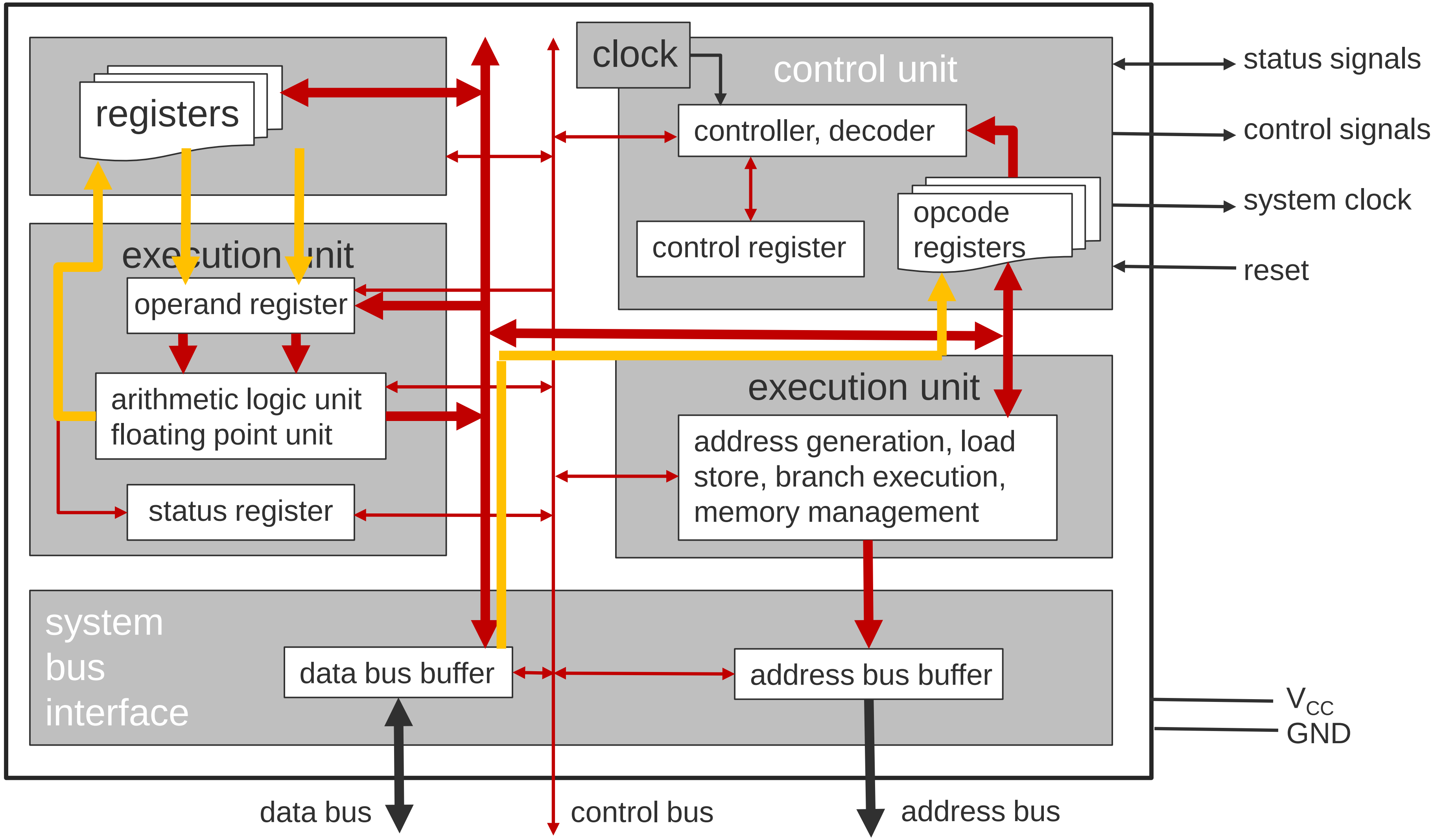
Internal bus system



Internal bus system - optimized

Example

- prefetch Bus
- two operand buses
- result bus



Additional components of a microprocessor

Current microprocessors can comprise:

- Cache memory (fast memory for instructions and operands, covered later)

- Vector processing unit

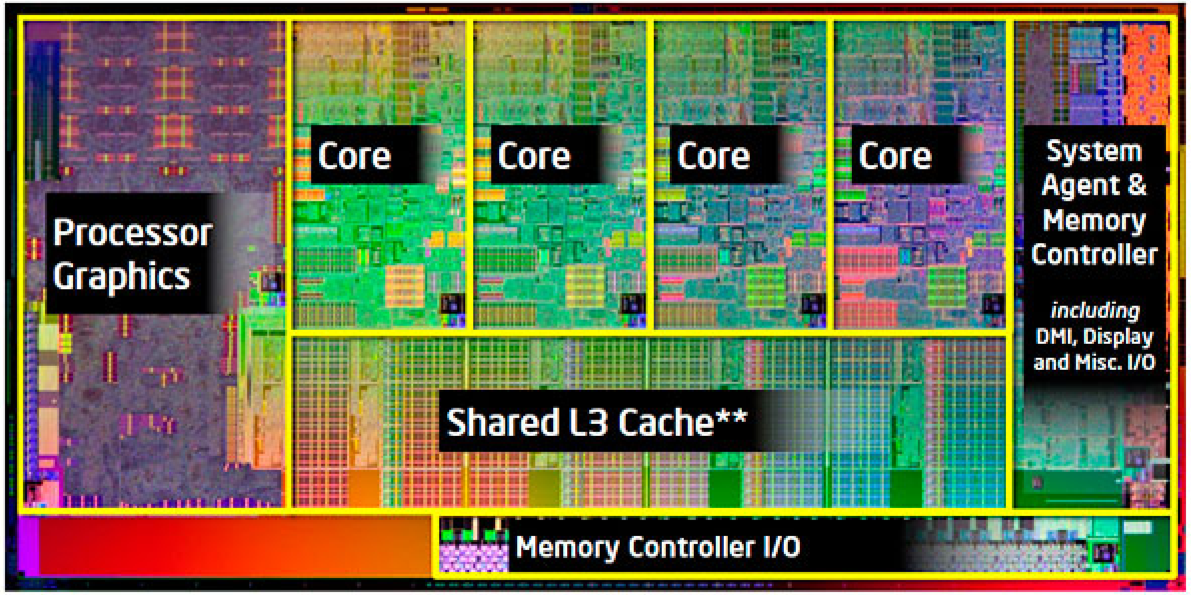
- Graphics processor

- Signal processing unit

- Neural networks, AI support

- Interrupt controller

- ...



Questions & Tasks

- Why are registers faster than the main memory?
- How can special purpose registers speed-up the processor?
- What is the purpose of a stack? Why not using a “normal” memory?
- What is the purpose of a base pointer + index? Can’t we directly address elements?
- How can we speed-up the internal processing in micro processors?
- Check the layout of current processors, GPUs, AI processors etc.! Can you find some of the components?

PERFORMANCE ENHANCEMENT

Performance enhancements in computer systems

How to enhance the performance of a computer system?

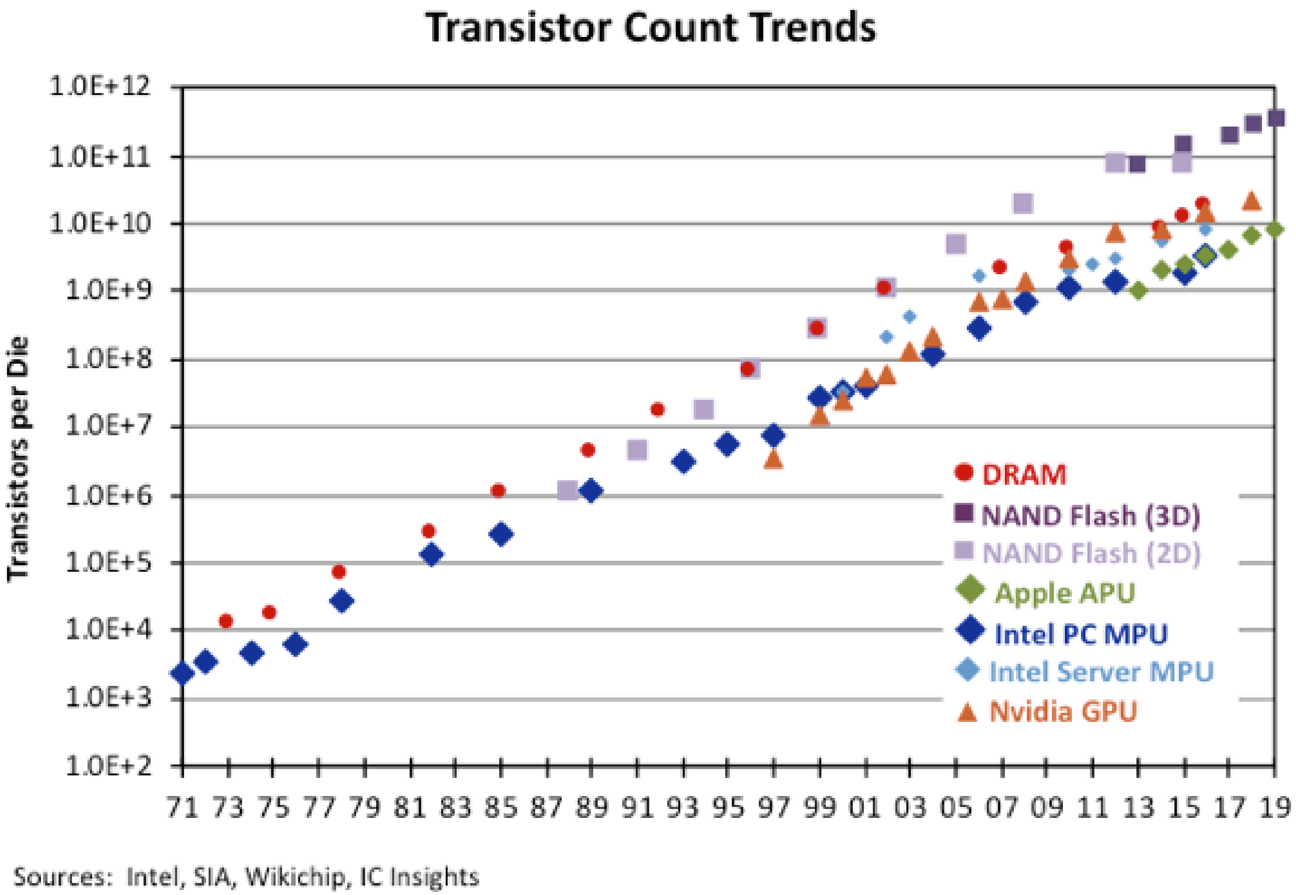
Technology

- Faster technologies, new materials, higher clock frequencies often require redesigns
- Expensive plus physical limitations

Architecture

- Increase parallelism (increased number of transistors, larger bus widths, replicated functional units)
- Multi-core processors go this way in basically all computer systems

Technological progress



Sources: Intel, SIA, Wikichip, IC Insights

https://www.semiconductor-digest.com/2020/03/10/transistor-count-trends-continue-to-track-with-moores-law/
https://en.wikipedia.org/wiki/Moore%27s_law

Performance enhancement

STRUCTURAL ENHANCEMENTS

Structural enhancements

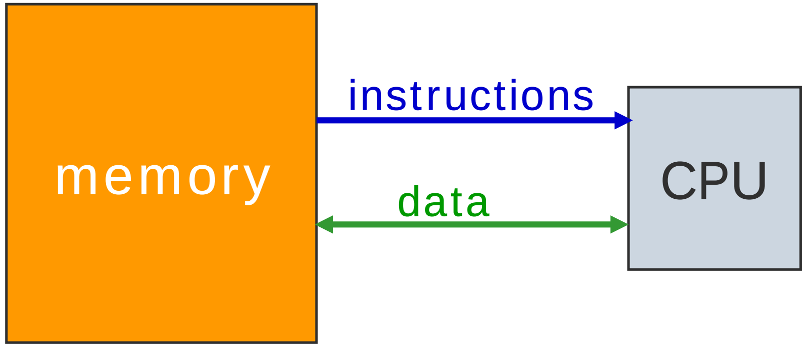
Classification of computer architectures according to Flynn

Be aware: historical classification, does not really fit anymore…

Considers parallelism in instructions and data (operands)

SISD (Single Instruction Single Data)

- A single serial stream of instructions operates on data (classical von-Neumann principle)

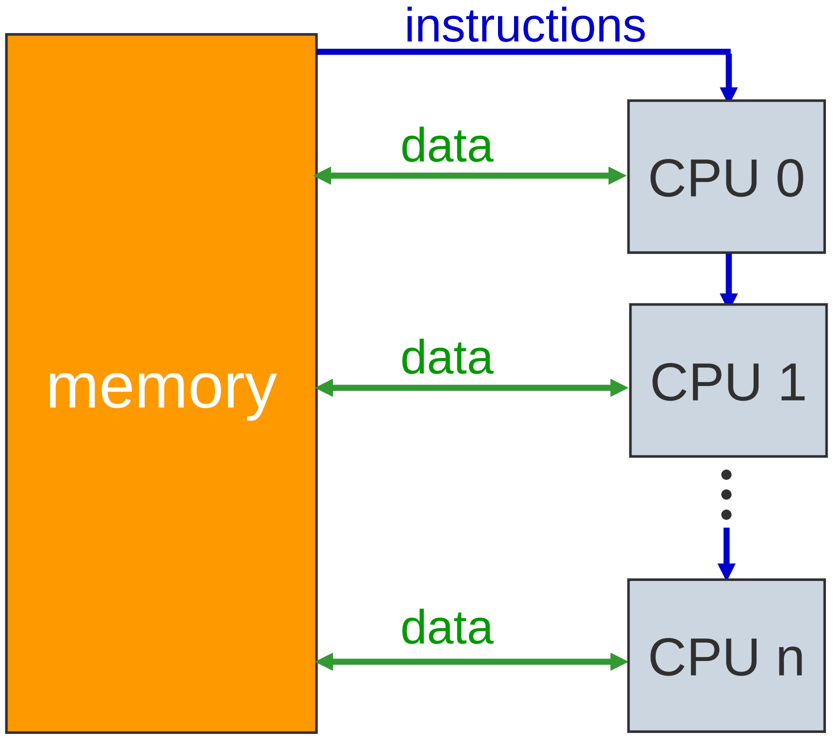


Classical:
IBM-PC, IBM 370,
DEC Micro-VAX,…

Structural enhancements

SIMD (Single Instruction Multiple Data)

- All processors perform the same instructions on different data (array processor)

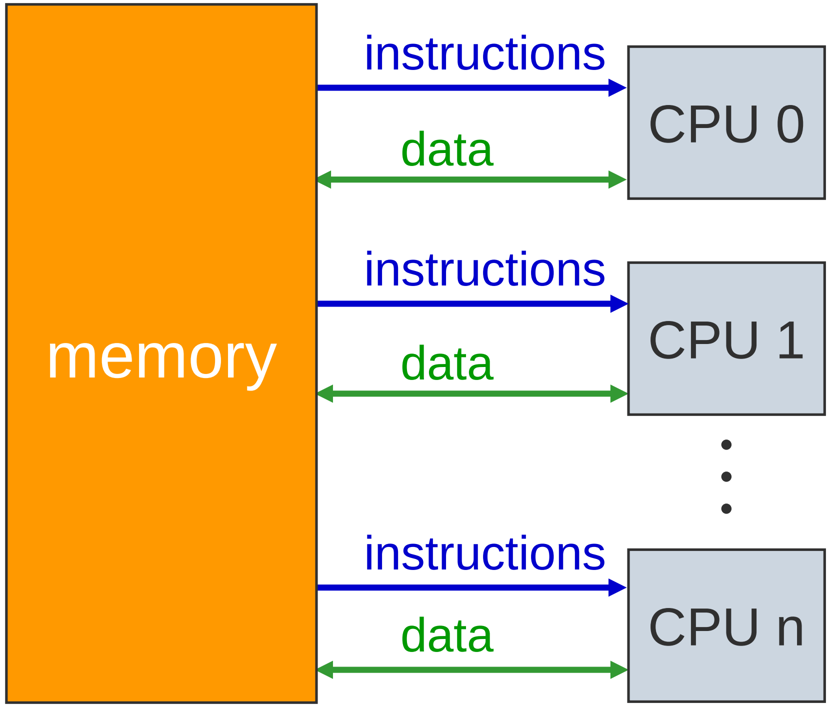


Example image processing: each processor operates on a part of the picture

Structural enhancements

MIMD (Multiple Instruction Multiple Data)

- All processors perform different instructions on different data



Classical:
IBM 3084, Cray-2,

Today:

Most computer systems are many core processors, have specialized components operating in parallel etc.

Structural enhancements

MISD (Multiple Instruction Single Data)

- Several instructions operate on the same data
- Uncommon, but why not: for fault tolerance the same computation can be done in parallel, then the results compared

Many authors leave this class empty – we can discuss this!

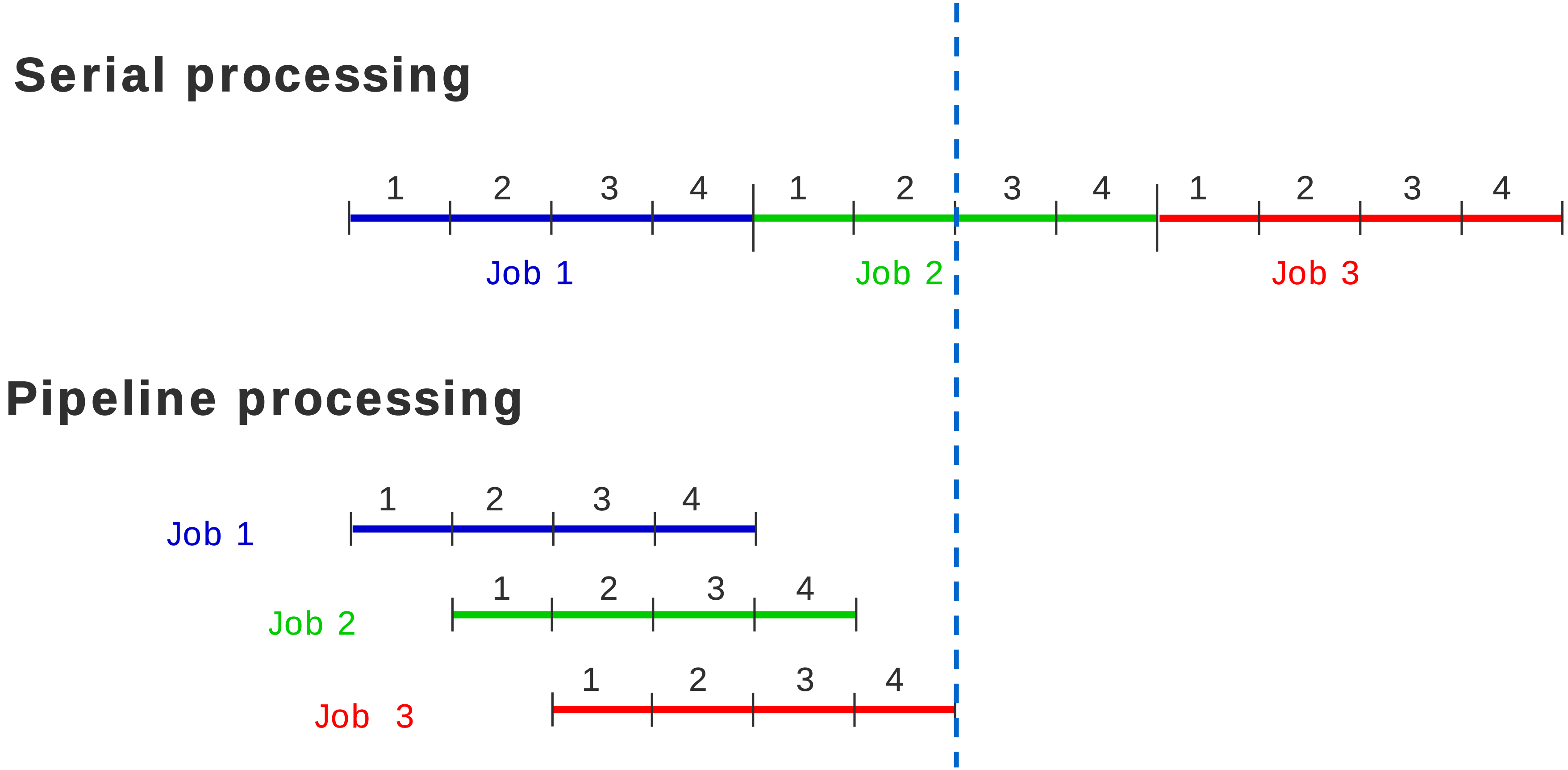
To summarize: this taxonomy does not really help today any longer as almost all of today's computer systems fall into the MIMD class…

Performance enhancement

PIPELINE PROCESSING

Pipeline processing

Processing of 3 similar jobs with 4 identical sub tasks each.



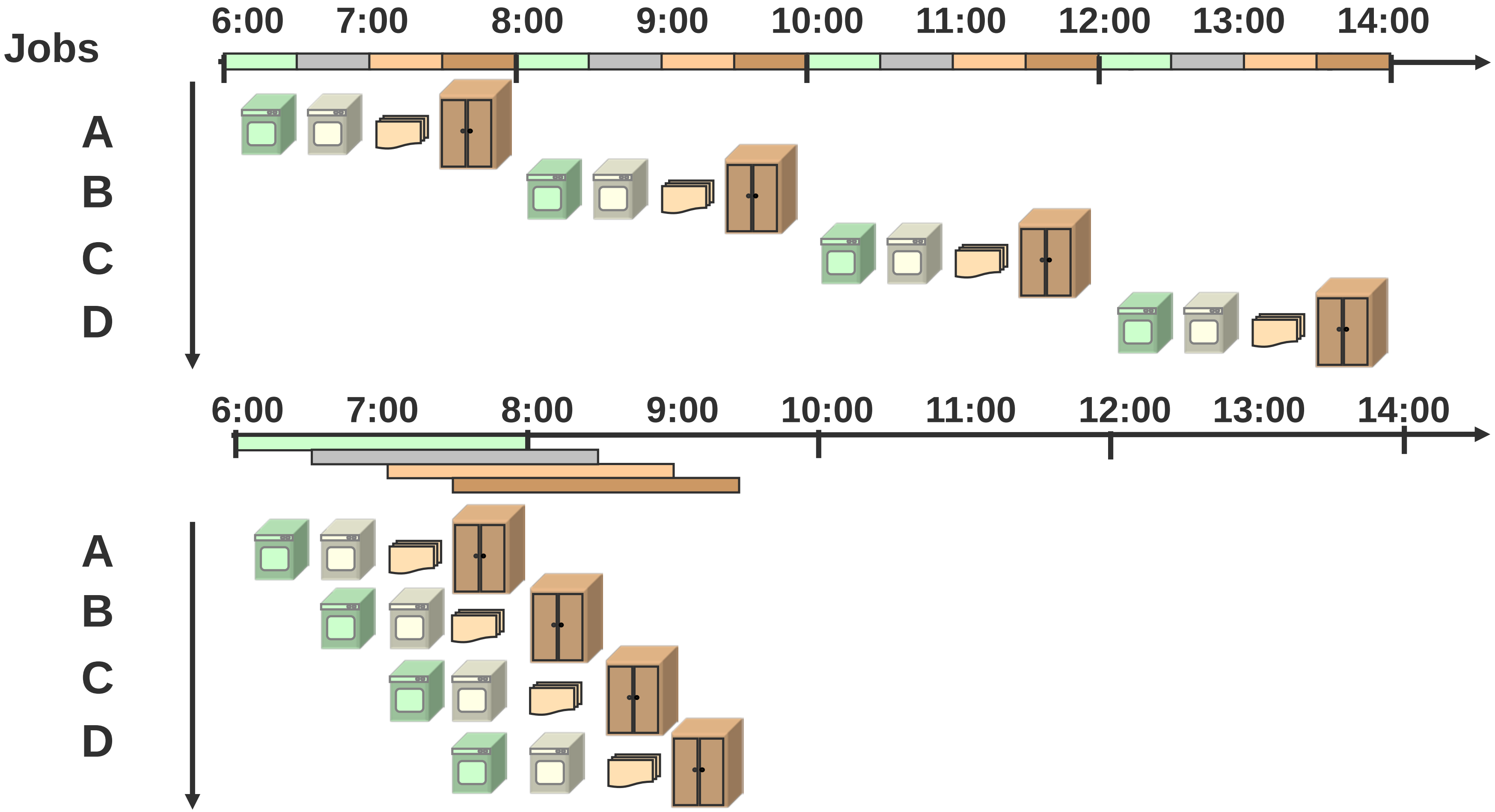
Example: laundry pipelining

Doing the laundry can be split into 4 separate tasks:

- Put the dirty laundry into the washing machine and start the program
- Put the wet clothes into the tumbler
- Iron, smooth out creases, pleat, fold …
- Put the clothes into the closet

Laundry pipelining

very realistic ….



Pipelining I

Pipelining

- Subdivision of an operation into several phases or sub operations
- Synchronous execution of the sub operations in different functional units
- Each functional unit is responsible for a single function

All functional units together plus their interconnection is called pipeline.

Instruction pipelining

- The pipeline principle is applied to processor instructions
- Successive instructions are executed one after another with a delay of a single cycle

Pipelining II

Each stage of a pipeline is called pipeline stage or pipeline segment.

The whole pipeline is clocked in a way that each cycle an instruction can be shifted one step further through the pipeline.

In an ideal scenario, an instruction is executed in a *k* stage pipeline within *k* cycles by *k* stages (…we will see problems due to hazards later).

If every clock cycle a new instruction is loaded into the pipeline, then *k* instructions are executed simultaneously and each instruction needs *k* cycles in the pipeline.

Pipelining III

Latency**:** Duration of the complete processing of an instruction. This is the time an instruction needs to go through all $\boldsymbol{k}$ stages of the pipeline.

Throughput**:** Number of instructions leaving the pipeline per clock cycle. This number should be close to 1 for a scalar processor.

Speed-up of Instruction execution

Hypothetical processor without pipeline: $n \times k$ cycles
(k stage pipeline, n instructions)

Pipelined processor with a k stage pipeline: $k+(n-1)$ cycles
(under ideal conditions: k cycles latency, throughput of 1)

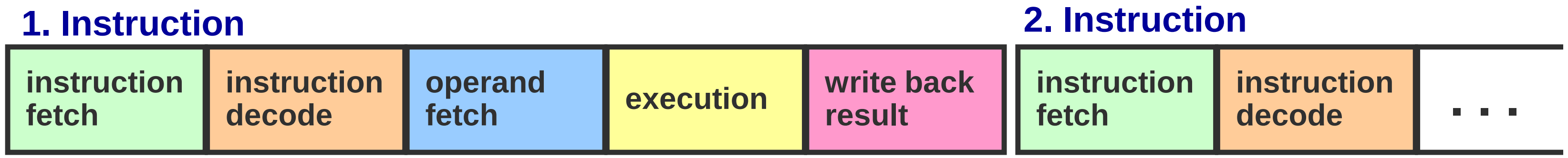
This results in a speed-up of:

$$S = \frac{nk}{k+n-1} = \frac{k}{\frac{k}{n}+1-\frac{1}{n}}$$

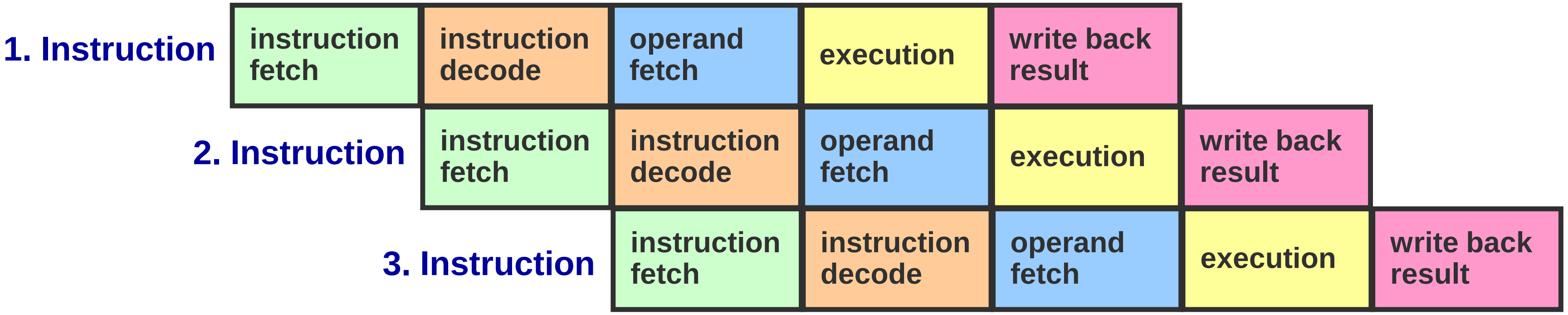
Assuming an infinite number of instructions ($n \rightarrow \infty$) the speed-up of a processor with a k stage pipeline equals k.

Pipelining

Sequential execution:



Pipelining:



Architecture of a 5-stage pipeline I

Instruction Fetch (IF)

- Load the opcode of the instruction from memory (or instruction cache) into the opcode register
- Increment the program counter

Instruction Decode (ID)

- Generate internal signals based on the opcode or jump to the appropriate microprogram

Operand Fetch (OF)

- Load the operands from the registers into the operand registers of the ALU
- Calculate the effective address using the address generating unit for load/store or branch instructions

Architecture of a 5-stage pipeline I

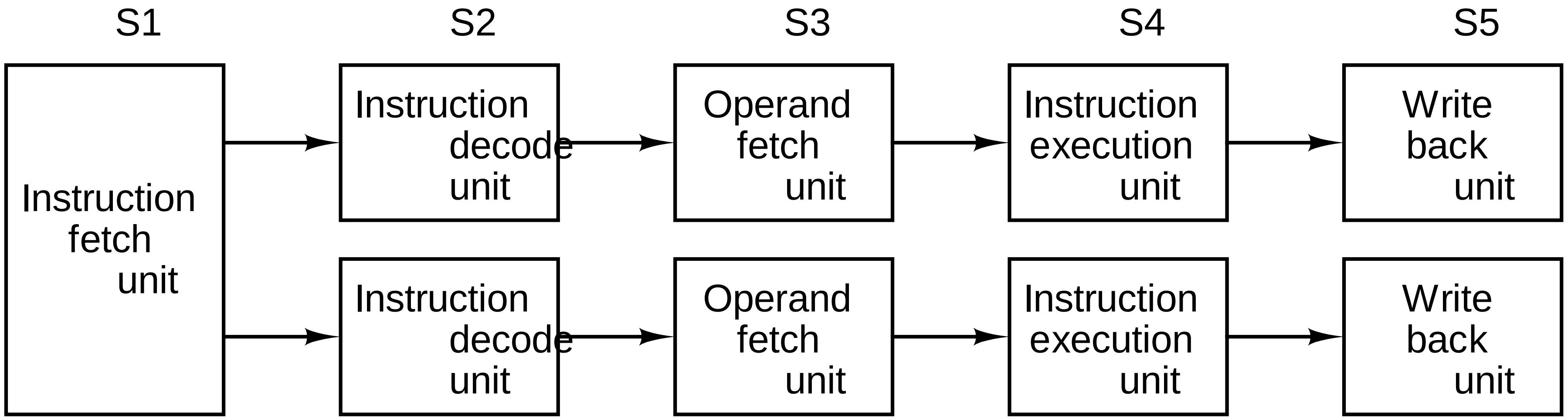
Execution (EXE, ALU operation)

- The execution unit performs the requested operation

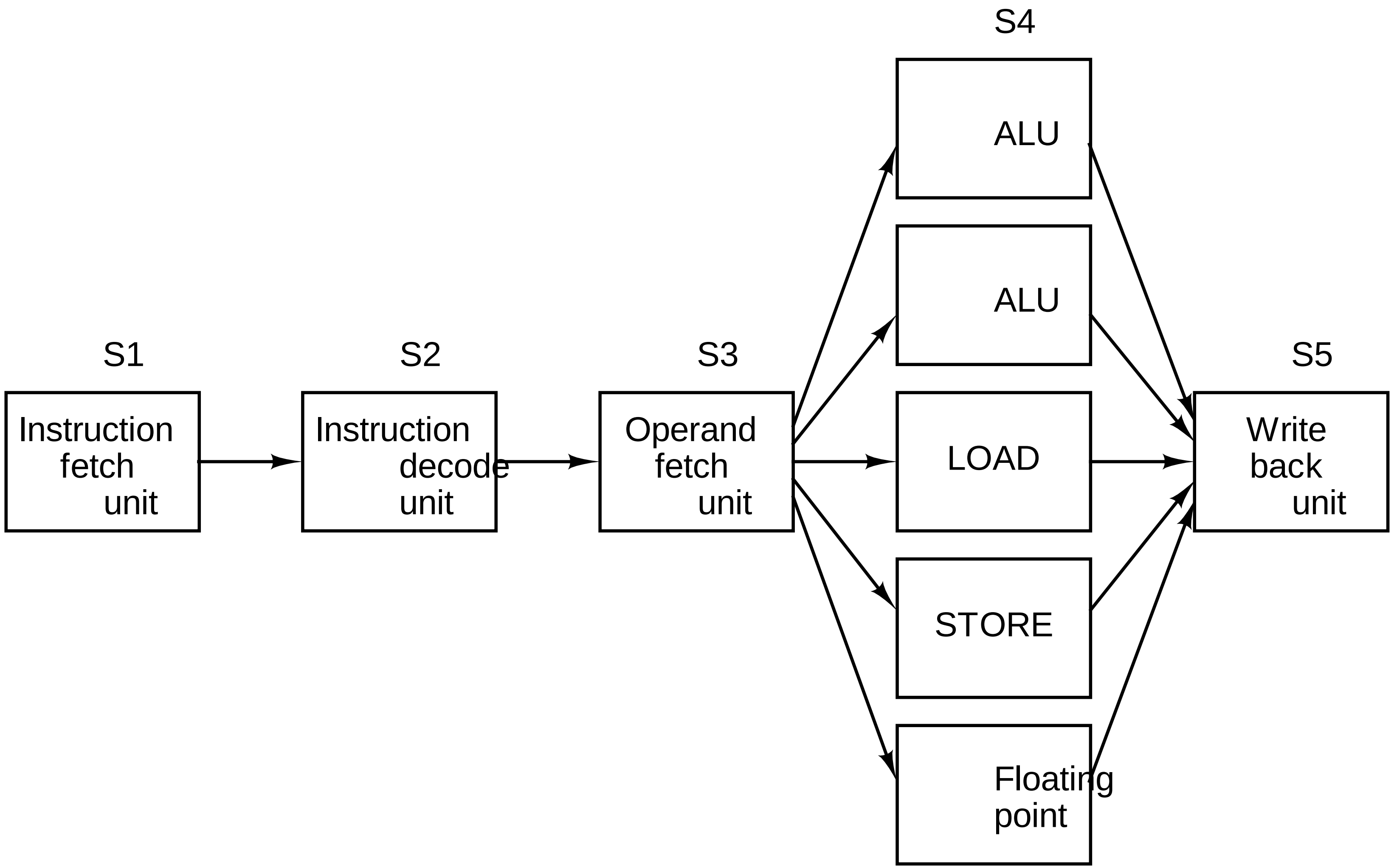
Result Write Back (WB)

- Write back the result into a register or memory
- Instructions without a result do nothing
- Load/store instructions put the address on the address bus and transfer the data between register and memory

Performance enhancement: two pipelines



Performance enhancement: specialized EXE-units



Questions & Tasks

- Do you know some more examples for pipelining in your life?
- How much can a pipeline speed-up a single instruction?
- What determines the clock speed of a pipeline?
- What could be bottlenecks in pipeline processing? Think of the tasks of the different stages!

Performance enhancement

PIPELINING - TYPES OF PIPELINE HAZARDS

Pipeline Hazards

Pipeline hazards: phenomena that disrupt the smooth execution of a pipeline.

Example:

- If we assume a unified cache with a single read port (instead of separate I- and D-caches)
  ➡ a memory read conflict appears among IF and OF stages.
- The pipeline has to stall one of the accesses until the required memory port is available.

A stall is also called a pipeline bubble.

Three types of pipeline hazards

Data hazards arise because of the unavailability of an operand

- For example, an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.

Structural hazards may arise from some combinations of instructions that cannot be accommodated because of resource conflicts

- For example, if the processor has only one register file write port and two instructions want to write in the register file at the same time.

Control hazards arise from branch, jump, and other control flow instructions

- For example, a taken branch interrupts the flow of instructions into the pipeline
  - ➡ the branch target must be fetched before the pipeline can resume execution.

Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline.

Types of Pipeline Hazards

DATA HAZARDS

Pipeline hazards due to data dependence

After a load instruction the loaded value is not available to the following instruction in the next cycle.

If an instruction needs the result of a preceding instruction it has to wait.

Example:

```
 ADD R1,R2,R1;              R1←R1+R2
 ADD R3,R1,R3;              R3←R3+R1
```

Pipeline hazards due to data dependence

After a load instruction the loaded value is not available to the following instruction in the next cycle.

If an instruction needs the result of a preceding instruction it has to wait.

Example:

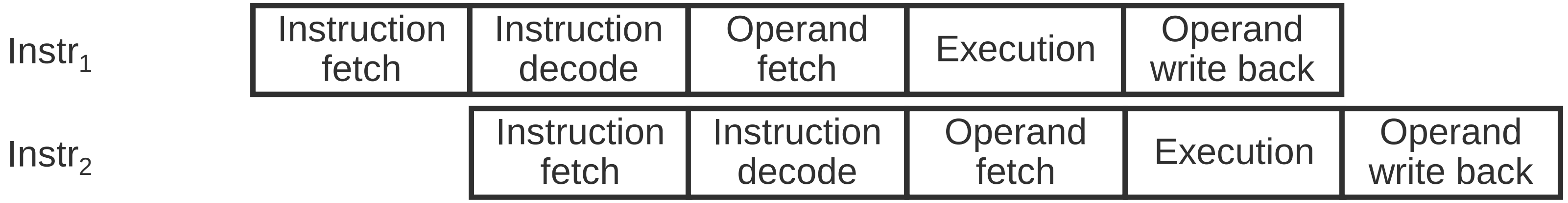
```
ADD R1,R2,R1;          R1←R1+R2
ADD R3,R1,R3;          R3←R3+R1
```

Data hazards

Dependencies between instructions may cause data hazards when Instr_1 and Instr_2 are so close that their overlapping within the pipeline would change their access order to registers.

Three types of data hazards

- Read After Write (RAW)
  - Instr_2 tries to read operand before Instr_1 writes it
- Write After Read (WAR)
  - Instr_2 tries to write operand before Instr_1 reads it
- Write After Write (WAW)
  - Instr_2 tries to write operand before Instr_1 writes it

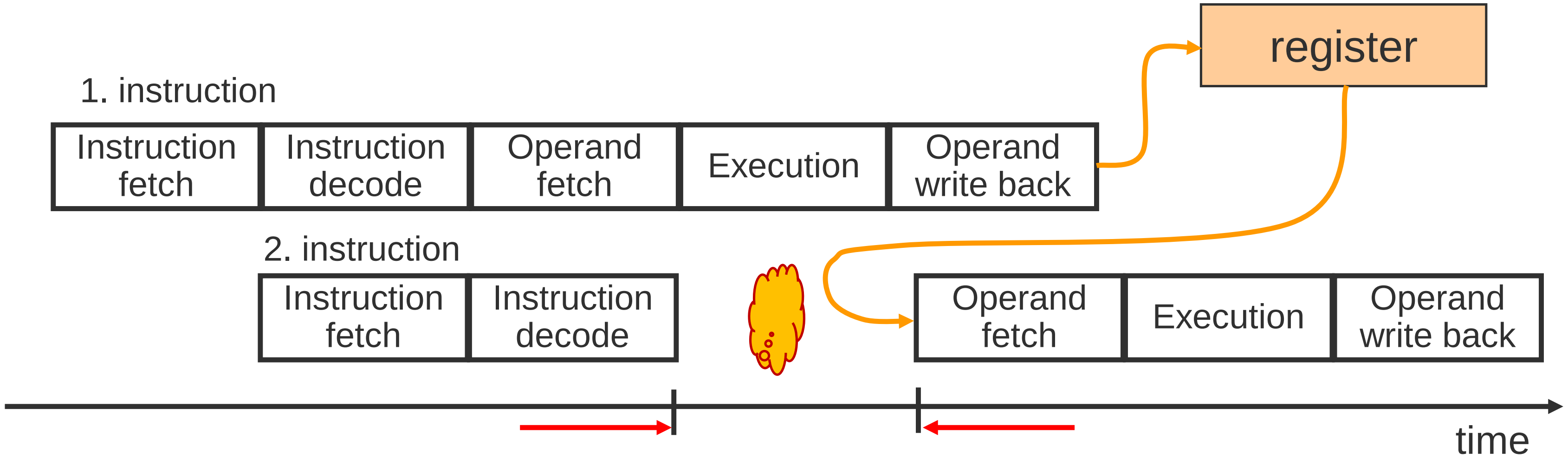


Instr_1	Instruction fetch	Instruction decode	Operand fetch	Execution	Operand write back	
Instr_2		Instruction fetch	Instruction decode	Operand fetch	Execution	Operand write back

Read-after-Write-Conflict (True Dependence)

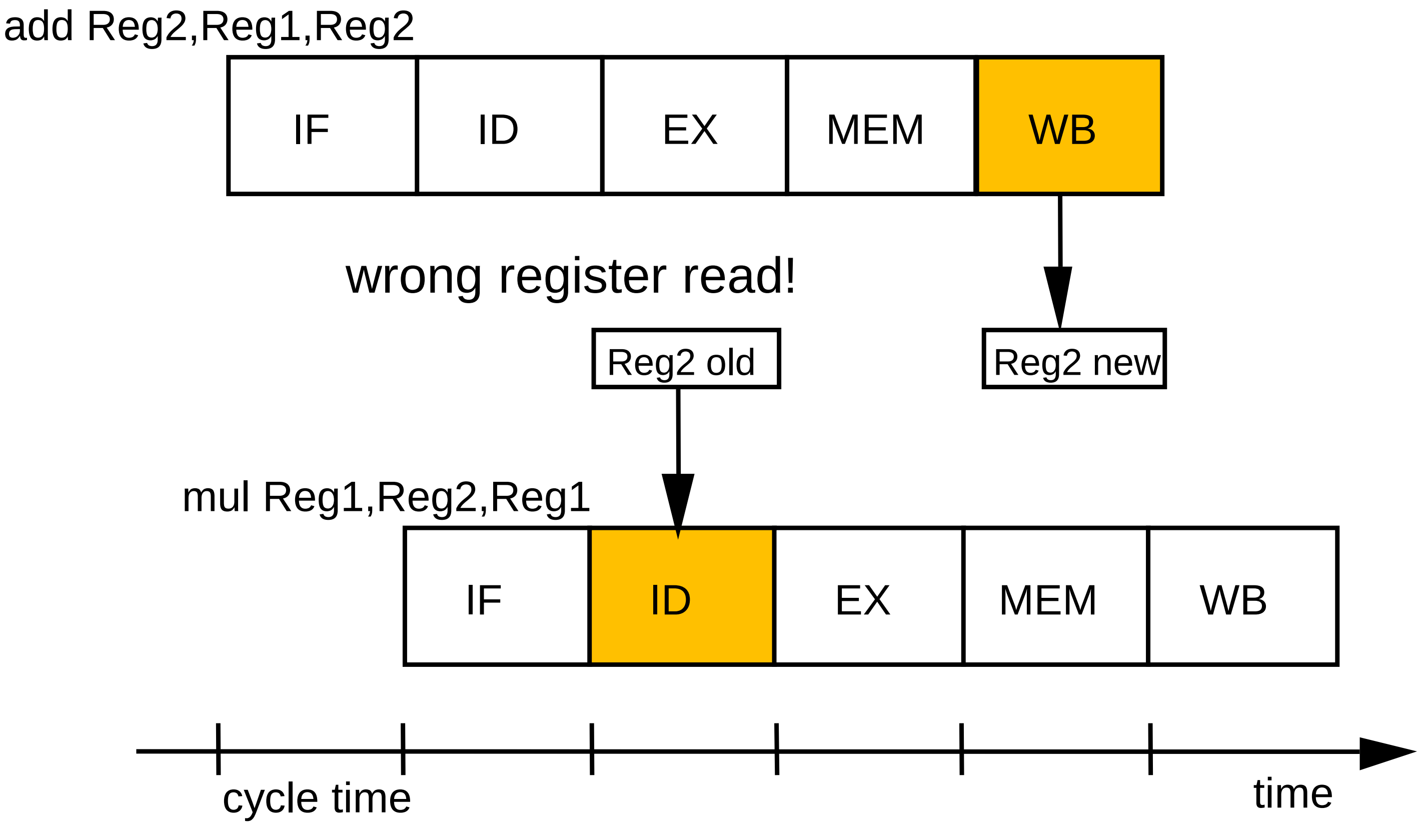
Using a simple 5 stage pipeline this example shows that the operand fetch phase of the 2nd instruction comes before the 1st instruction writes back its result

- Delaying the pipeline is necessary!



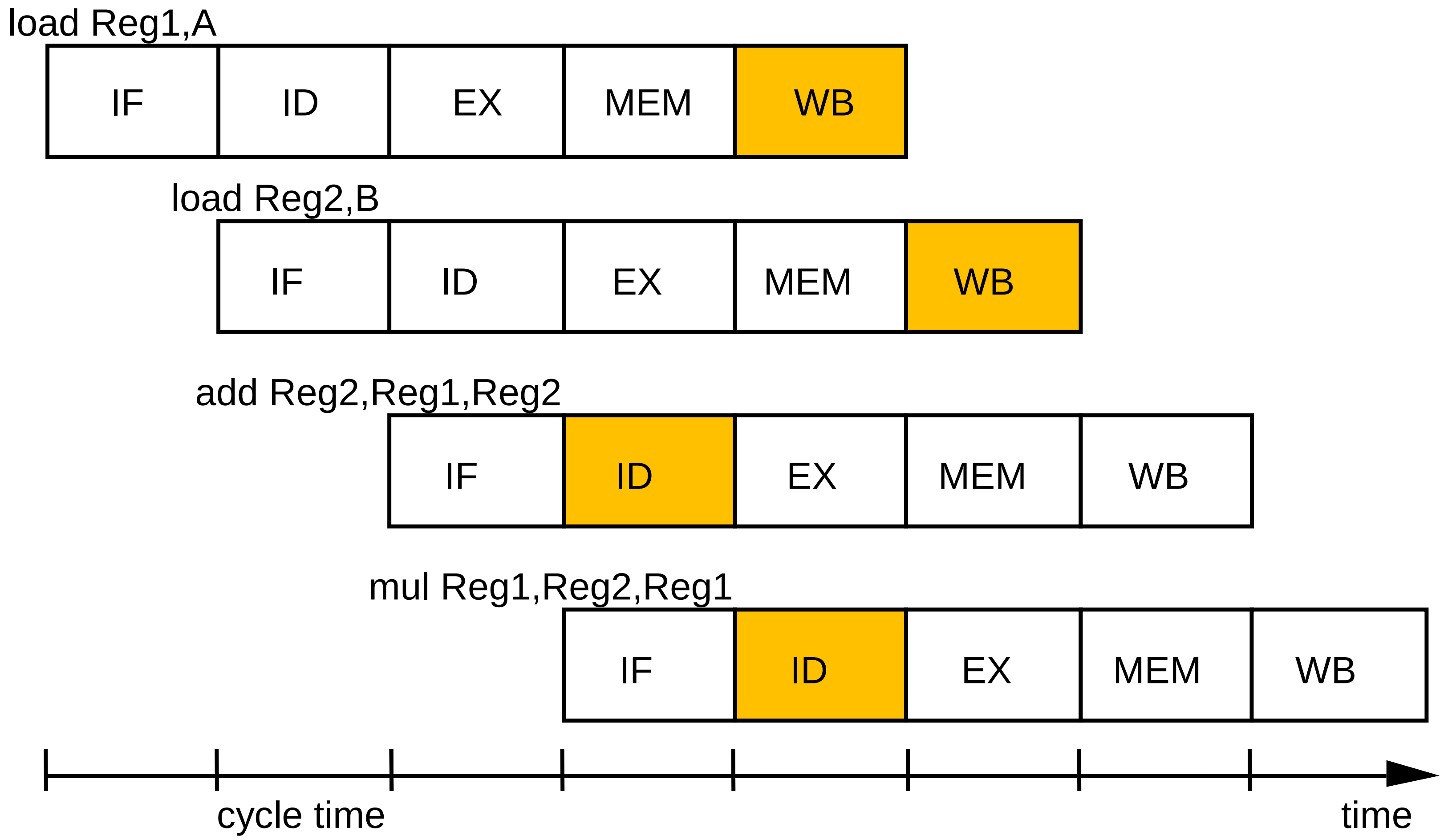
Pipeline – bubble (delay)

Pipeline conflict due to a data hazard

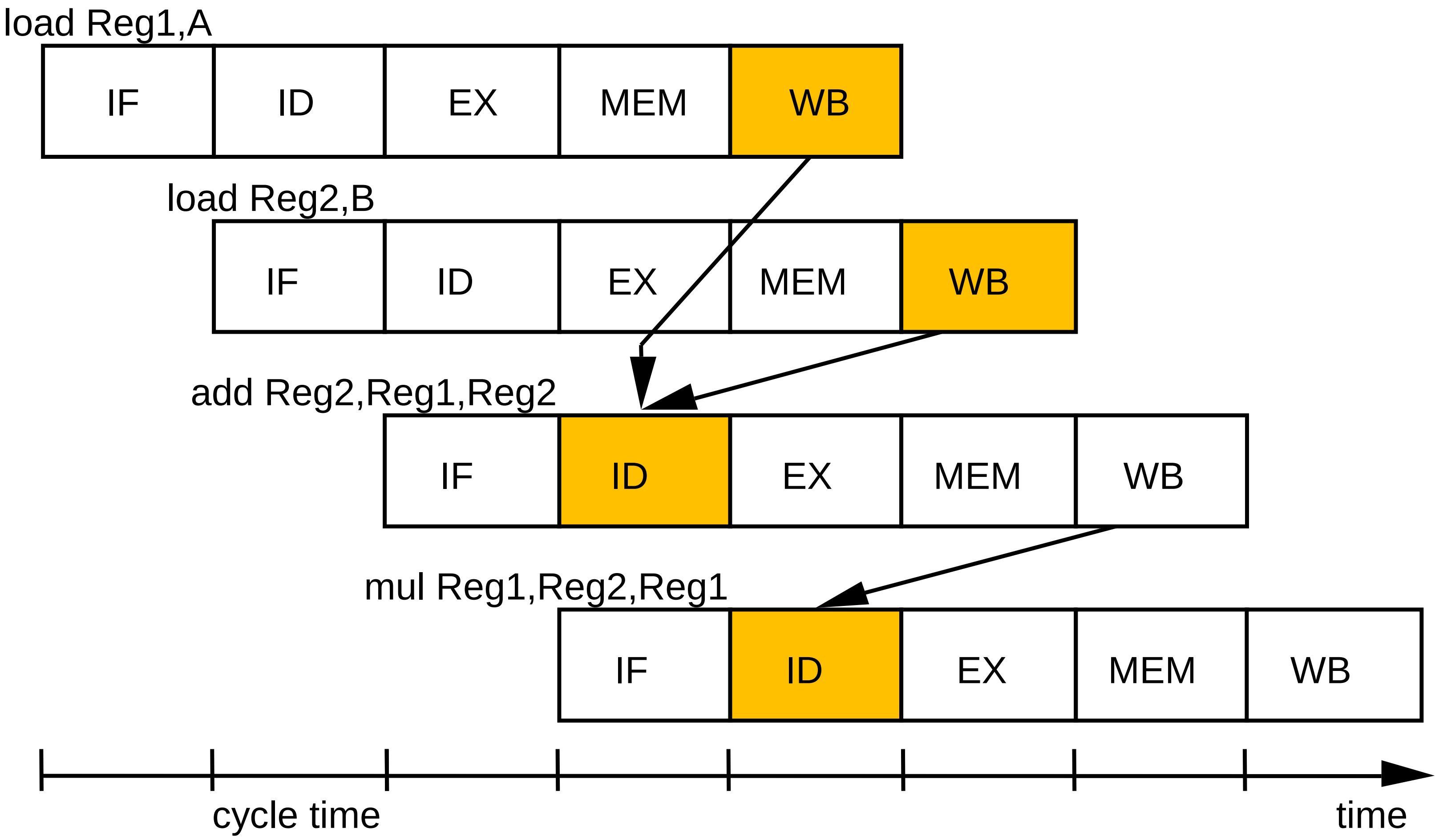


add Reg2,Reg1,Reg2; Reg2 ← Reg1 + Reg2
mul Reg1,Reg2,Reg1; Reg1 ← Reg1 * Reg2

Data hazards in an instruction pipeline



Data hazards in an instruction pipeline



WAR and WAW - Can they happen in our simple pipeline?

WAR and WAW can’t happen in the simple 5 stage pipeline, because:

- All instructions take 5 stages
- Register reads are always in stage 2
- Register writes are always in stage 5

WAR and WAW may happen e.g. in superscalar pipes.

Solutions for data hazards from true data dependences

Software solution (Compiler scheduling)

- putting no-op (NOP) instructions after each instruction that may cause a hazard

- instruction scheduling
  ➡ rearrange code to reduce no-ops

Software solutions

Insertion of NOP instructions by the compiler

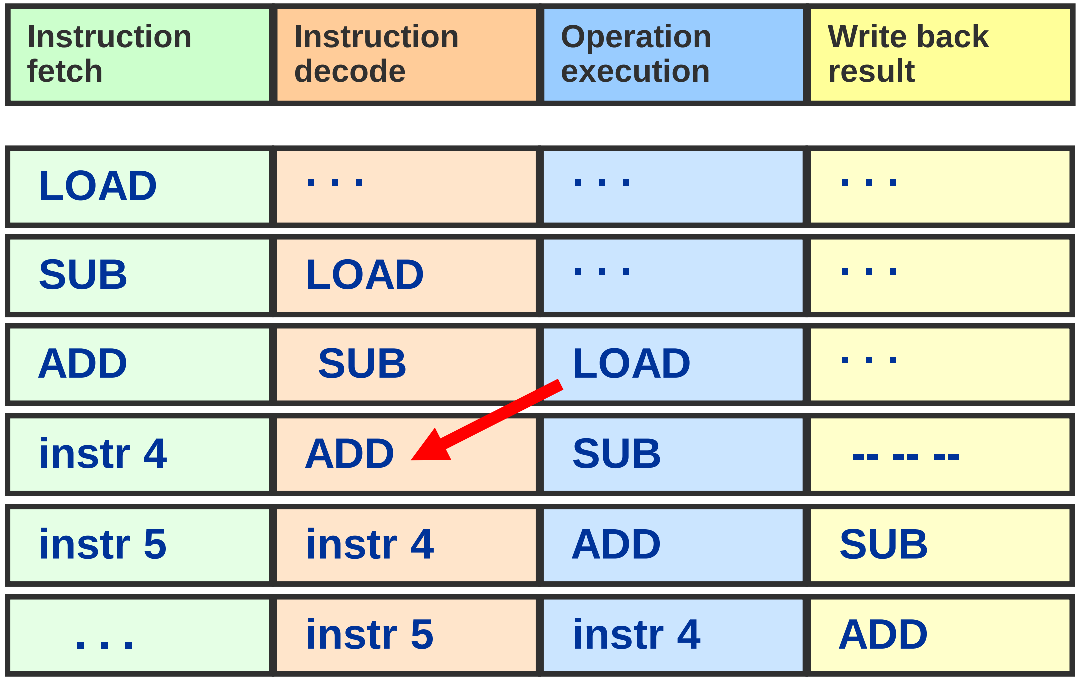
```
LOAD    R1, <addr>
NOP
ADD     R1, R2, R3
SUB     R4, R5, R6
```

Instruction fetch	Instruction decode	Operation execution	Write back result
LOAD	. . .	. . .	. . .
NOP	LOAD	. . .	. . .
ADD	NOP	LOAD	. . .
SUB	ADD	NOP	-- -- --
instr 4	SUB	ADD	NOP
instr 5	instr 4	SUB	ADD
. . .	instr 5	instr 4	SUB

Software solutions

Reordering of instructions by the compiler

```
LOAD    R1, <addr>
SUB     R4, R5, R6
ADD     R1, R2, R3
```



Instruction fetch	Instruction decode	Operation execution	Write back result
LOAD	. . .	. . .	. . .
SUB	LOAD	. . .	. . .
ADD	SUB	LOAD	. . .
instr 4	ADD	SUB	-- -- --
instr 5	instr 4	ADD	SUB
. . .	instr 5	instr 4	ADD

Hardware solutions

Hardware solutions: Hazard detection logic necessary!

Delay Insertion

- Stalling/Interlocking: stall pipeline for one or more cycles

Bypass techniques

- Forwarding:
  - Result Forwarding
    Example: The result in ALU output of Instr$_1$ in EX stage can immediately be forwarded back to ALU input of the EX stage as an operand for Instr$_2$
  - Load Forwarding
    Example: The load memory data register from MEM stage can be forwarded to ALU input of EX stage
- Forwarding with interlocking: Assuming that Instr$_2$ is data dependent on the load instruction Instr$_1$ then Instr$_2$ has to be stalled until the data loaded by Instr$_1$ becomes available in the load memory data register in MEM stage.
- Even when forwarding is implemented from MEM back to EX, one bubble occurs that cannot be removed.

Questions & Tasks

- Are hazards rather rare events or common? Think of typical programs, processes, operating systems, tasks of a computer etc.!
- Can we remove a true dependence? What can we do?
- Why is result forwarding really helpful? Think of typical program sequences!
- WAR and WAW sound strange. Find examples when they may happen!

Side Note: The MIPS Pipeline

As defined in Patterson & Hennessy, *Computer Organization and Design – The Hardware/Software Interface*, Section 4.5

Instruction Fetch (IF)

- Fetch instruction from memory. [→ main memory!]

Instruction Decode (ID)

- Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.

Execution (EXE)

- Execute the operation [all arithmetical and logical operations] or calculate an address [load and store].

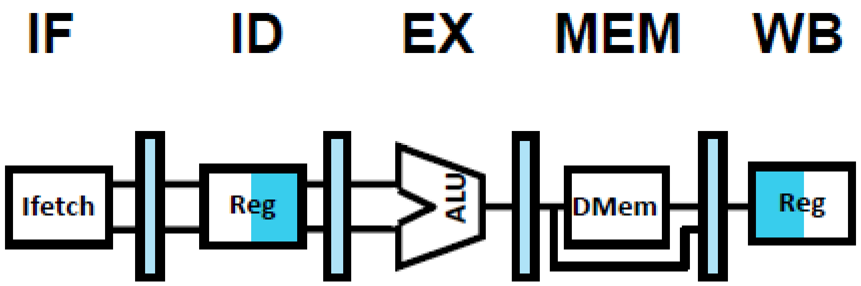
Memory Access (MEM)

- Access an operand in data memory. [Only relevant for load/store instructions, otherwise passive stage]

Write Back (WB)

- Write the result into a register.

Side Note: The MIPS Pipeline

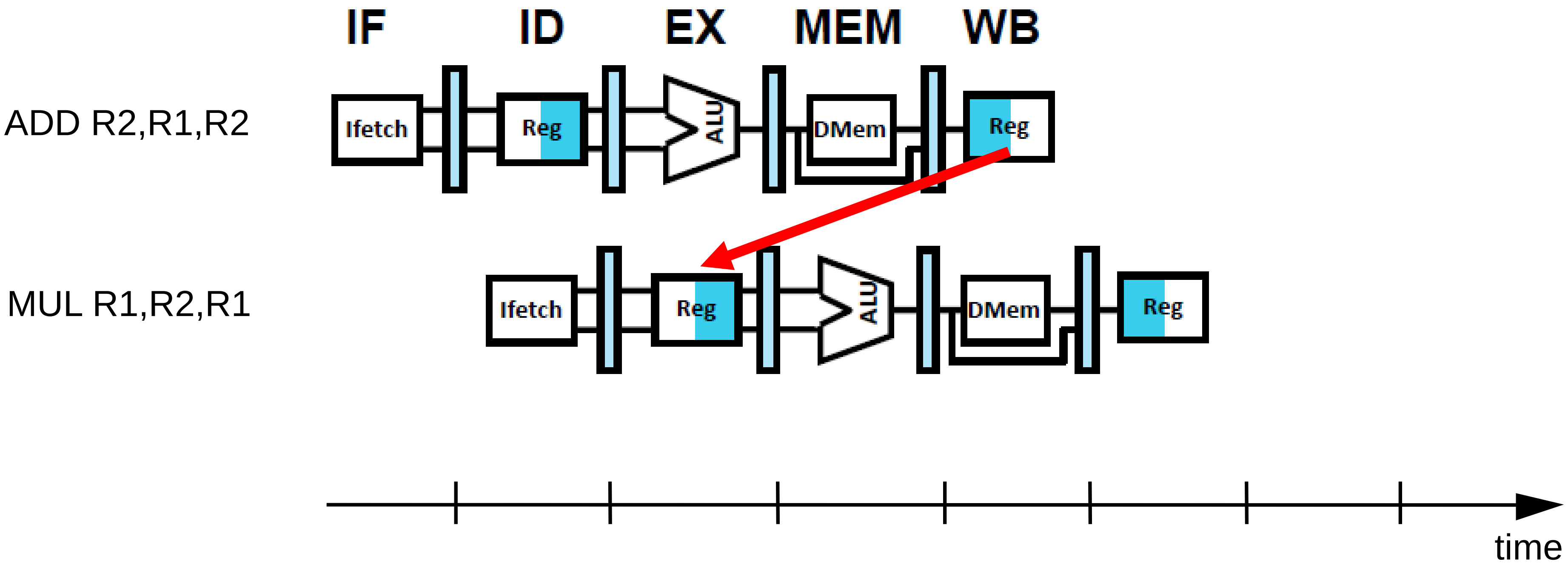


Source: Ben Juurlink, TU Berlin, lecture slides for „Advanced Computer Architectures“, 2015

→ Write affected register during first half of write back stage

→ Read operand registers during second half of instruction decode stage

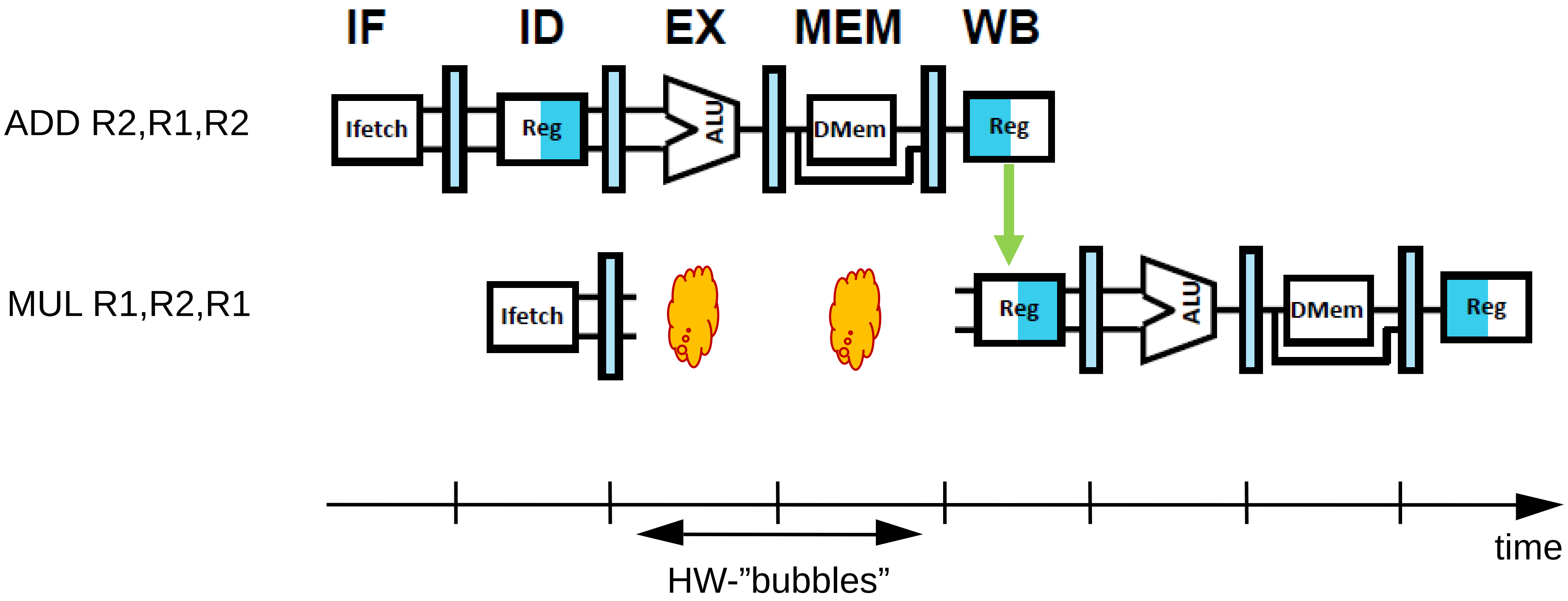
Data hazard: Hardware solution by stalling



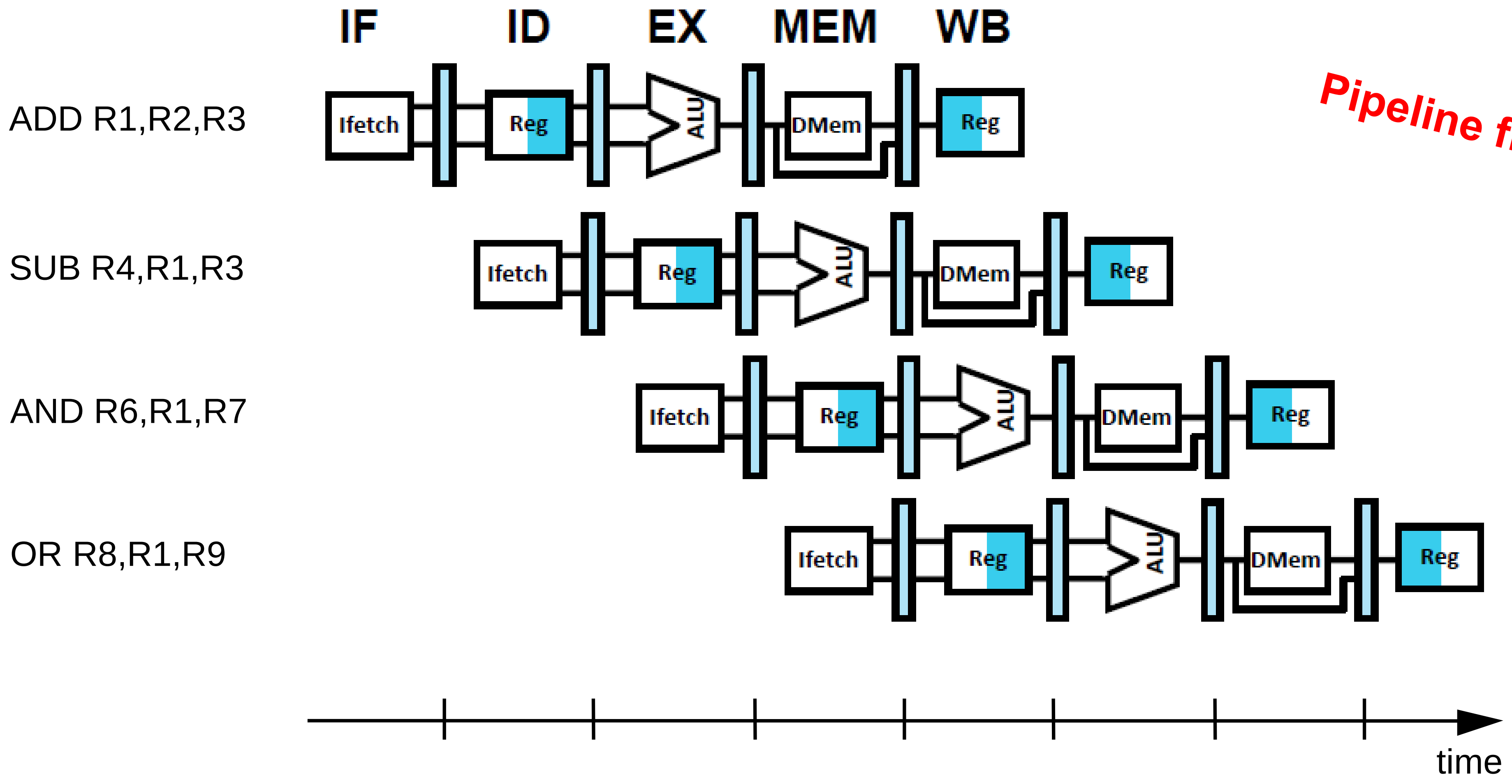
ADD R2,R1,R2

MUL R1,R2,R1

Data hazard: Hardware solution by stalling

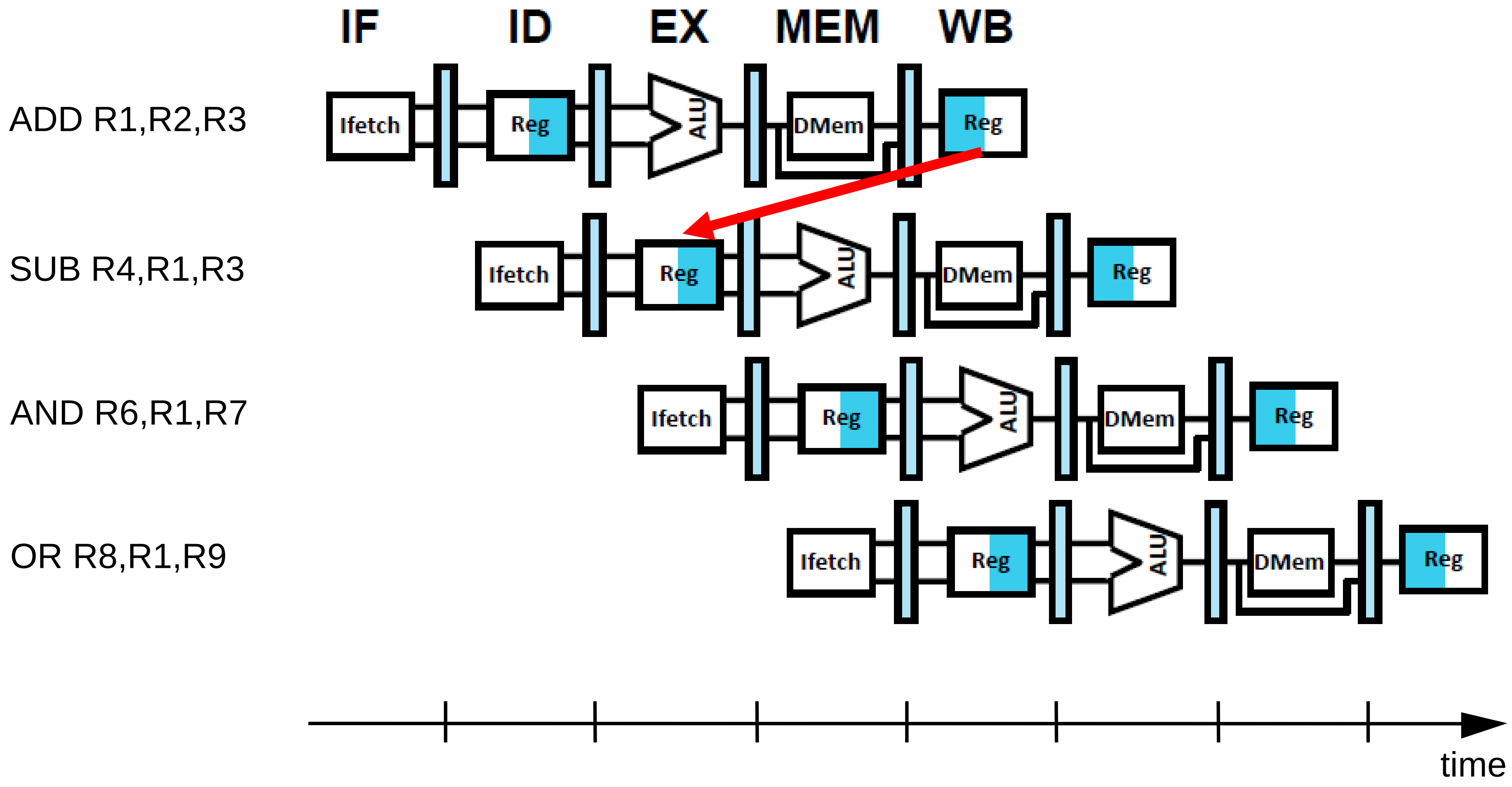


Data hazard: Hardware solution by forwarding



Pipeline filled with problems…

Data hazard: Hardware solution by forwarding



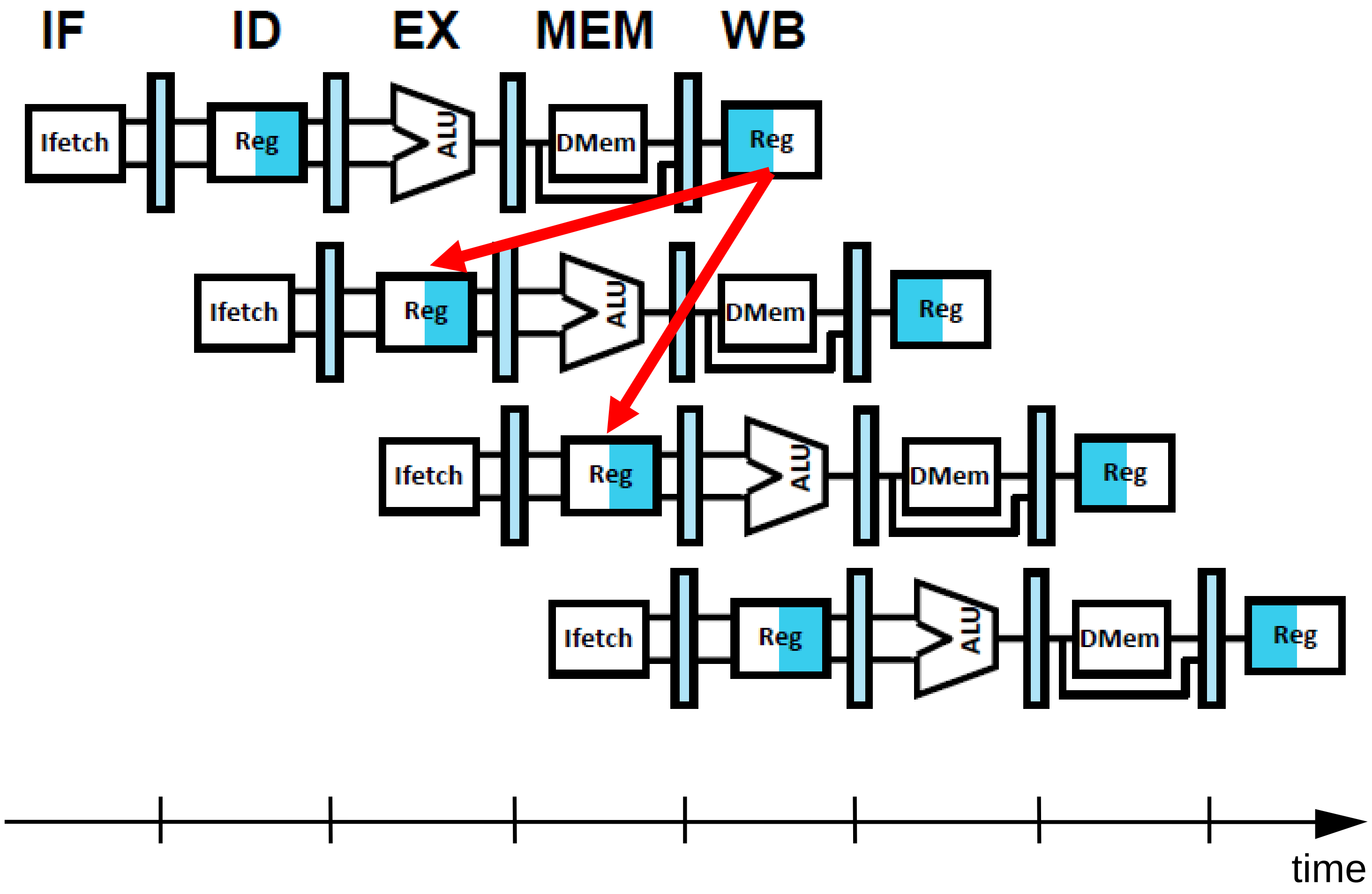
Data hazard: Hardware solution by forwarding

ADD R1,R2,R3

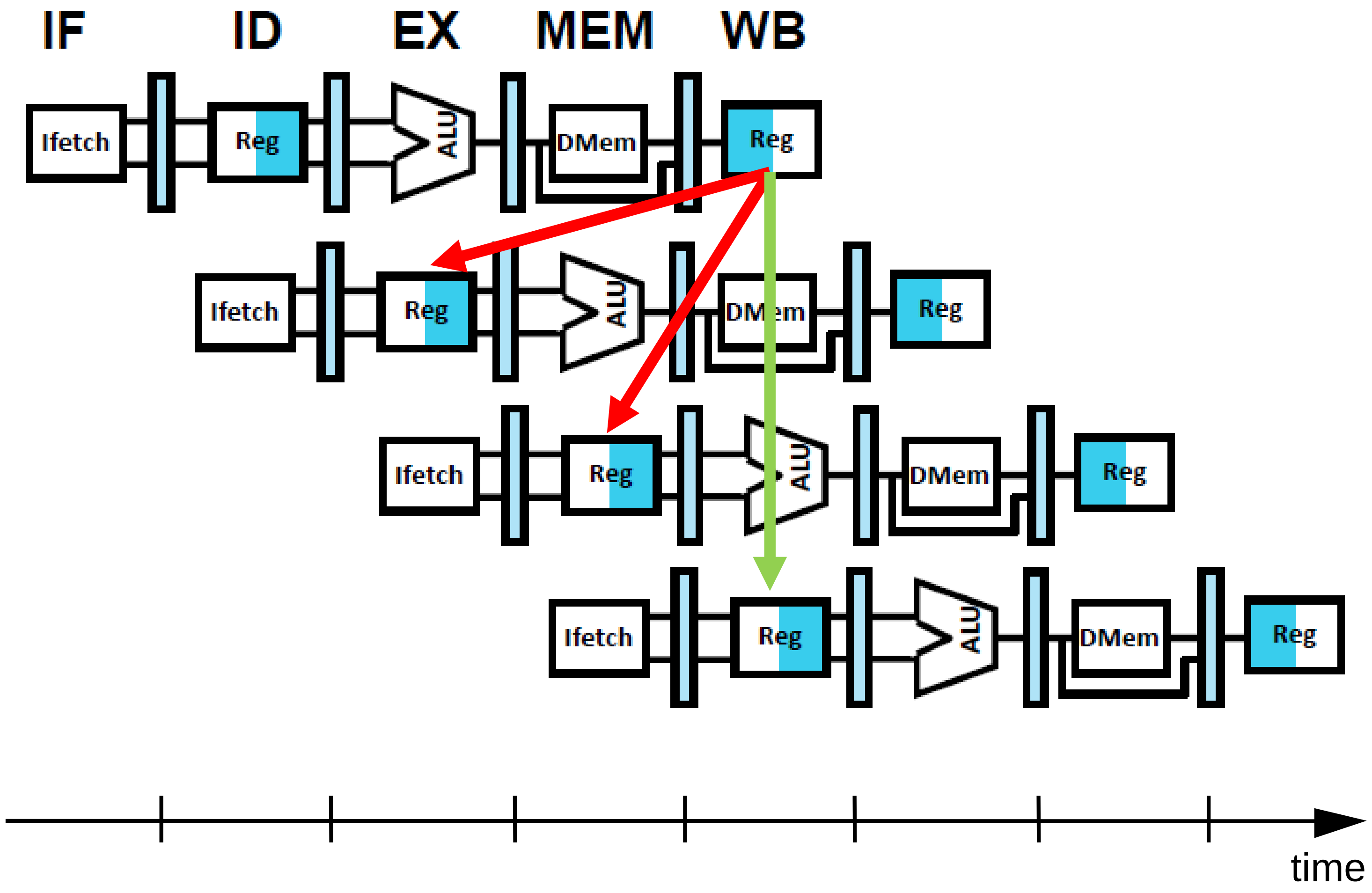
SUB R4,R1,R3

AND R6,R1,R7

OR R8,R1,R9



Data hazard: Hardware solution by forwarding



ADD R1,R2,R3

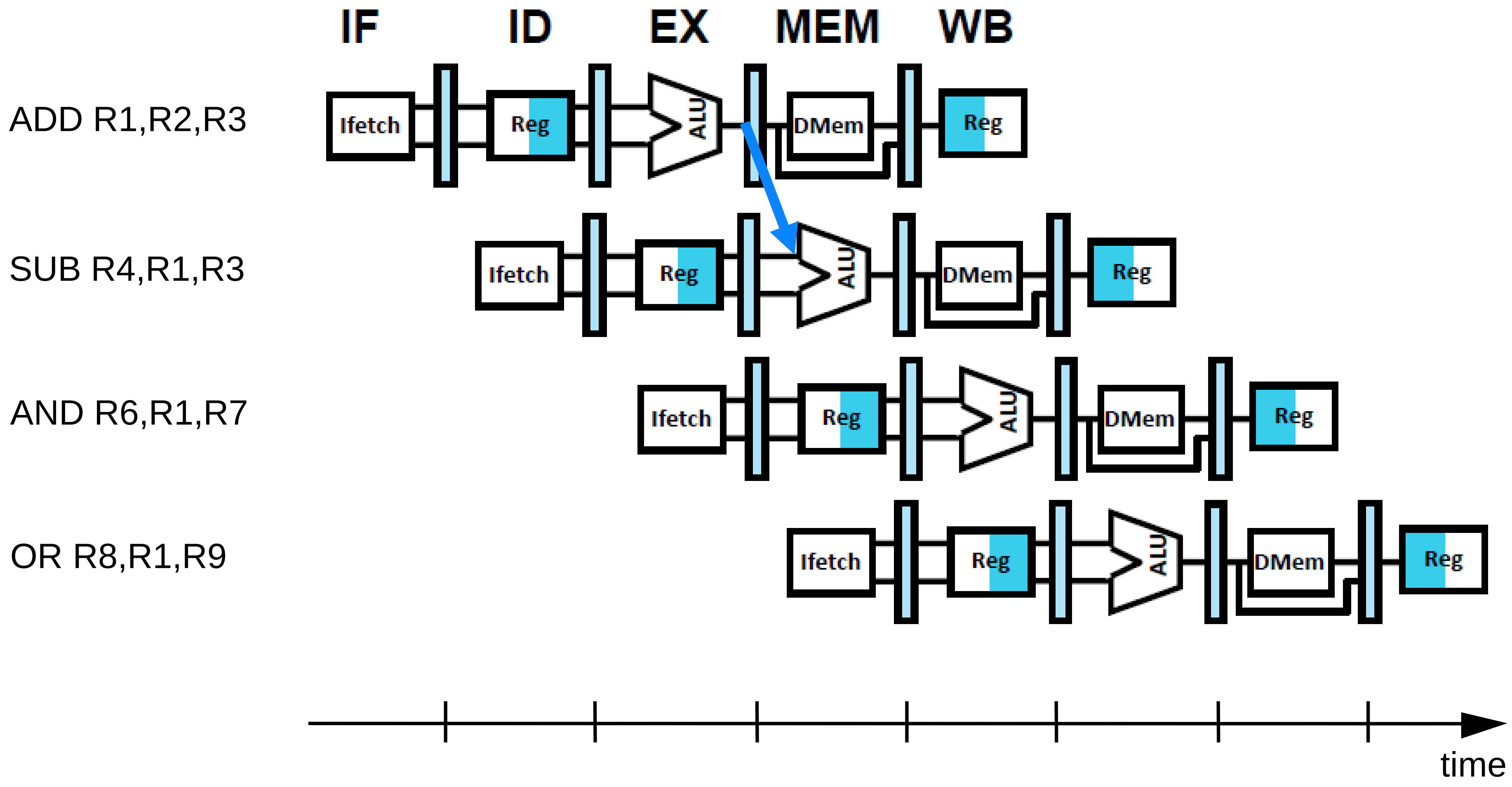
SUB R4,R1,R3

AND R6,R1,R7

OR R8,R1,R9

→ **Solution: Don’t wait until result is written back to register, but forward it to the next stage immediately**

Data hazard: Hardware solution by forwarding



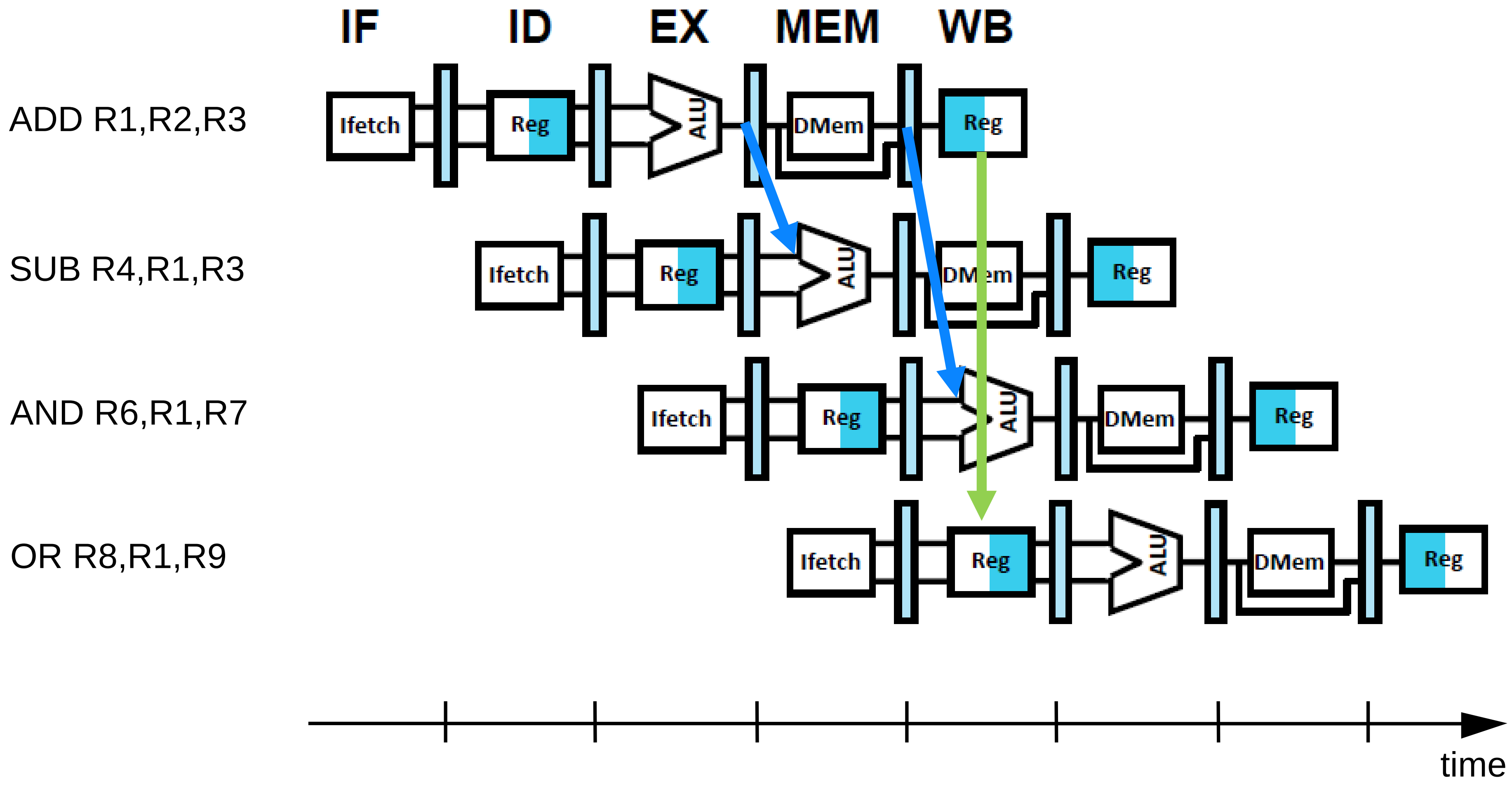
→ **Result Forwarding from EX to EX**

Result forwarding

Example:

The result in ALU output of Instr_1 in EX stage can immediately be forwarded back to ALU input of EX stage as an operand for Instr_2

Data hazard: Hardware solution by forwarding



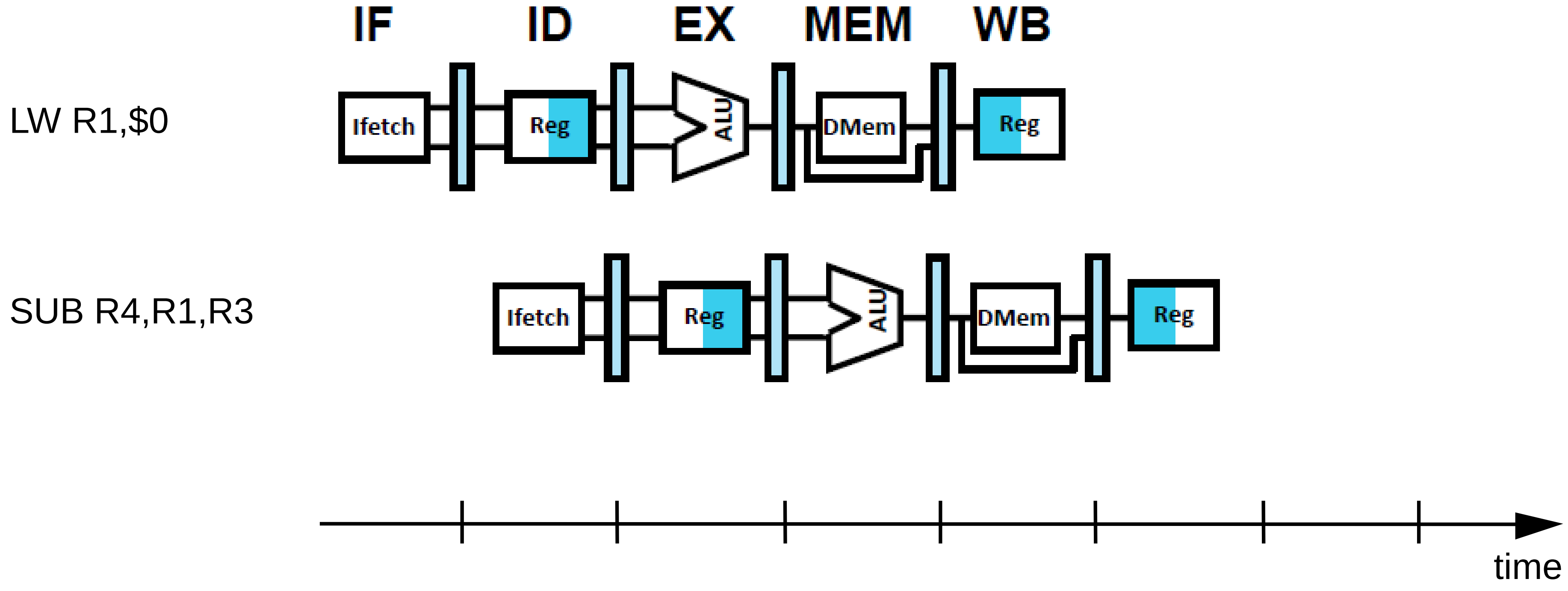
→ **Forward from MEM to EXE**

Load forwarding

Example:

The load memory data register from MEM stage can be forwarded to ALU input of EX stage.

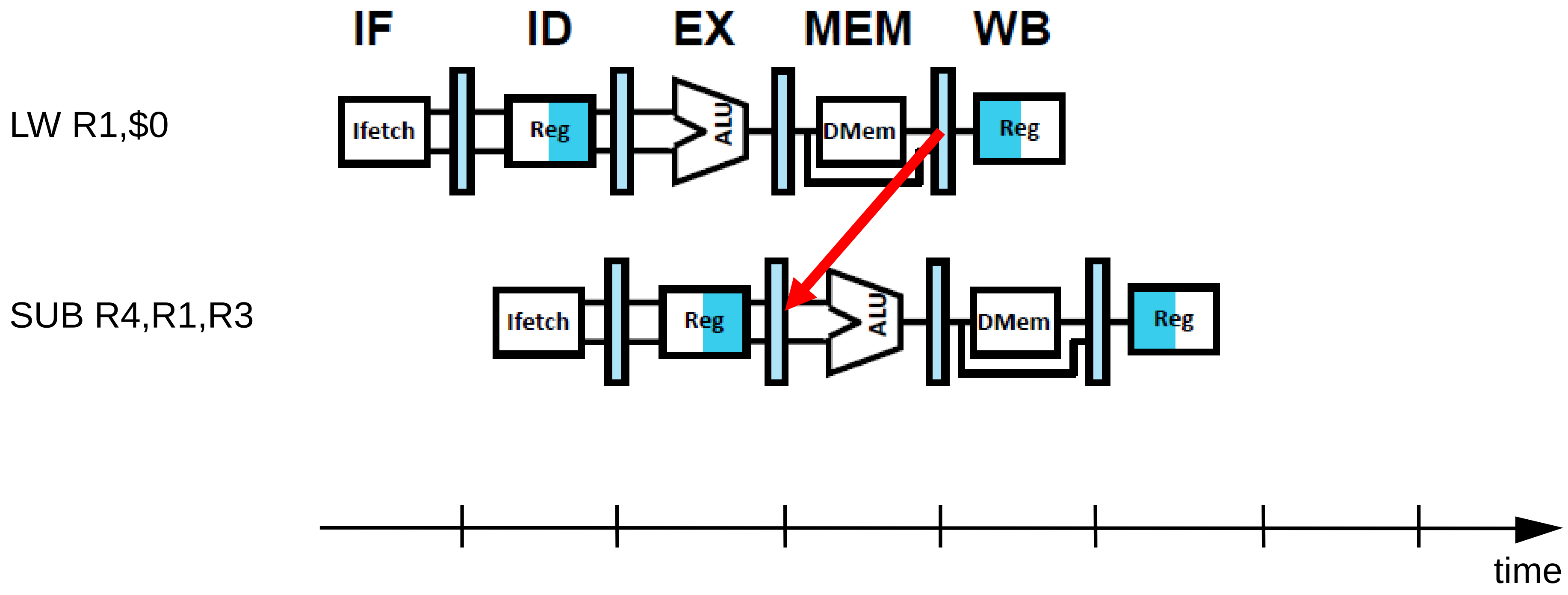
Load-use data hazard



LW R1,$0

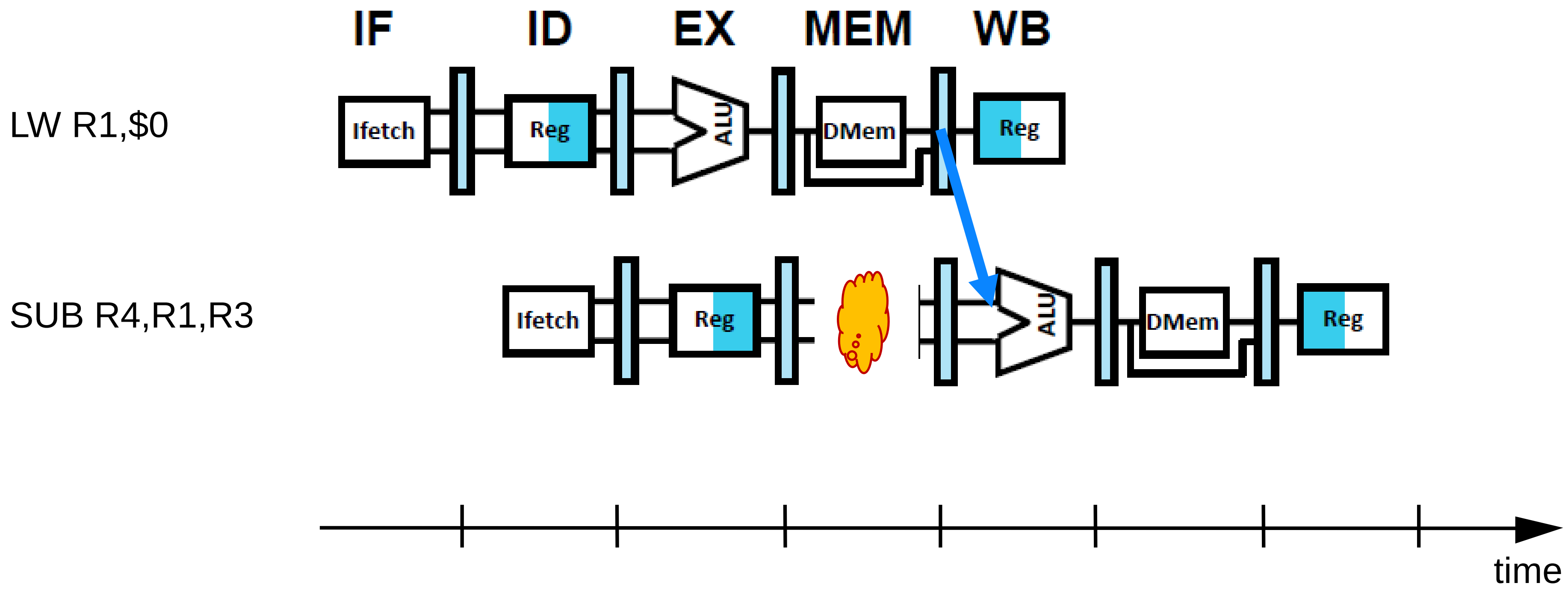
SUB R4,R1,R3

Load-use data hazard



→ **Data hazard even with forwarding!**

Load-use data hazard



→ **Need stalling AND forwarding**

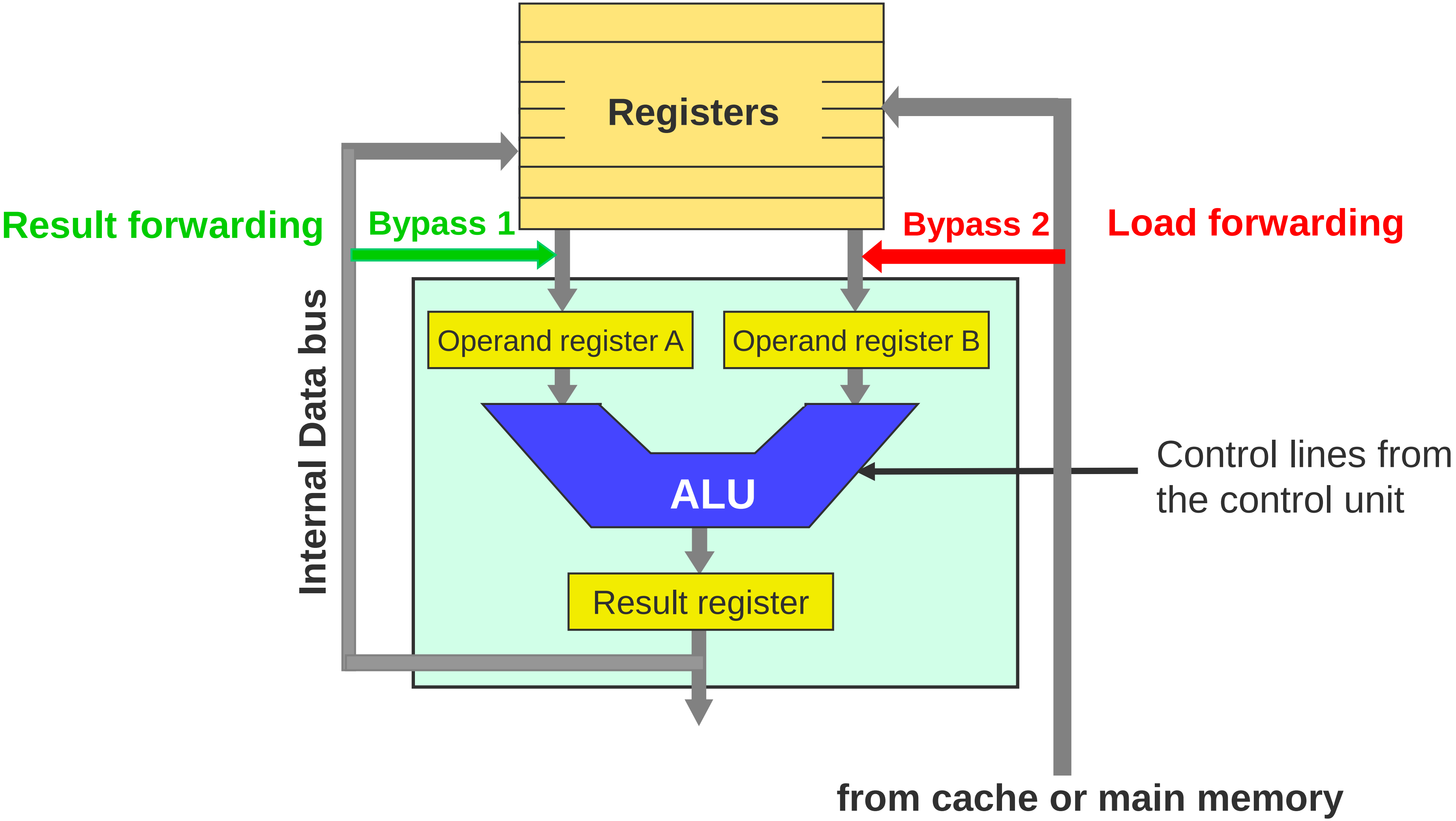
Forwarding with interlocking

Assuming that Instr_2 is data dependent on the load instruction Instr_1.

Then, Instr_2 has to be stalled until the data loaded by Instr_1 becomes available in the load memory data register in MEM stage.

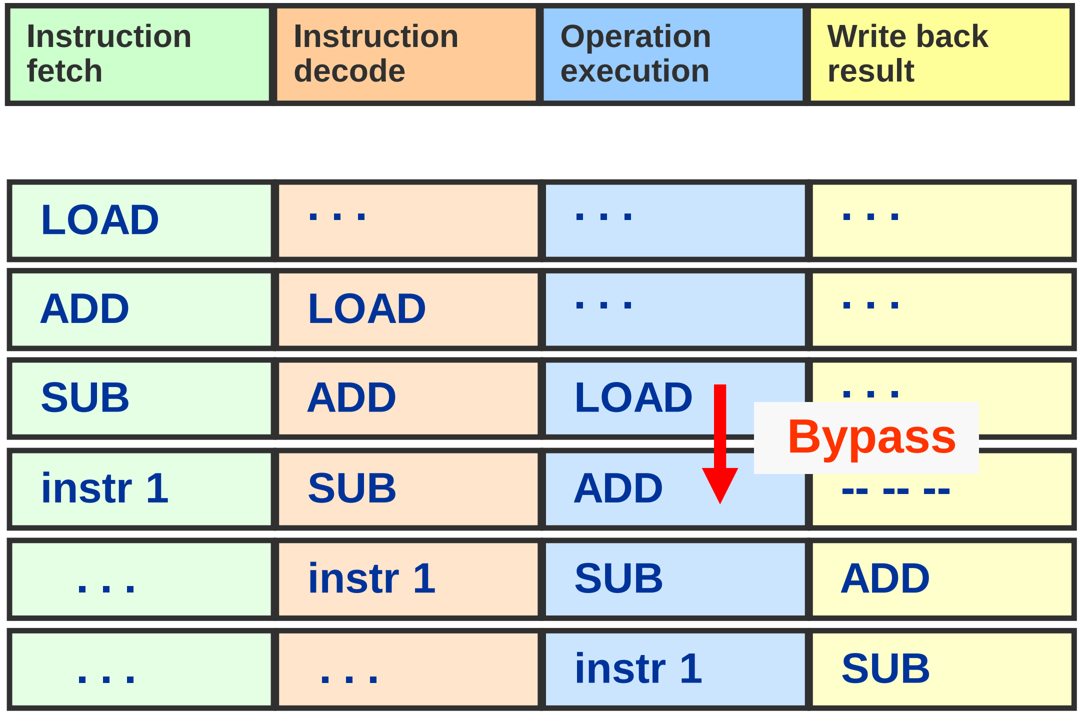
→ Even when forwarding is implemented from MEM back to EX, one bubble occurs that **cannot be removed**.

Bypass techniques



Example

```
LOAD   <Address>, R1      R1 ← (<Address>)
ADD    R1, R2, R3         R3 ← R1 + R2
SUB    R4, R5, R6         R6 ← R4 - R5
```



Instruction fetch	Instruction decode	Operation execution	Write back result
LOAD	. . .	. . .	. . .
ADD	LOAD	. . .	. . .
SUB	ADD	LOAD	. . .
instr 1	SUB	ADD	-- -- --
. . .	instr 1	SUB	ADD
. . .	. . .	instr 1	SUB

Bypass

Yet another (simple) pipeline…

During the execution phase of the ADD instruction the result of the LOAD is written into the register and, thus, the bypass is needed to provide the result early enough.

Questions & Tasks

- Bypassing looks fine – what is the price to pay?
- What is an assumption for load forwarding to work? Think of memory access times!

Types of Pipeline Hazards

STRUCTURAL HAZARDS

Three types of pipeline hazards

Data hazards arise because of the unavailability of an operand ✓

- For example, an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.

Structural hazards may arise from some combinations of instructions that cannot be accommodated because of resource conflicts

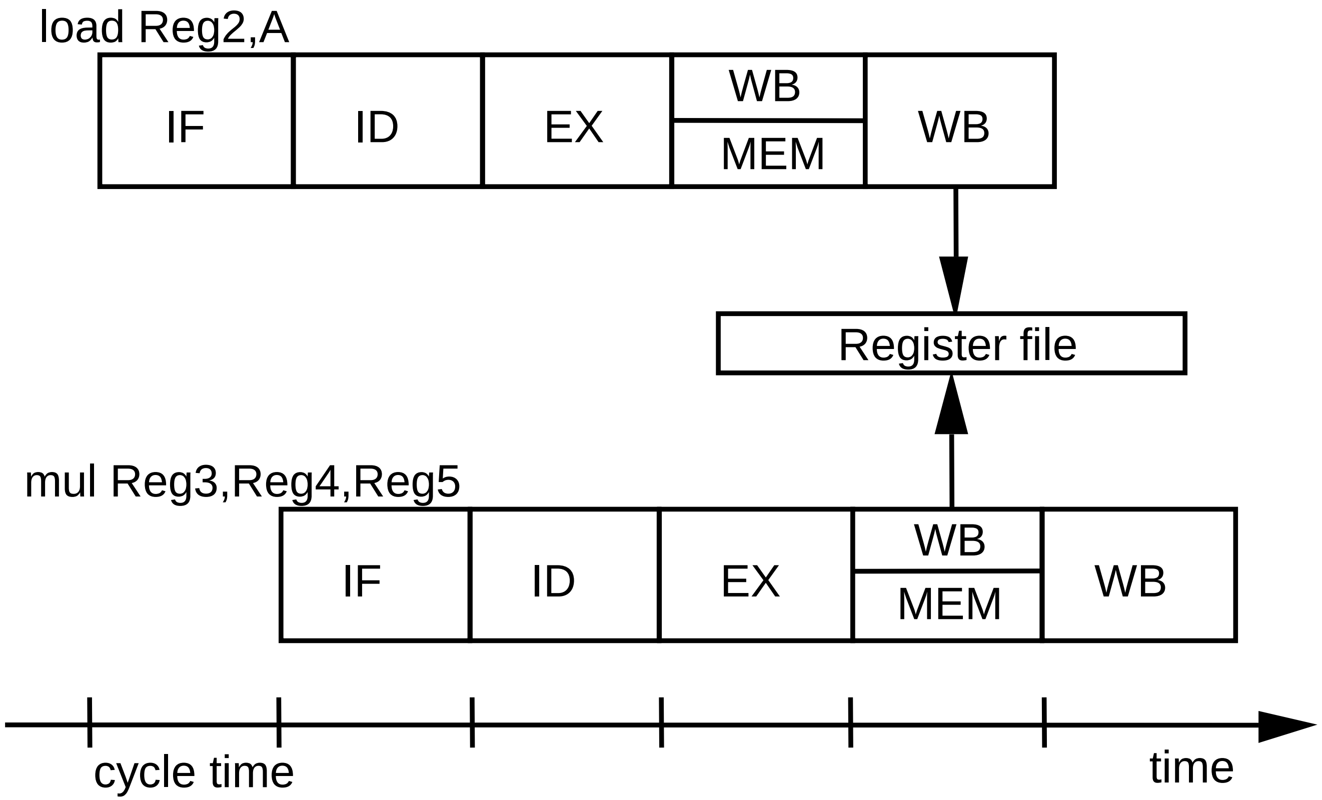
- For example, if the processor has only one register file write port and two instructions want to write in the register file at the same time.

Control hazards arise from branch, jump, and other control flow instructions

- For example, a taken branch interrupts the flow of instructions into the pipeline
  - ➡ the branch target must be fetched before the pipeline can resume execution.

Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline.

Pipeline bubble due to a structural hazard



Solutions to the structural hazard

Arbitration with interlocking: hardware that performs resource conflict arbitration and interlocks one of the competing instructions

Resource replication: In the example a register file with multiple write ports would enable simultaneous writes.

- However, now output dependencies may arise!
- Therefore, additional arbitration and interlocking necessary
- or the first (in program flow) value is discarded and the second used.

Types of Pipeline Hazards

CONTROL HAZARDS

Three types of pipeline hazards

Data hazards arise because of the unavailability of an operand ✓

- For example, an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.

Structural hazards may arise from some combinations of instructions that cannot be accommodated because of resource conflicts ✓

- For example, if the processor has only one register file write port and two instructions want to write in the register file at the same time.

Control hazards arise from branch, jump, and other control flow instructions

- For example, a taken branch interrupts the flow of instructions into the pipeline
  - ➡ the branch target must be fetched before the pipeline can resume execution.

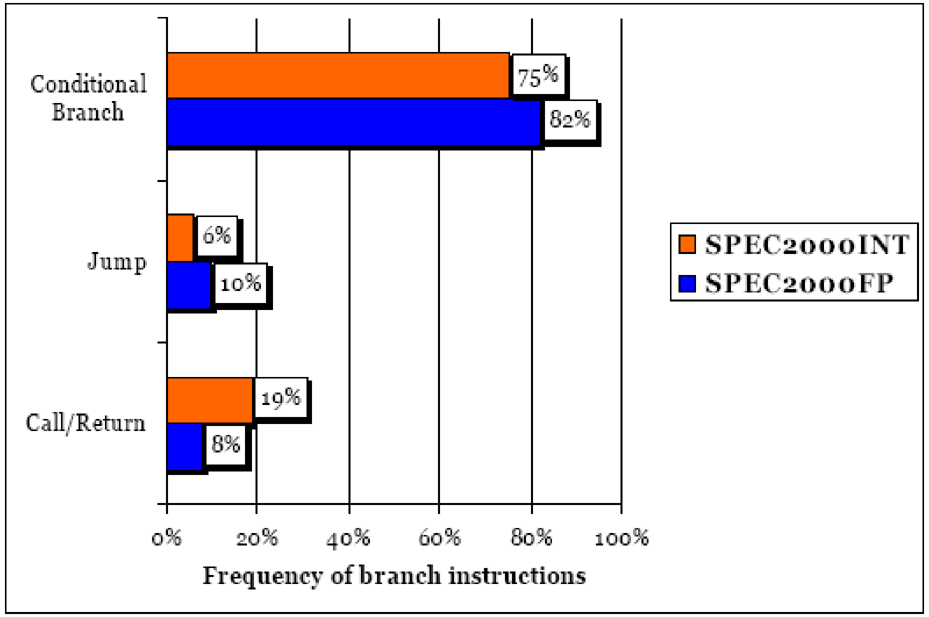
Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline.

Hazards due to control dependence

Conditional Jumps and branches stop linear program execution, program might continue elsewhere

Jump instruction normally detected in the Instruction Decode stage of the pipeline

- If a jump is detected, the pipeline already contains instructions, that are immediately behind this instruction



Source: H&P using Alpha

Hazards due to control dependence

Jumps are very common in programs

```
C Syntax

n=10
s=0
for(i=0; i<n; i++) {
  s = s + i;
}
…
```

```
MMIX Syntax
        LOC     #100
s       IS      $1
i       IS      $2
test    IS      $3
n       IS      10

Main    SETL    s,0
        SETL    i,0
For     ADD     s,s,i
        ADD     i,i,1
        SUB     test,i,n
        BNZ     test,For
        ...
        TRAP    0,Halt,0
```

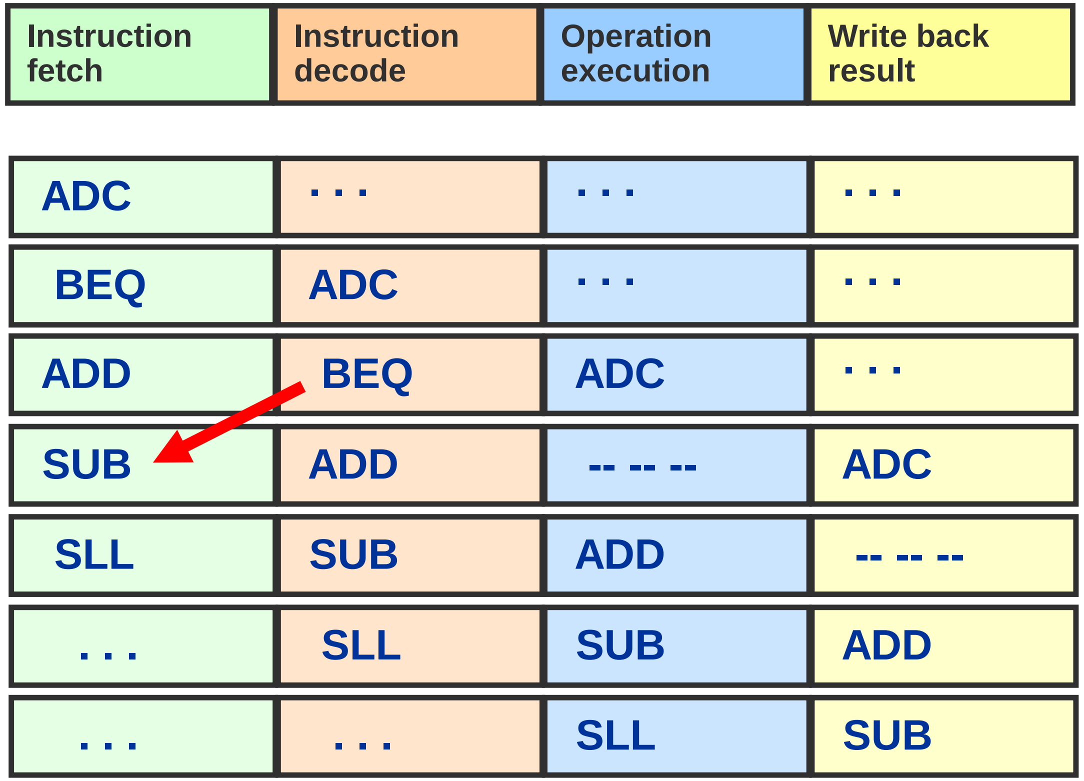
Example

```
        ADC   R4,R5,R4     ; R4 ← R4 + R5 + C
        CMP   R1,R2        ; R1 = R2 ?
        BEQ   Label        ; PC ← <Label Adresse>
        ADD   R3,R1,R2     ; R3 ← R1 + R2

Label:  SUB   R6,R4,R5     ; R6 ← R4 - R5
        SLL   R0           ; R0 ← shift_left(R0)
```

Example

```
           CMP    R1,R2
           ADC    R4,R5,R4
           BEQ    Label
           ADD    R3,R1,R2
Label:     SUB    R6,R4,R5
           SLL    R0
```



Instruction fetch	Instruction decode	Operation execution	Write back result
ADC	. . .	. . .	. . .
BEQ	ADC	. . .	. . .
ADD	BEQ	ADC	. . .
SUB	ADD	-- -- --	ADC
SLL	SUB	ADD	-- -- --
. . .	SLL	SUB	ADD
. . .	. . .	SLL	SUB

The ADD instruction is still in the pipeline and therefore executed before the jump is realized!

Solutions

Hardware Solutions

Pipeline Flushing

- Flush (empty) pipeline before realizing the jump

Speculative Branch

- In case of a conditional jump: Estimate result of condition and load pipeline (speculative)
- Wrong speculation: Pipeline Flushing
- Branch prediction used in most modern processors

Simple solutions

Hardware Interlocking

- This is the simplest way to deal with control hazards: the hardware must detect the branch and apply hardware interlocking to stall the next instruction(s).

Software Solutions

- Insertion of NOP instructions by the compiler after every branch
- Re-Ordering of instructions by the compiler
- Instead of NOPs, instructions that will be executed anyway and that do not influence the branch condition will be put into the pipeline immediately after the branch instruction (early days of RISC processors)

Solution: Decide branch direction earlier

Flushing or Locking is often not acceptable

Reordering is often not possible

Calculation of the branch direction and of the branch target address should be done in the pipeline as early as possible.

Best solution

- Already in ID stage after the instruction has become recognized as branch instruction.
- or even earlier: if history shows that address contains a branch

Branch Prediction

Branch prediction foretells the outcome of conditional branch instructions, excellent branch handling techniques are essential for today's and for future microprocessors.

IF stage finds a branch instruction

- predict branch direction

The branch delay slots are speculatively filled with instruction

- of the consecutively following path
- of the path at the target address

After resolving of the branch direction

- decide upon correctness of prediction

In case of misprediction ➡ discard wrongly fetched instructions

- rerolling when a branch is mispredicted is expensive:
  - 9 cycles on Itanium
  - 11 or more cycles in the Pentium II

Branch-Target Buffer or Branch-Target Address Cache

The Branch Target Buffer (BTB) or Branch-Target Address Cache (BTAC) stores branch and jump target addresses.

It should be known already in the IF stage whether the as-yet-undecoded instruction is a jump or branch.

The BTB is accessed during the IF stage.

The BTB consists of a table with branch addresses, the corresponding target addresses, and prediction information.

Variations

- Branch Target Cache (BTC): stores one or more target instructions additionally.
- Return Address Stack (RAS): a small stack of return addresses for procedure calls and returns is used additional to and independent of a BTB.

Branch-Target Buffer or Branch-Target Address Cache

Branch address	Target address	Prediction Bits

Two Basic Techniques of Branch Prediction

Static Branch Prediction

- The prediction direction for an individual branch remains always the same.

Dynamic Branch Prediction

- The prediction direction depends upon previous (the “history” of) branch executions.

Static Branch Prediction

The prediction direction for an individual branch remains always the same!

- The machine cannot dynamically alter the branch prediction (in contrast to dynamic branch prediction which is based on previous branch executions).

Static branch prediction comprises

- machine-fixed prediction (e.g. always predict taken)
- compiler-driven prediction.

If the prediction followed the wrong instruction path, then the wrongly fetched instructions must be squashed from the pipeline.

Static Branch Prediction - machine-fixed

Wired taken/not-taken prediction

- The static branch prediction can be wired into the processor by predicting that all branches will be taken (or all not taken).

Direction based prediction

- Backward branches are predicted to be taken and forward branches are predicted to be not taken
  ➡ helps for loops

Static Branch Prediction - compiler-based

Opcode bit in branch instruction allows the compiler to reverse the hardware prediction.

There are two approaches the compiler can use to statically predict which way a branch will go:

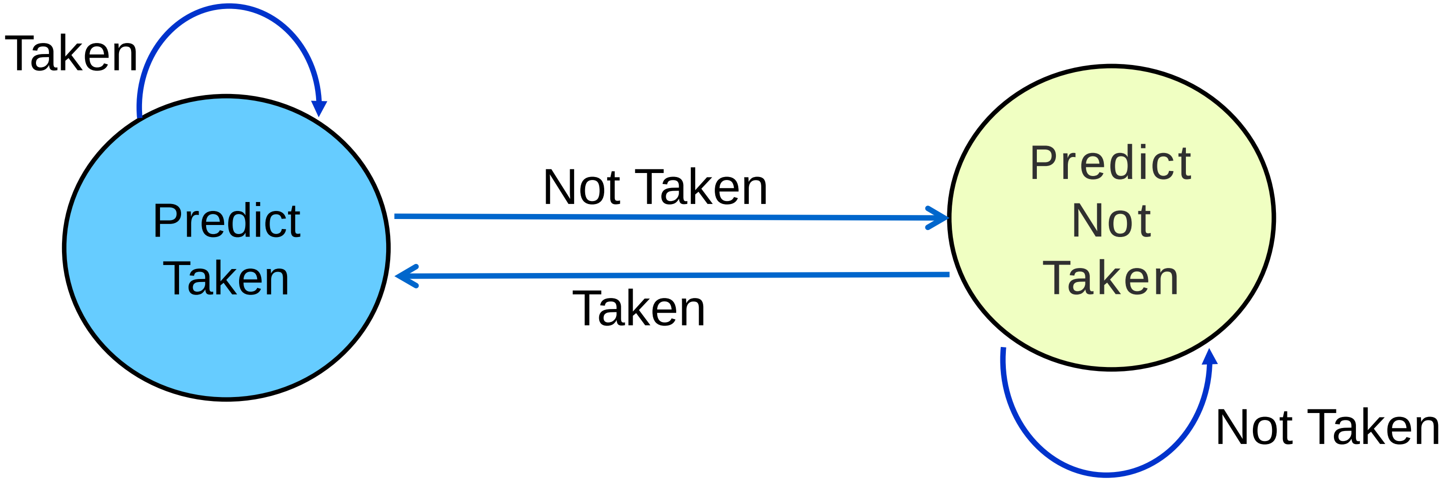
- it can examine the program code, or
- it can use profile information (collected from earlier runs)

Dynamic Branch Prediction

In dynamic branch prediction the prediction is decided on the computation history of the program execution.

In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.

Example: One-bit predictor



Formerly used: Alpha 21064 (1bit in instruction cache), Motorola PowerPC 604

One-bit vs. Two-bit Predictors

A one-bit predictor correctly predicts a branch at the end of a loop iteration, as long as the loop does not exit.

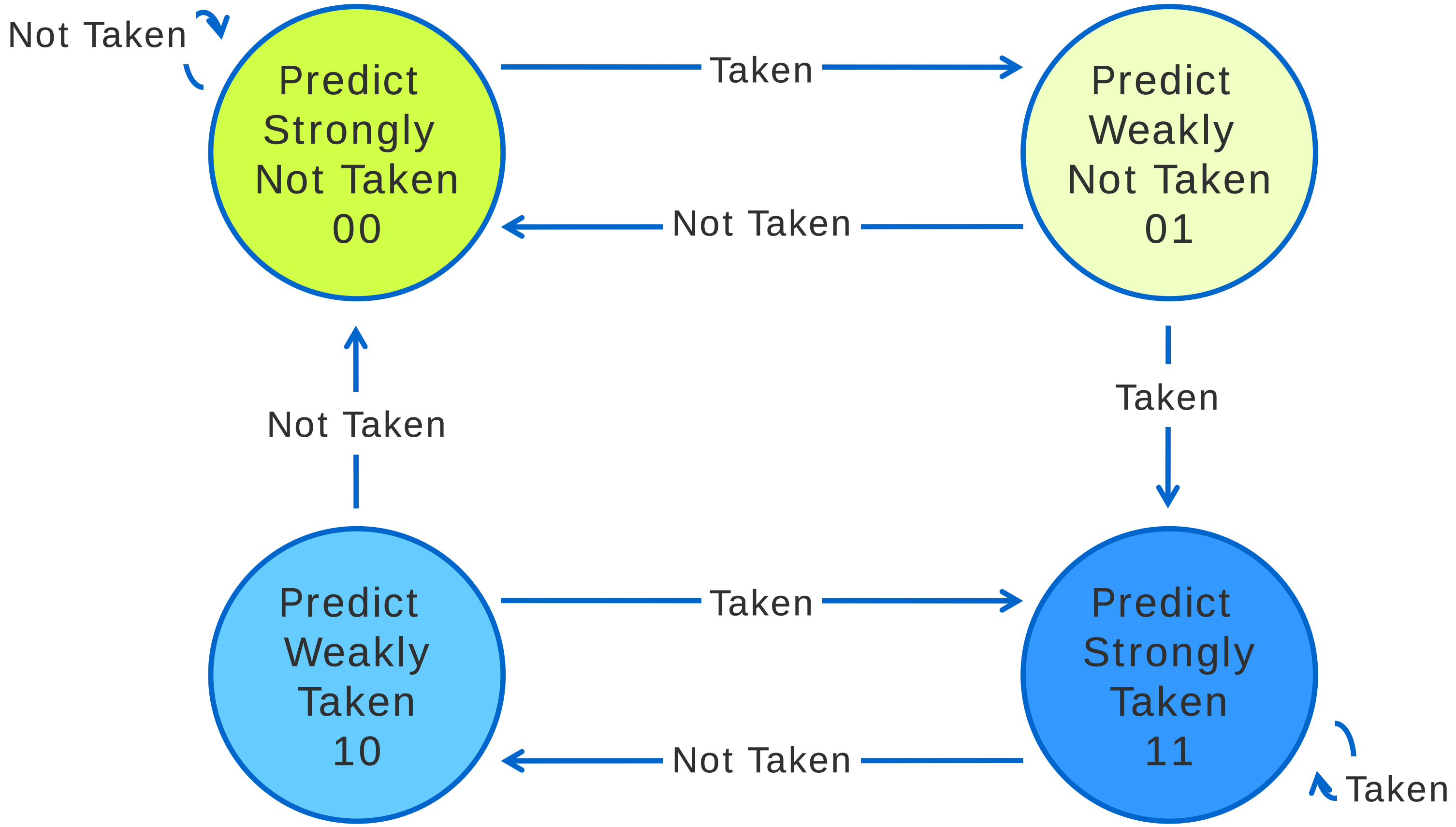
In nested loops, a one-bit prediction scheme will cause two mispredictions for the inner loop:

- One at the end of the loop, when the iteration exits the loop instead of looping again, and
- One when executing the first loop iteration, when it predicts exit instead of looping.

Such a double misprediction in nested loops is avoided by a two-bit predictor scheme.

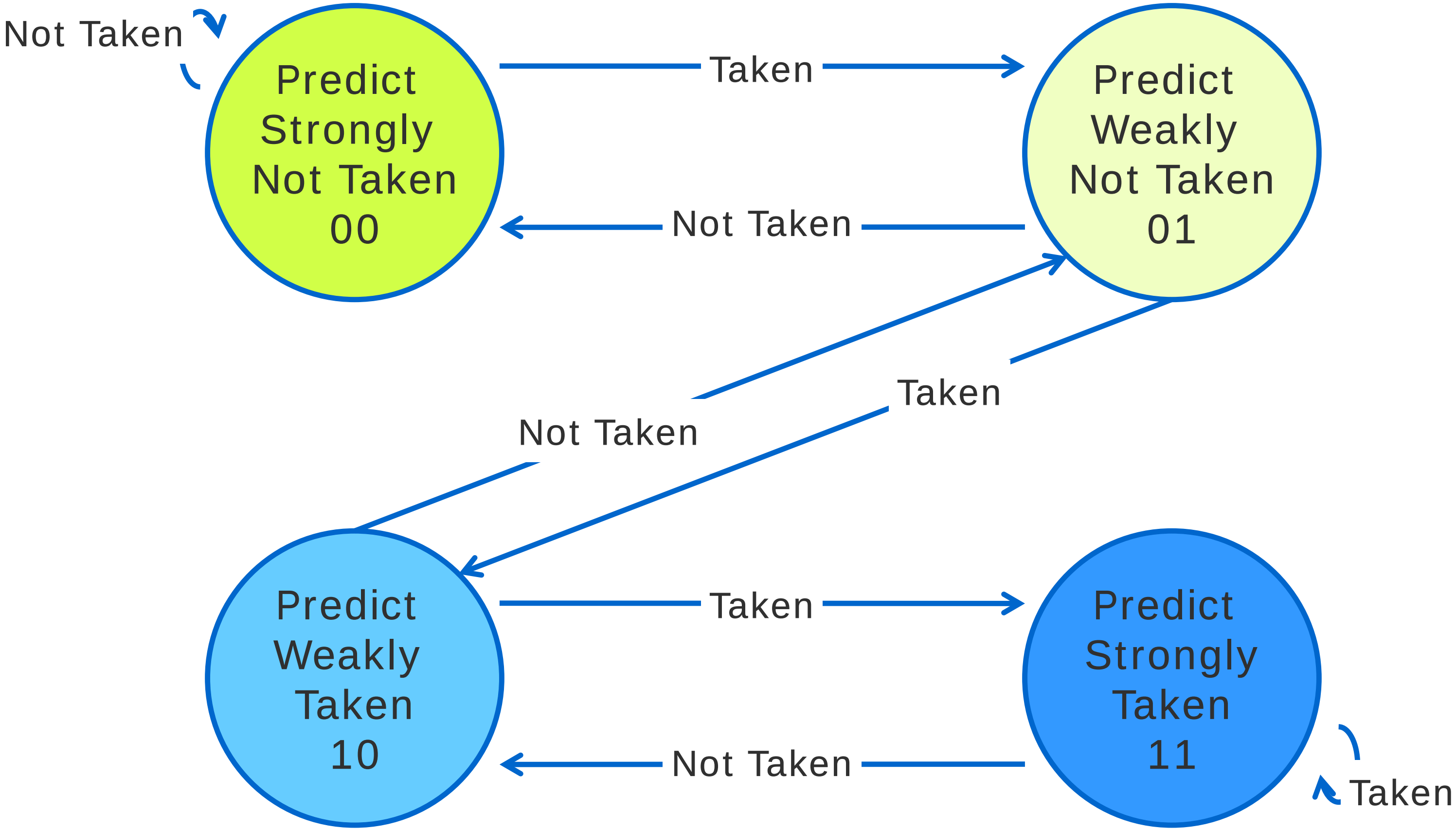
Two-bit Prediction**:** A prediction must miss twice before it is changed when a two-bit prediction scheme is applied.

Two-bit Predictors (Hysteresis Scheme)


Not Taken
Predict Strongly Not Taken 00
Taken
Predict Weakly Not Taken 01
Not Taken
Not Taken
Taken
Predict Weakly Taken 10
Taken
Predict Strongly Taken 11
Not Taken
Taken

Realization: Intel XScale, Sun UltraSPARC IIi

Two-bit Predictors
(Saturation Counter Scheme)


Not Taken
Predict Strongly Not Taken 00
Taken
Predict Weakly Not Taken 01
Not Taken
Not Taken
Taken
Predict Weakly Taken 10
Taken
Predict Strongly Taken 11
Not Taken
Taken

Predicated Instructions

Provide predicated or conditional instructions and one or more predicate registers.

Predicated instructions use a predicate register as additional input operand.

The Boolean result of a condition testing is recorded in a (one-bit) predicate register.

Predicated instructions are fetched, decoded, and placed in the instruction window like non predicated instructions.

Predication Example

```
if (x == 0) {                         /* branch b1 */
        a = b + c;
        d = e - f;
}
g = h * i;                            /* instruction independent of branch b1         */


(Pred = (x == 0) )                    /* branch b1: Pred is set to true if x equals 0 */
if Pred then a = b + c;               /* The operations are only performed            */
if Pred then d = e - f;               /* if Pred is set to true                       */

g = h * i;
```

Predication

Pro

- Able to eliminate a branch and therefore the associated branch prediction ➡ increasing the distance between mispredictions.
- The run length of a code block is increased ➡ better compiler scheduling.

Contra

- Predication affects the instruction set, adds a port to the register file, and complicates instruction execution.
- Predicated instructions that are discarded still consume processor resources; especially the fetch bandwidth.

Predication is most effective when control dependencies can be completely eliminated, such as in an `if-then` with a small `then` body.

The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence.

Branch handling techniques and implementations

Technique	Implementation examples
No branch prediction	Intel 8086
Static prediction	
always not taken	Intel i486
always taken	Sun SuperSPARC
backward taken, forward not taken	HP PA-7x00
semistatic with profiling	early PowerPCs
Dynamic prediction	
1-bit	DEC Alpha 21064, AMD K5
2-bit	PowerPC 604, MIPS R10000, Cyrix 6x86 and M2, NexGen 586
two-level adaptive	Intel PentiumPro, Pentium II, AMD K6
Hybrid prediction	DEC Alpha 21264
Predication	Intel/HP Merced, most DSPs, ARM processors, TI TMS320C6201, …
Eager execution (limited)	IBM mainframes: IBM 360/91, IBM 3090
Disjoint eager execution	none yet

Performance of branch handling techniques

Class	Technique	Rough Accuracy (Spec 89)
Static	always not taken	40%
	always taken	60%
	backward taken, forward not taken	65%
Software	Static analysis	70%
	Profiling	75%
Dynamic	1-bit	80%
	2-bit	93%
	two-level adaptive	95 – 97.5%

Adapted from: Dave Archer, Branch Prediction: Introduction and Survey, 2007

Pipelining basics: Summary

Hazards limit performance

- Structural hazards: need more HW resources
- Data hazards: need detection and forwarding
- Control hazards: early evaluation, delayed branch, prediction

Compilers may reduce cost of data and control hazards

- Compiler Scheduling
- Branch delay slots
- Static branch prediction

Increasing length of pipe increases impact of hazards

Pipelining helps instruction bandwidth, not latency

Multi-cycle operations (floating-point) and interrupts make pipelining harder

Questions & Tasks

- How can we resolve structural hazards?
- What are the limits of branch prediction? Think of multi-tasking, interrupts, multi-user etc.
- Why can compilers sometimes optimize branches better than the processor?
- A k-stage pipeline can result in a speed-up of k. So why not having very long pipelines?

Pipelining

VECTOR PIPELINING

Vektor Pipelining

Vector processor: a processor operating on a one-dimensional array of floating point numbers using a vector pipeline in the execution unit following the SIMD principle (single instruction, multiple data)

Vector = Array of floating point numbers (≠ math. vector!)

Operation on single values often called scalar processing.

Vector computers often contain scalar units besides many vector processing units – or CPUs may contain several vector units operating in parallel to scalar units.

Today, this approach is common in graphic processors but also standard CPUs (MMX, SSE, AVX, AltiVec, …)

Example: Addition

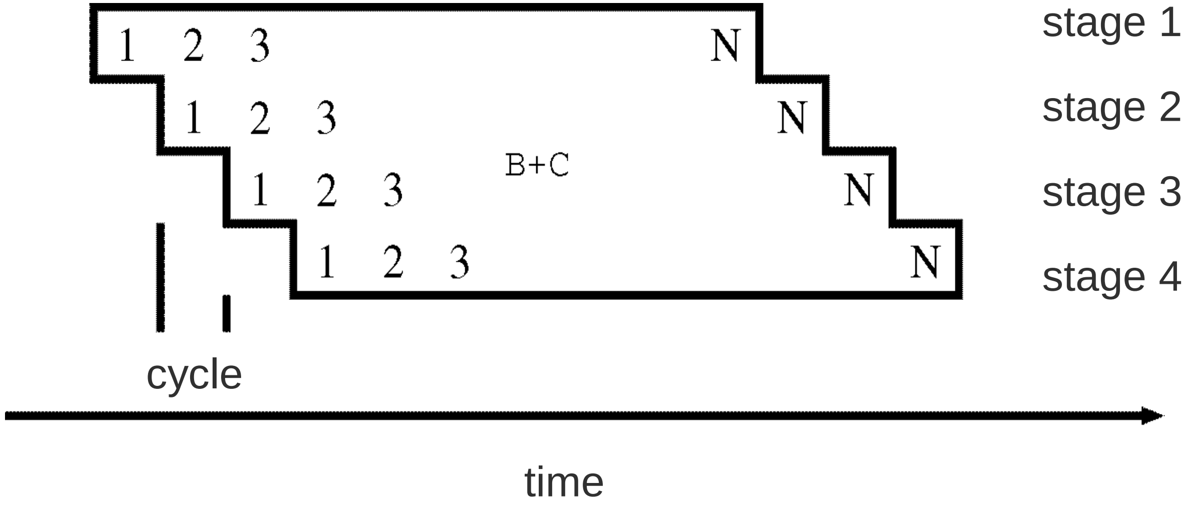
$$A(J) = B(J) + C(J), \qquad J = 1,2,\ldots,N$$

Component wise addition of the vectors B and C, i.e. the single floating point numbers of the arrays $B(1),\ldots,B(N)$ and $C(1),\ldots,C(N)$, with a single instruction and storing the result in the result vector A.

The execution of the operation is overlapping, i.e. first the operation of $B(1)+C(1)$ starts, then $B(2)+C(2)$etc.

Typical stages in the vector pipeline:

- Operand fetch
- Exponent alignment
- Shift of significand
- Addition
- Normalization
- Rounding
- Write back



Characteristics of vector pipelines

A single vector instruction allows the processing of two arrays of floating point numbers using this pipeline.

No separate address calculations for each operand needed (as this is the case for scalar units) – special hardware fetches the elements of an array from registers/memory.

Useful only for “longer” vectors as the pipeline need some time to fill before it produces a result per cycle.

- “long” depends on the architecture

No dependency checking needed between the operations as they are independent by definition (different elements of an array).

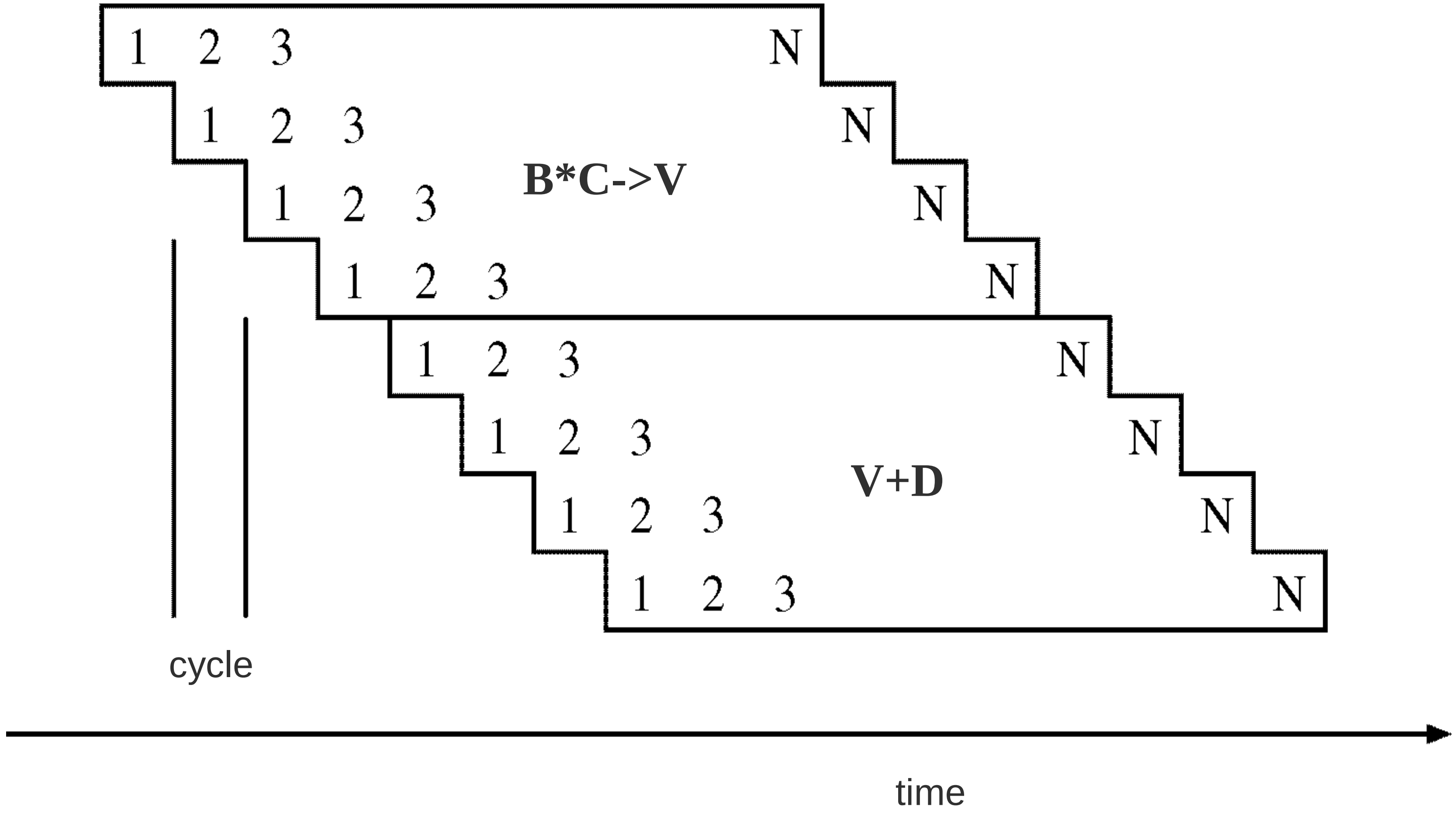
One can think of an enhanced EXE stage for floating point operations in CPUs.

Chaining of vector pipelines

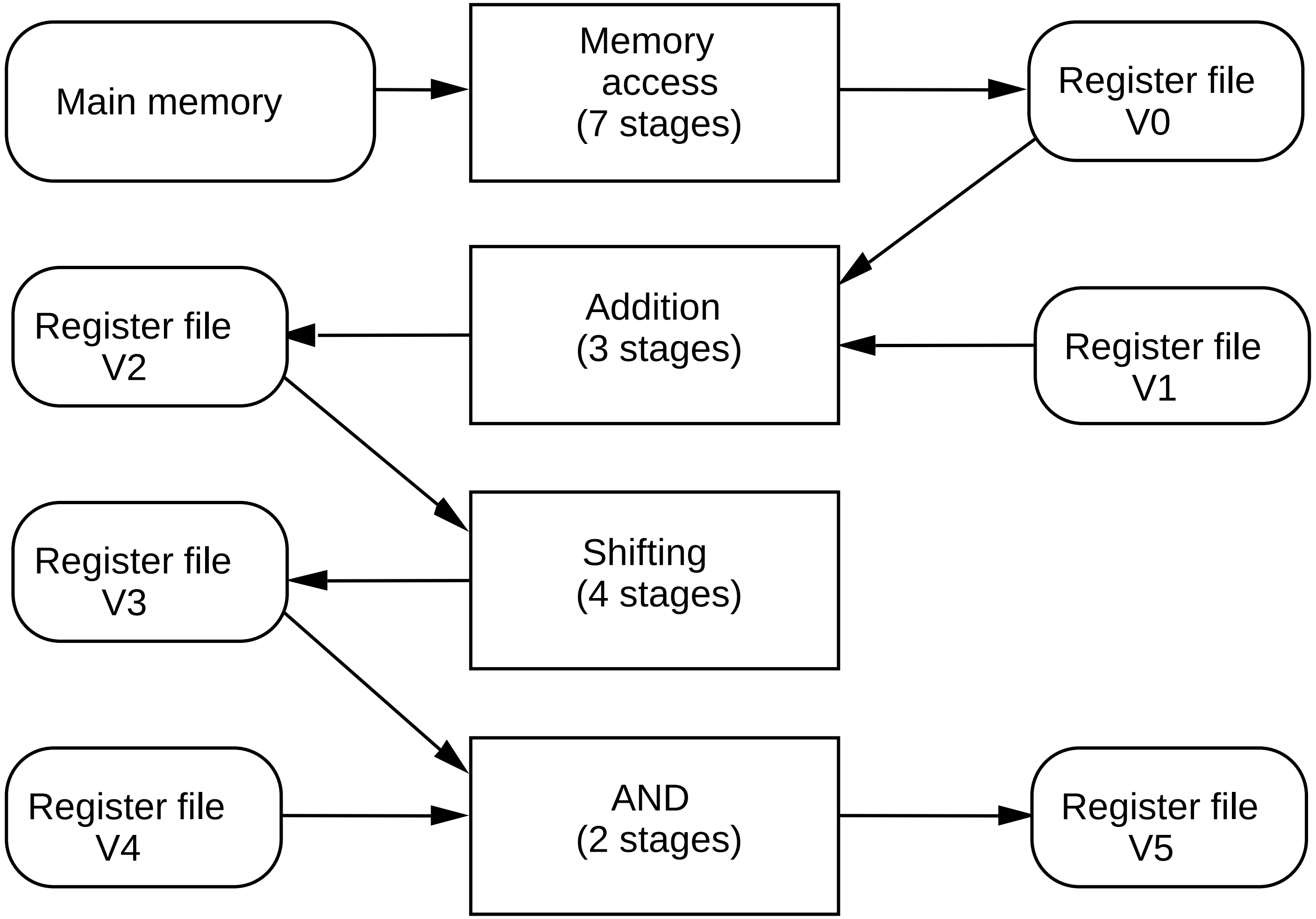
Chaining

- Apply the pipelining idea to a sequence of vector instructions.
- Forward the results of each element to the next pipeline immediately.

B(*J*)**C*(*J*)+*D*(*J*), *J*=1,2,...,*N*



History: chaining of 4 pipelines (Cray 1, 1976)



https://en.wikipedia.org/wiki/Cray-1

Simple Example

Component-wise multiplication of vectors

- A(i)= B(i) * C(i), i = 1, …, 100

Standard processor

```
i = 1
b = b(i)
c = c(i)
a = b * c
a(i) = a
i++
if i<101
```

Vector processor

```
vload b, 1, 100
vload c, 1, 100
vmult a, b, c
vstore a, 1, 100
```

Pipelining

SUPERSCALAR PROCESSORS

Superscalar processors

Definition

- Superscalar machines are distinguished by their ability to (dynamically) issue multiple instructions each clock cycle from a conventional linear instruction stream.

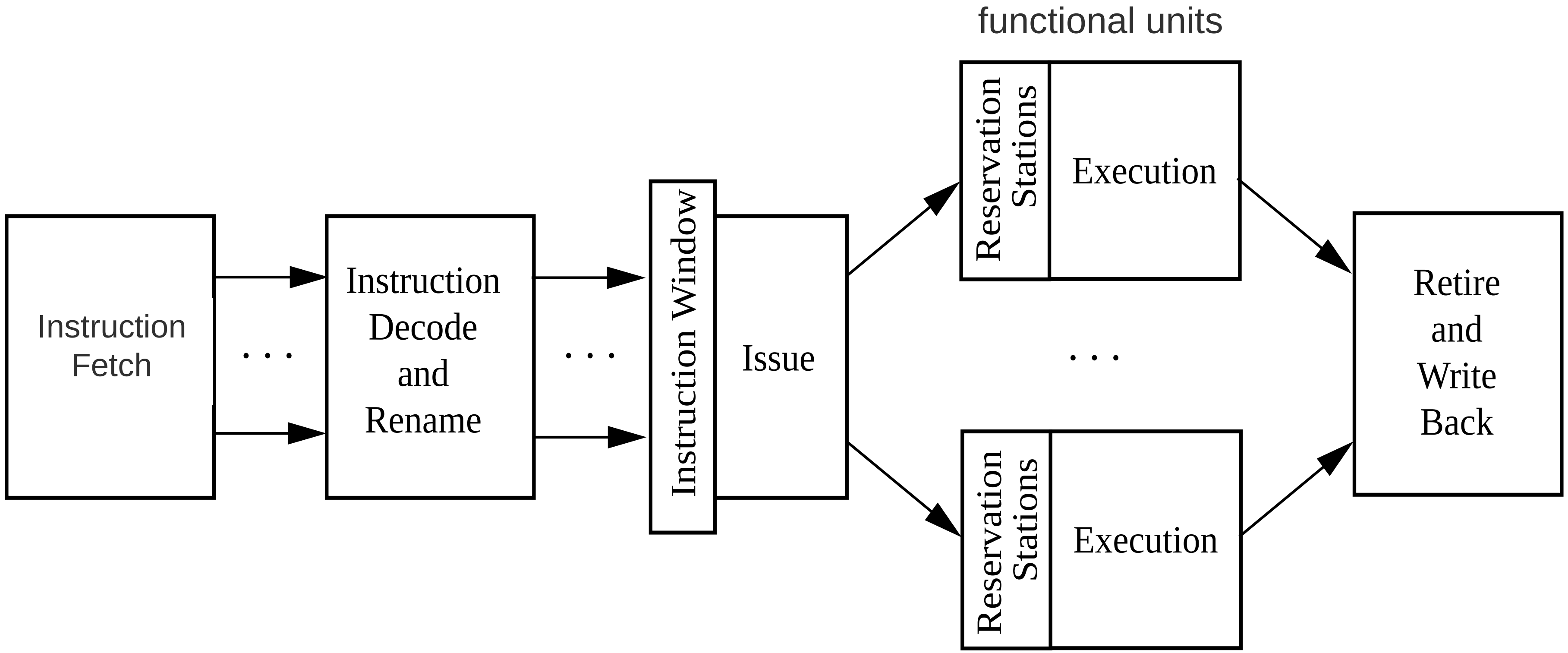
Consequence

- value of CPI (cycles per instructions) << 1.0 possible!

Superscalar Pipeline

Instructions in the instruction window are free from control dependences due to branch prediction, and free from name dependences due to register renaming.

So, only (true) data dependencies and structural conflicts remain to be solved.



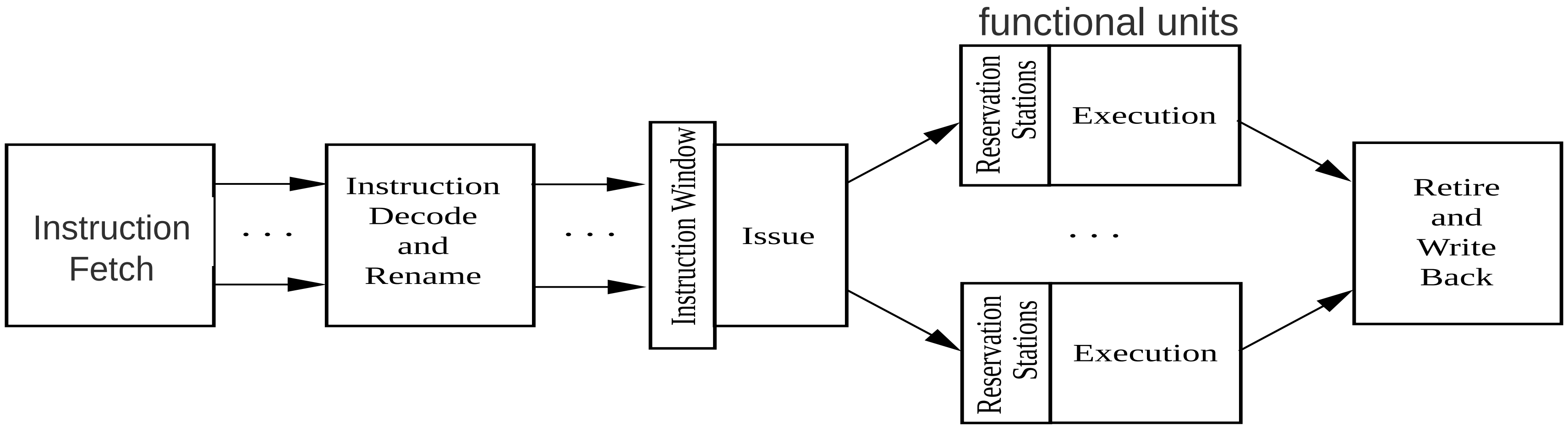
Sections of a Superscalar Pipeline

The ability to issue and execute instructions out-of-order partitions a superscalar pipeline in three distinct sections

- in-order section with the instruction fetch, decode and rename stages - the issue is also part of the in-order section in case of an in-order issue,

- out-of-order section starting with the issue in case of an out-of-order issue processor, the execution stage, and usually the completion stage, and again an

- in-order section that comprises the retirement and write-back stages.

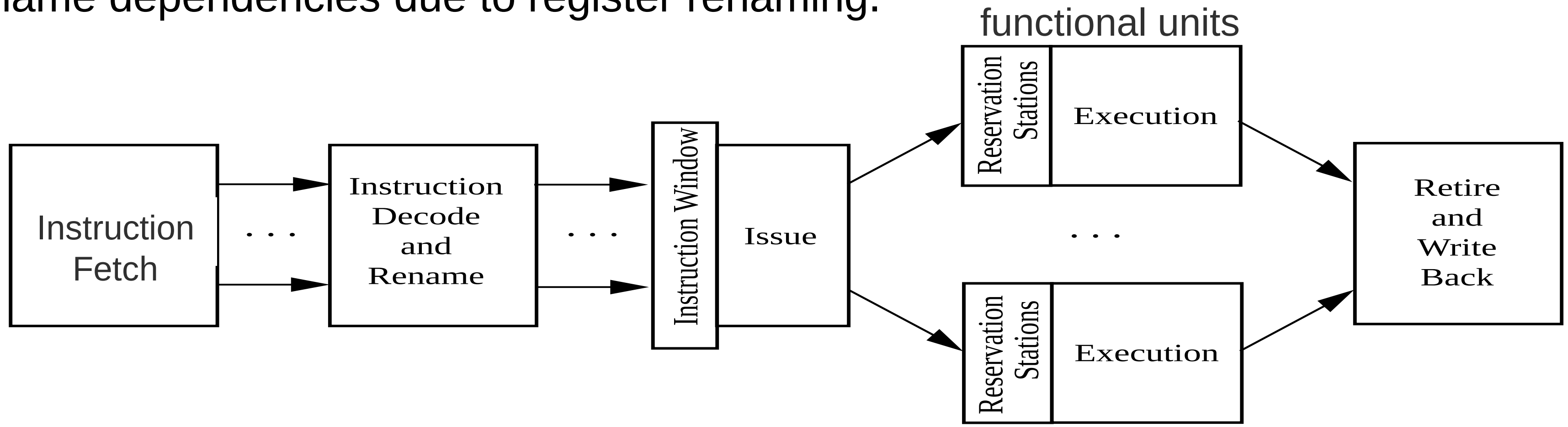


Fetch, decode, rename

Fetch
 - Get a bunch of commands

Decode
 - Decode the new instruction

Rename
 - Externally visible register are mapped to internal shadow registers
 - ➡ Avoid WAW/WAR-conflicts
 - ➡ Mapping stored in a rename map (Intel: alias table)
 - ➡ core execution units free from name dependencies due to register renaming.



Issue

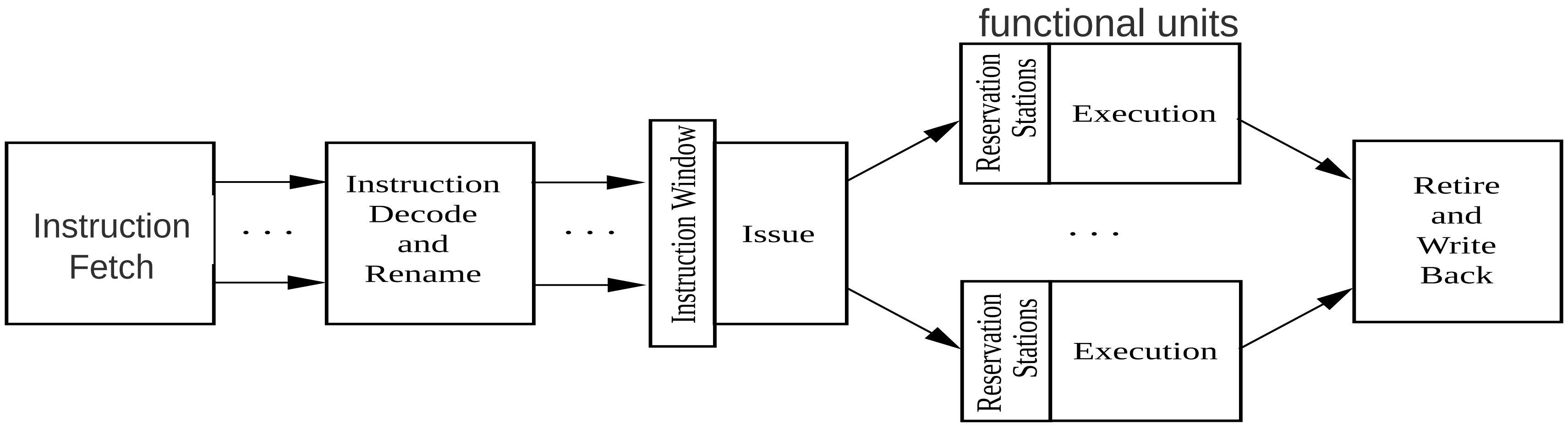
The issue logic examines the waiting instructions in the instruction window and simultaneously assigns (issues, dispatches) a number of instructions to the functional units (FUs) up to a maximum issue bandwidth.

Several instructions can be issued simultaneously (the issue bandwidth).

The program order of the issued instructions is stored in the reorder buffer.

Instruction issue from the instruction buffer can be:

- in-order (only in program order) or out-of-order

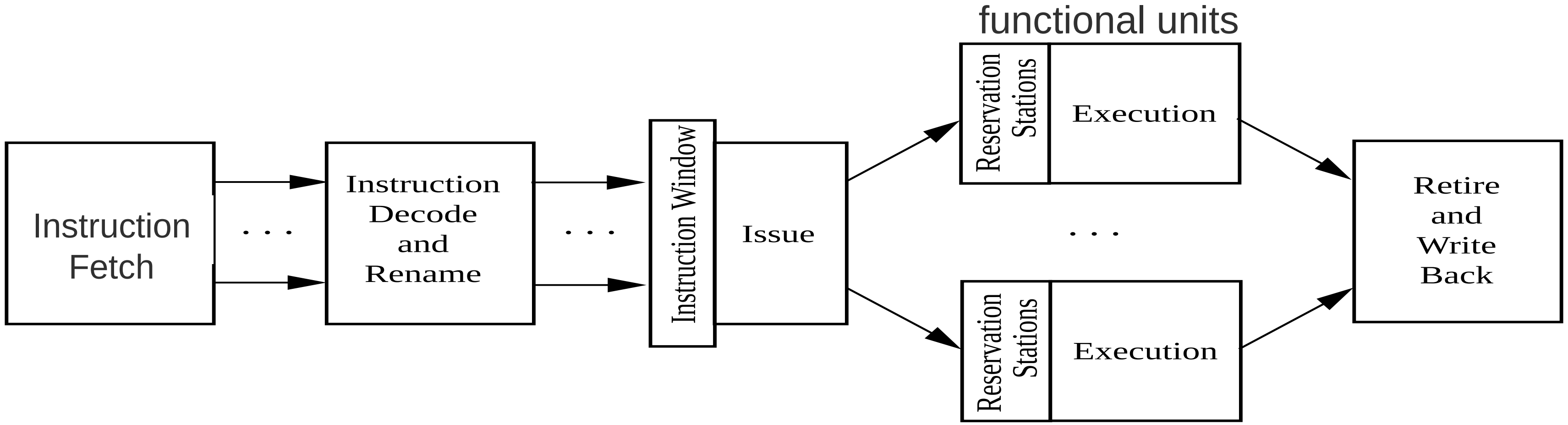


Reservation Station(s)

A reservation station is a buffer for a single instruction with its operands.

Reservation stations can be central to a number of FUs
or each FU has one or more own reservation stations.

Instructions await their operands in reservation stations.



Dispatch

An instruction is said to be dispatched from a reservation station to the FU when all operands are available, and execution starts.

If all its operands are available during issue and the FU is not busy, an instruction is immediately dispatched, starting execution in the next cycle after the issue.

So, the dispatch is usually not a pipeline stage.

An issued instruction may stay in the reservation station for zero to several cycles.

Dispatch and execution is performed *out of program order*.

Other authors interchange the meaning of *issue* and *dispatch* or use different semantic.

Completion

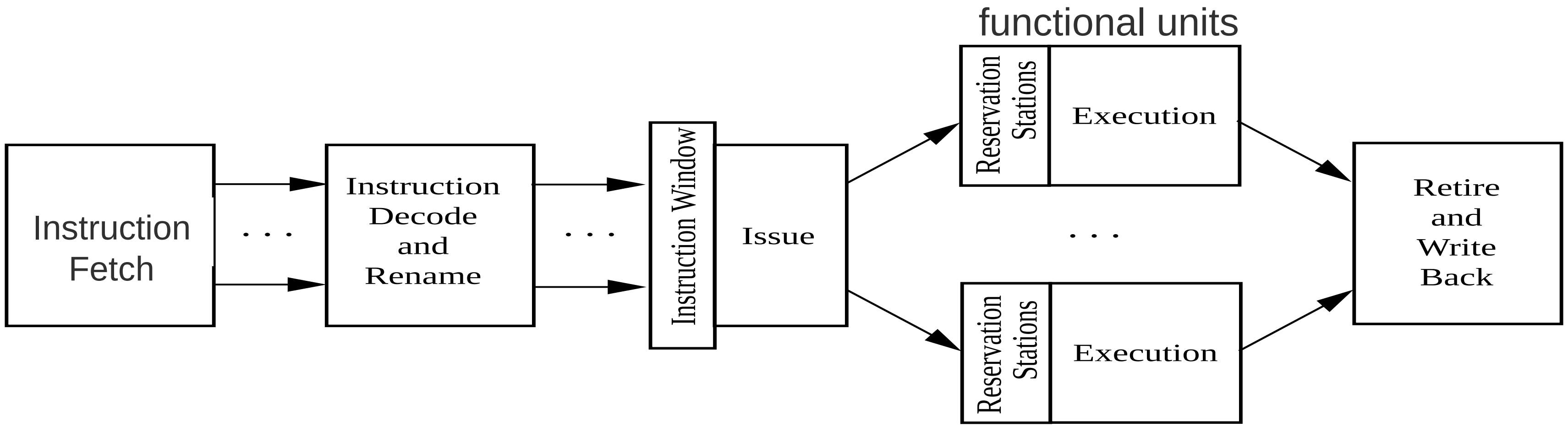
When the FU finishes the execution of an instruction and the result is ready for forwarding and buffering, the instruction is said to complete.

Instruction completion is out of program order.

During completion the reservation station is freed and the state of the execution is noted in the reorder buffer.

The state of the reorder buffer entry can denote an interrupt occurrence.

The instruction can be completed and still be speculatively assigned, which is also monitored in the reorder buffer.



Commitment

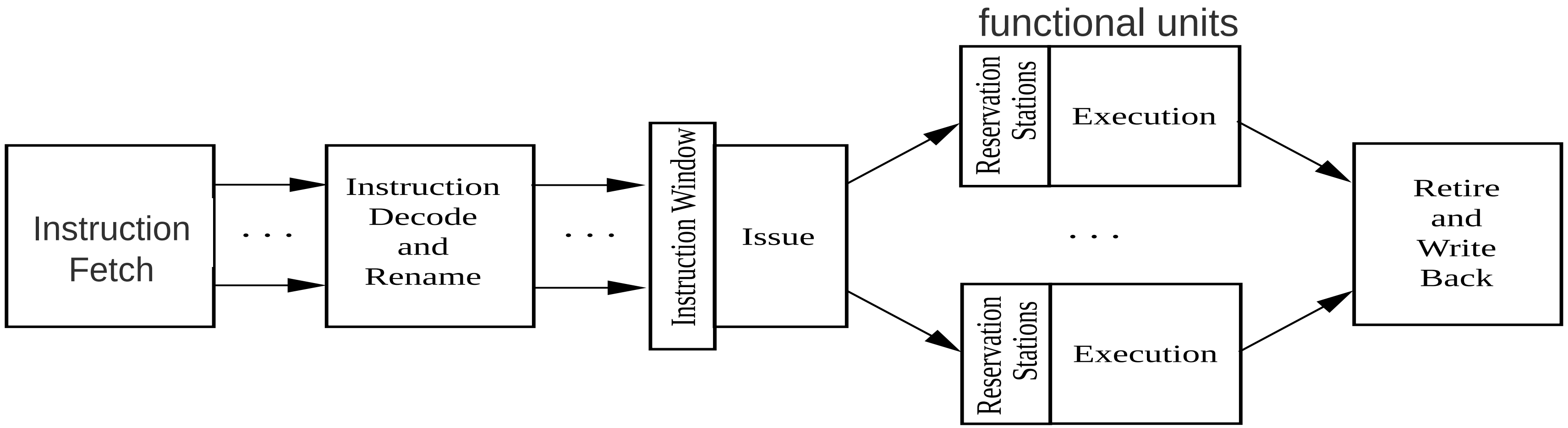
After completion, operations are committed in-order.

An instruction can be committed:

- if all previous instructions due to the program order are already committed or can be committed in the same cycle,
- if no interrupt occurred before and during instruction execution, and
- if the instruction is no more on a speculative path.

By or after commitment, the result of an instruction is made permanent in the architectural register set,

- usually by writing the result back from the rename register to the architectural register.



Precise Interrupt / Precise Exception

If an interrupt occurred, all instructions that are in program order before the interrupt signaling instruction are committed, and all later instructions are removed.

Precise exception means that all instructions before the faulting instruction are committed and those after it can be restarted from scratch.

Depending on the architecture and the type of exception, the faulting instruction should be committed or removed without any lasting effect.

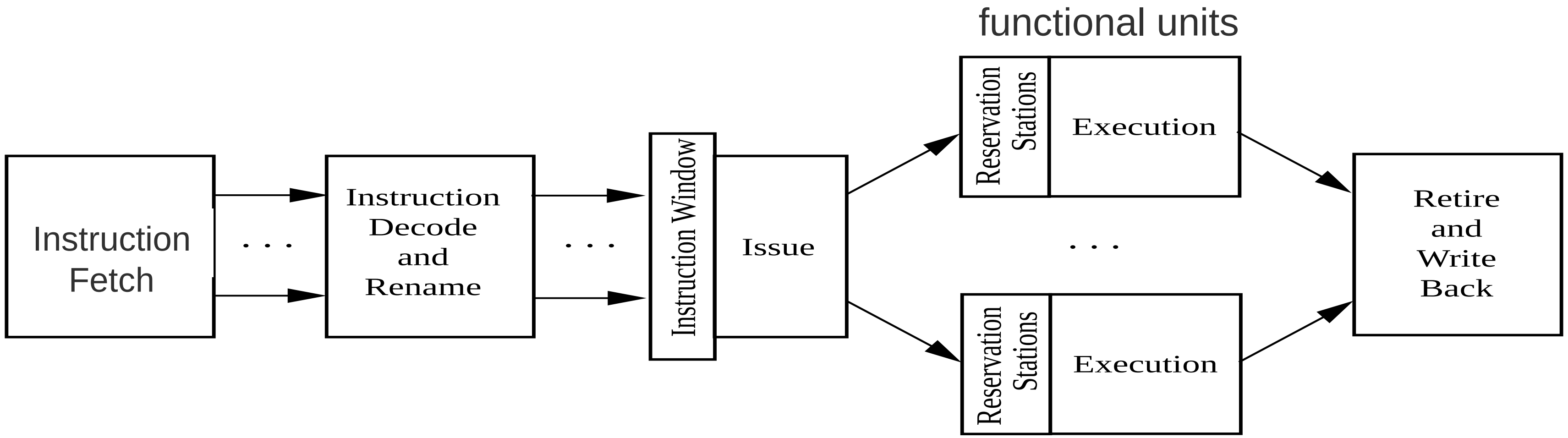
Retirement

An instruction retires when the reorder buffer slot of an instruction is freed either

- because the instruction commits (the result is made permanent) or
- because the instruction is removed (without making permanent changes).

A result is made permanent by copying the result value from the rename register to the architectural register.

- This is often done in an own stage after the commitment of the instruction with the effect that the rename register is freed one cycle after commitment.



Questions & Tasks

- Where do we have vector pipelines today?
- What makes vector pipelines so efficient?
- How do super scalar pipelines avoid WAR and WAW hazards?
- What does the programmer or compiler see from the super scalar pipeline or renaming registers? How is the ISA affected?
- Where does the vector pipeline fit into a super scalar pipeline?

Example: Dynamic Scheduling for an FP-Unit using Tomasulo’s Algorithm

Roberto Tomasulo, IBM, 1967 (https://en.wikipedia.org/wiki/Tomasulo_algorithm)

Key features: hardware support for register renaming, all EXE units have a reservation station, a common data bus (CDB) broadcasts results to all reservation stations

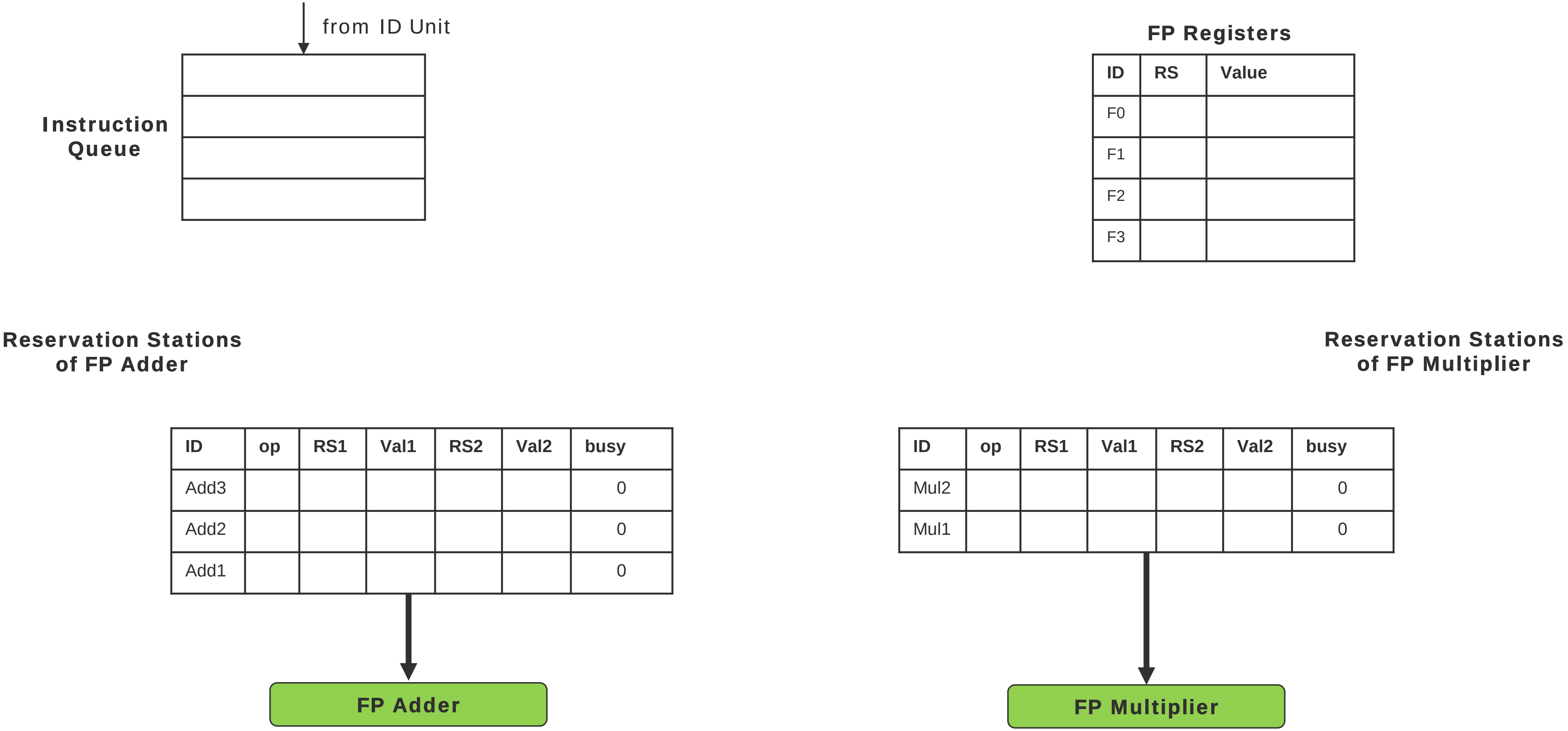
Common in many of today’s processors

Simplifications in the following example:

- Single Issue Processor (in-order)
- Instruction Queue can hold 4 instructions from ID-Unit
- up to 3 instructions can be fetched and decoded in parallel
- 1 FP Adder with 3 Reservation Stations
- 1 FP Multiplier with 2 Reservation Stations
- no speculative execution (→ no Reorder Buffer needed)

- only 4 FP registers

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



from ID Unit

Instruction Queue

FP Registers

ID	RS	Value
F0		
F1		
F2		
F3		

Reservation Stations of FP Adder

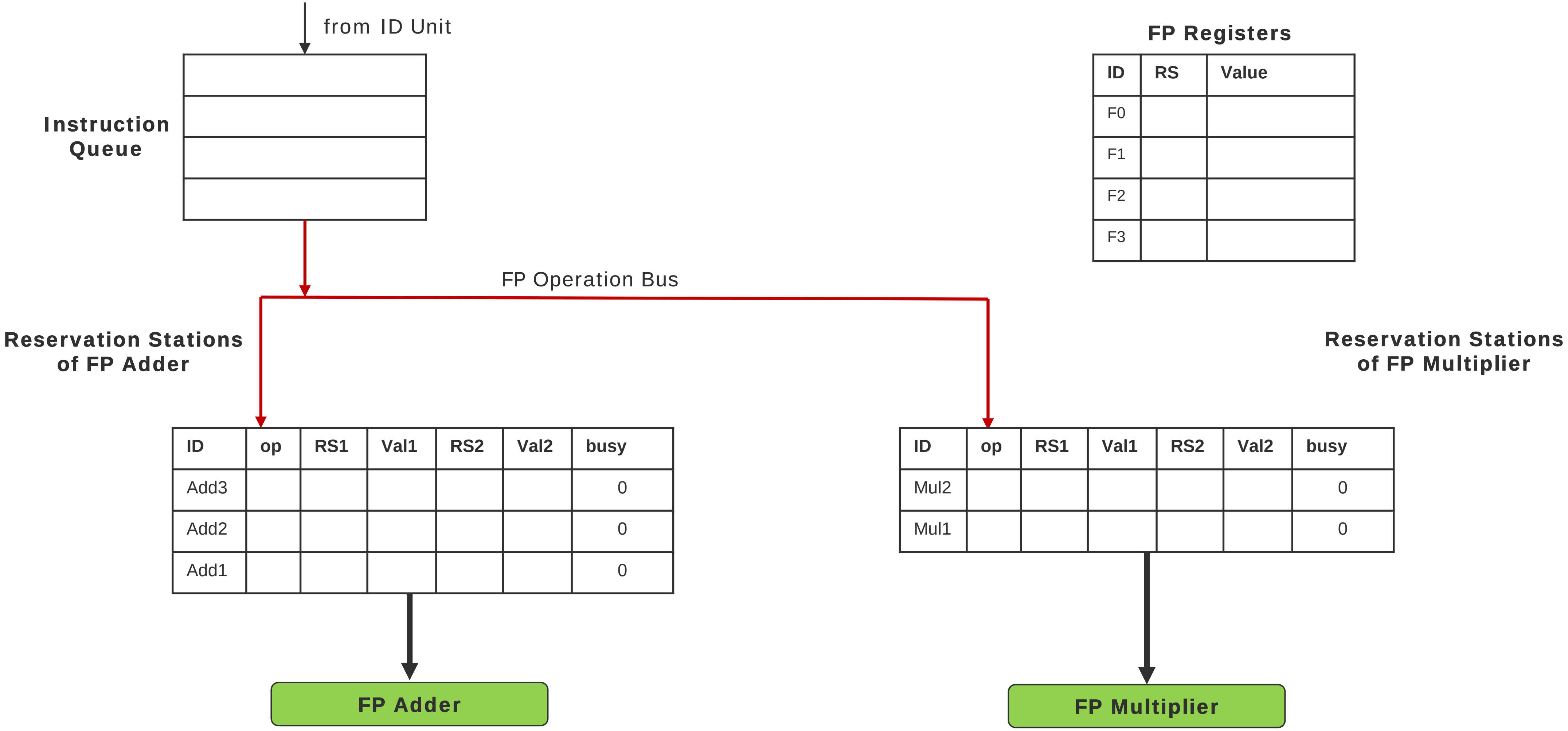
ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Source: Ben Juurlink, TU Berlin, lecture slides for „Advanced Computer Architectures“, 2015

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



FP Registers

ID	RS	Value
F0		
F1		
F2		
F3		

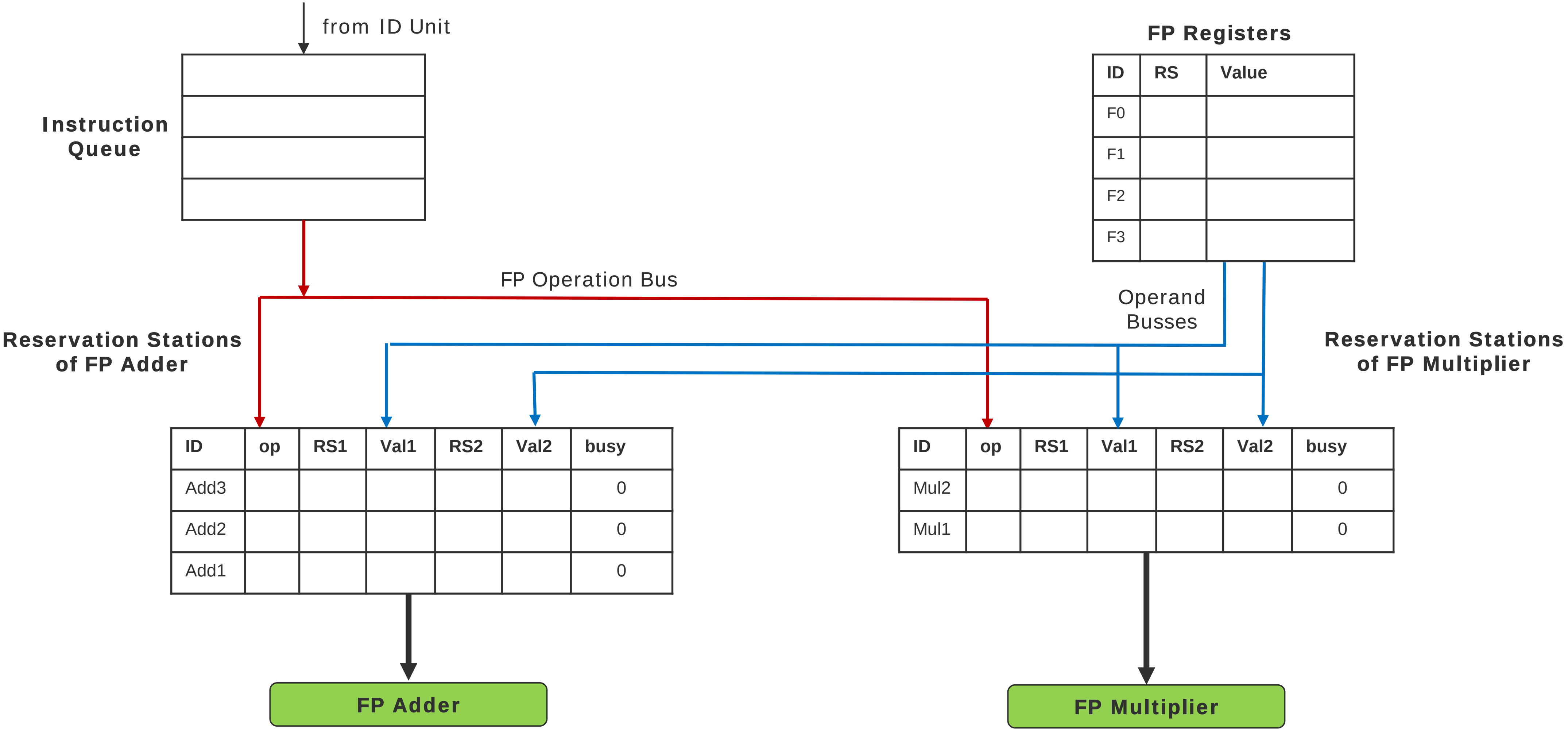
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Example: Dynamic Scheduling for a FP-Unit using Tomasulo's Algorithm



FP Registers

ID	RS	Value
F0		
F1		
F2		
F3		

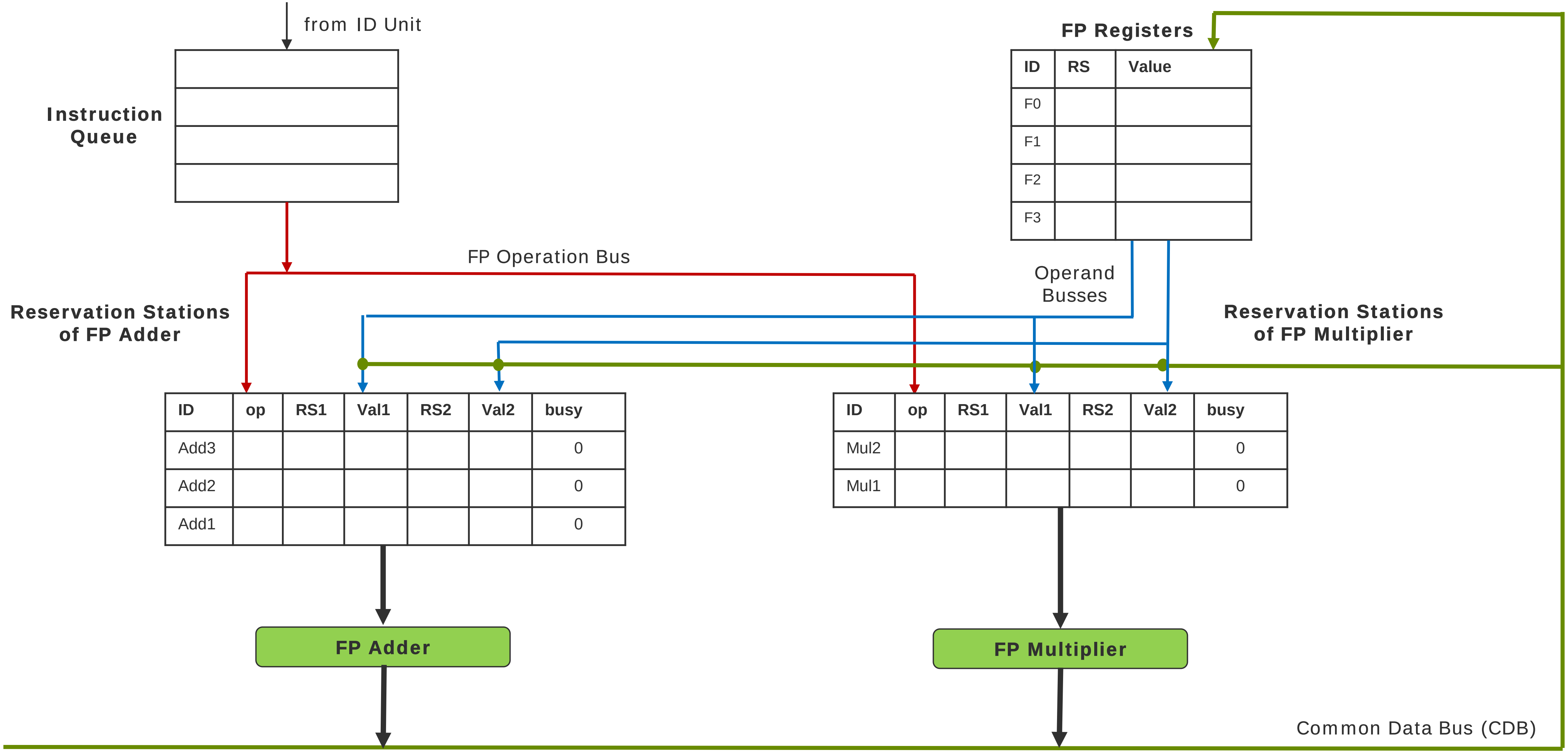
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



from ID Unit

Instruction Queue

FP Registers

ID	RS	Value
F0		
F1		
F2		
F3		

FP Operation Bus

Operand Busses

Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

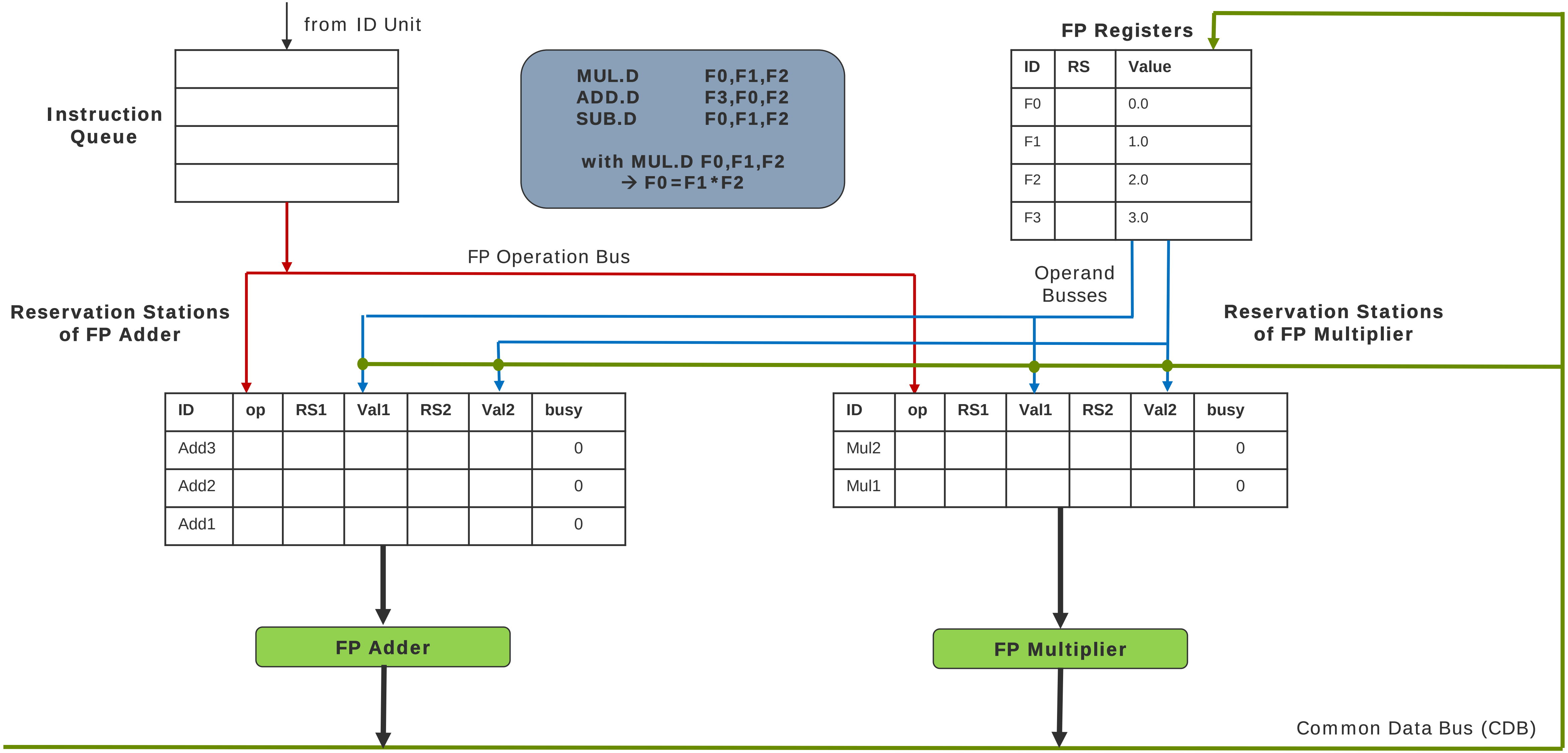
ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

FP Adder

FP Multiplier

Common Data Bus (CDB)

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



```
MUL.D      F0,F1,F2
ADD.D      F3,F0,F2
SUB.D      F0,F1,F2
```

with MUL.D F0,F1,F2
→ F0=F1*F2

FP Registers

ID	RS	Value
F0		0.0
F1		1.0
F2		2.0
F3		3.0

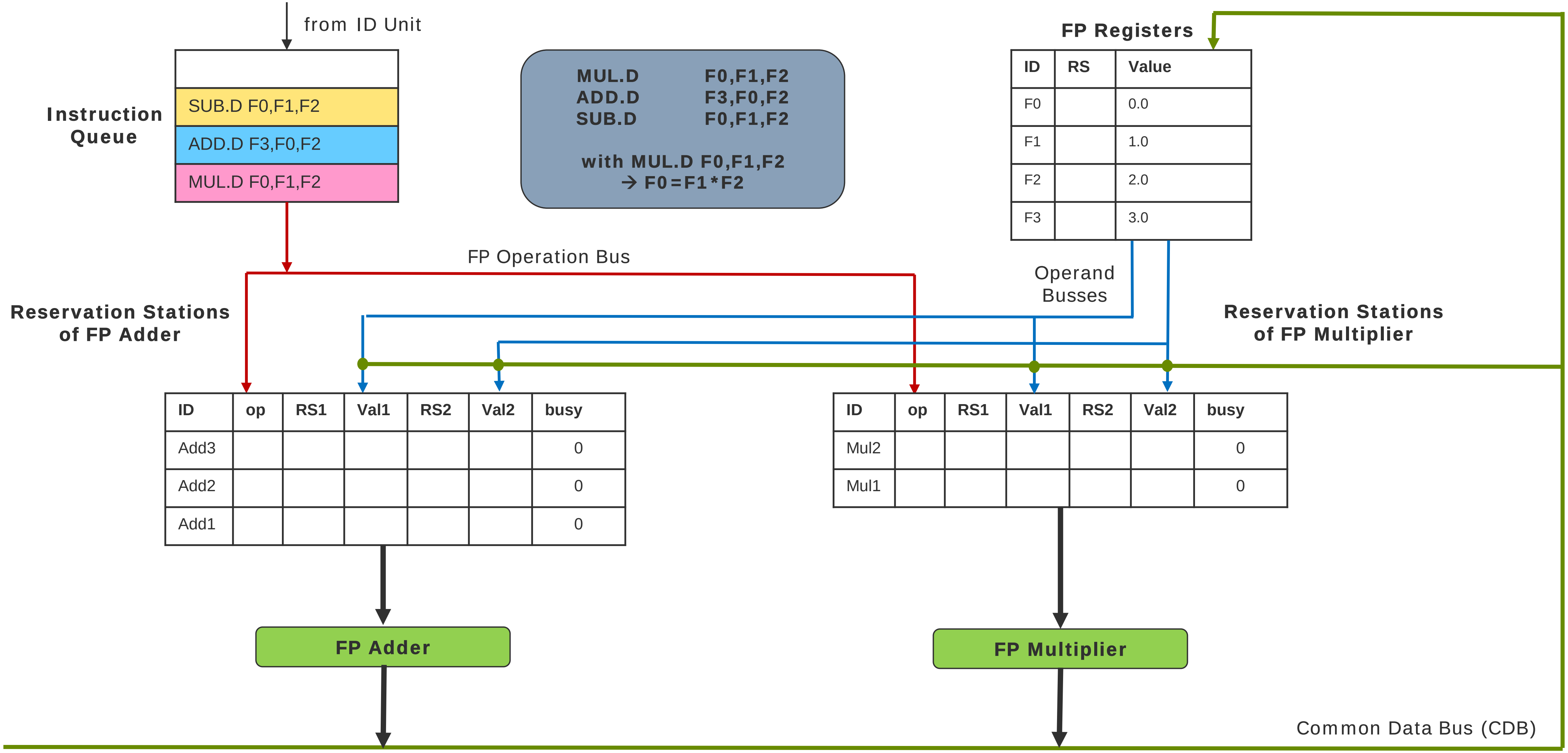
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



Instruction Queue

SUB.D F0,F1,F2
ADD.D F3,F0,F2
MUL.D F0,F1,F2

```
MUL.D    F0,F1,F2
ADD.D    F3,F0,F2
SUB.D    F0,F1,F2

with MUL.D F0,F1,F2
→ F0=F1*F2
```

FP Registers

ID	RS	Value
F0		0.0
F1		1.0
F2		2.0
F3		3.0

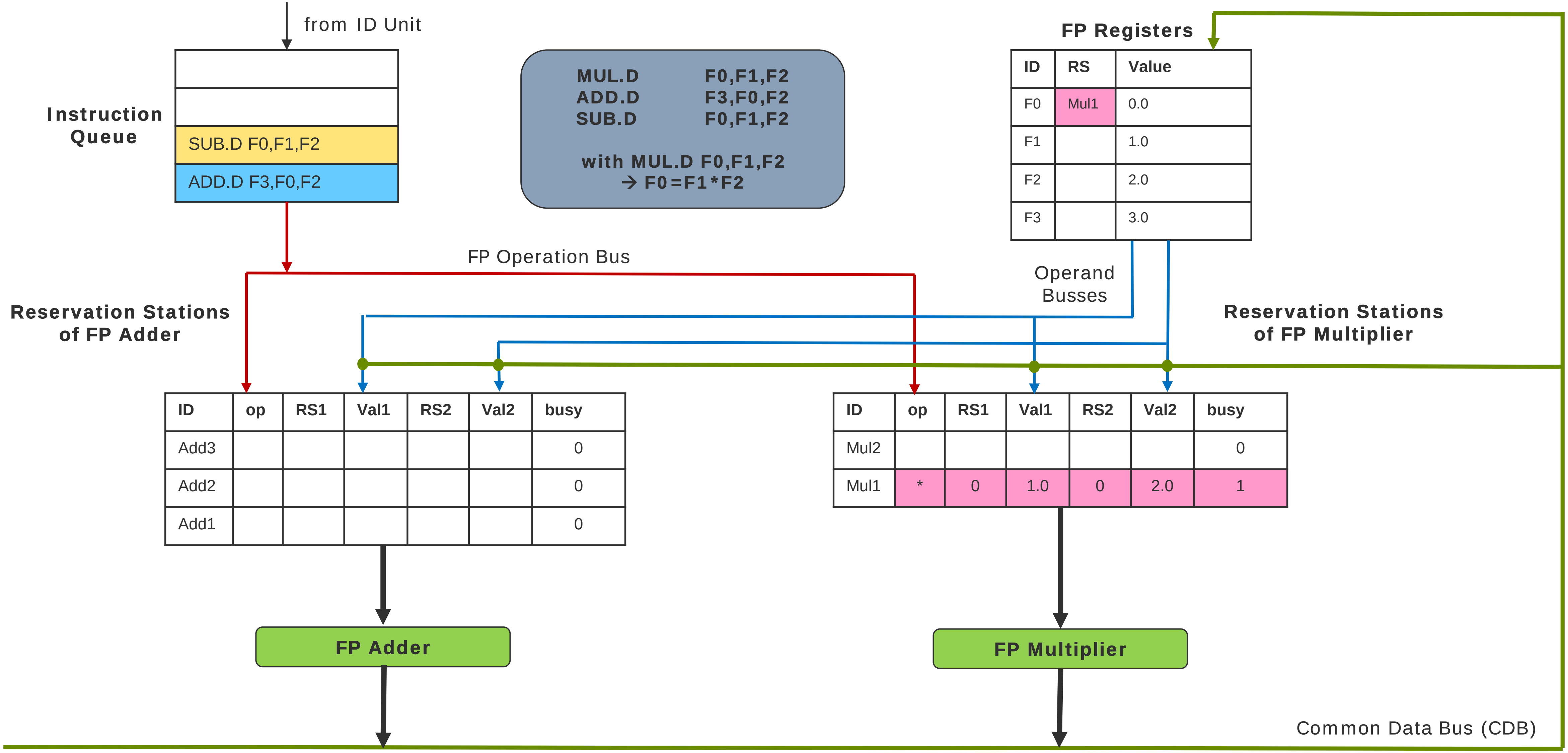
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



Instruction Queue

SUB.D F0,F1,F2
ADD.D F3,F0,F2

```
MUL.D        F0,F1,F2
ADD.D        F3,F0,F2
SUB.D        F0,F1,F2

with MUL.D F0,F1,F2
→ F0=F1*F2
```

FP Registers

ID	RS	Value
F0	Mul1	0.0
F1		1.0
F2		2.0
F3		3.0

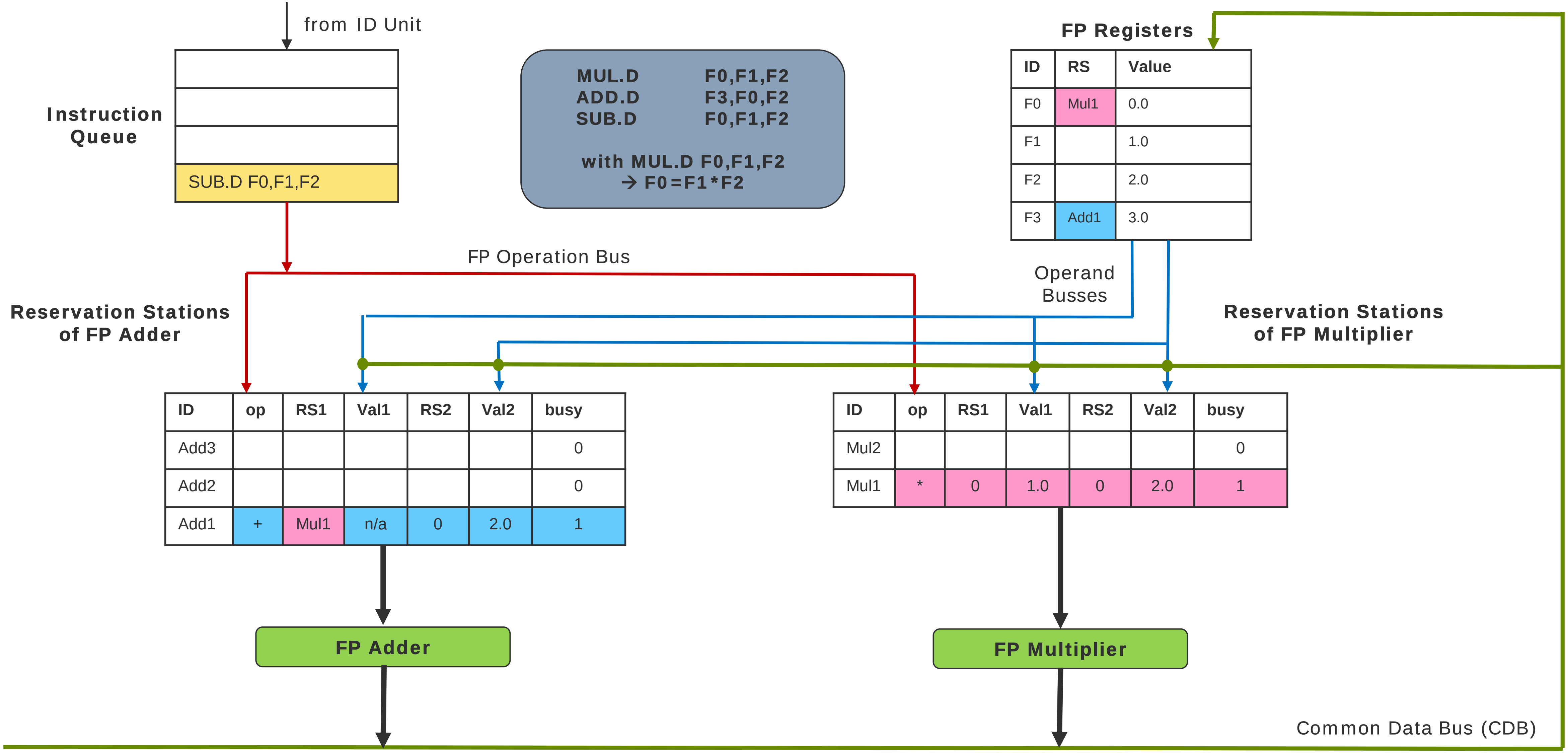
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1	*	0	1.0	0	2.0	1

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



Instruction Queue

SUB.D F0,F1,F2

```
MUL.D    F0,F1,F2
ADD.D    F3,F0,F2
SUB.D    F0,F1,F2

with MUL.D F0,F1,F2
→ F0=F1*F2
```

FP Registers

ID	RS	Value
F0	Mul1	0.0
F1		1.0
F2		2.0
F3	Add1	3.0

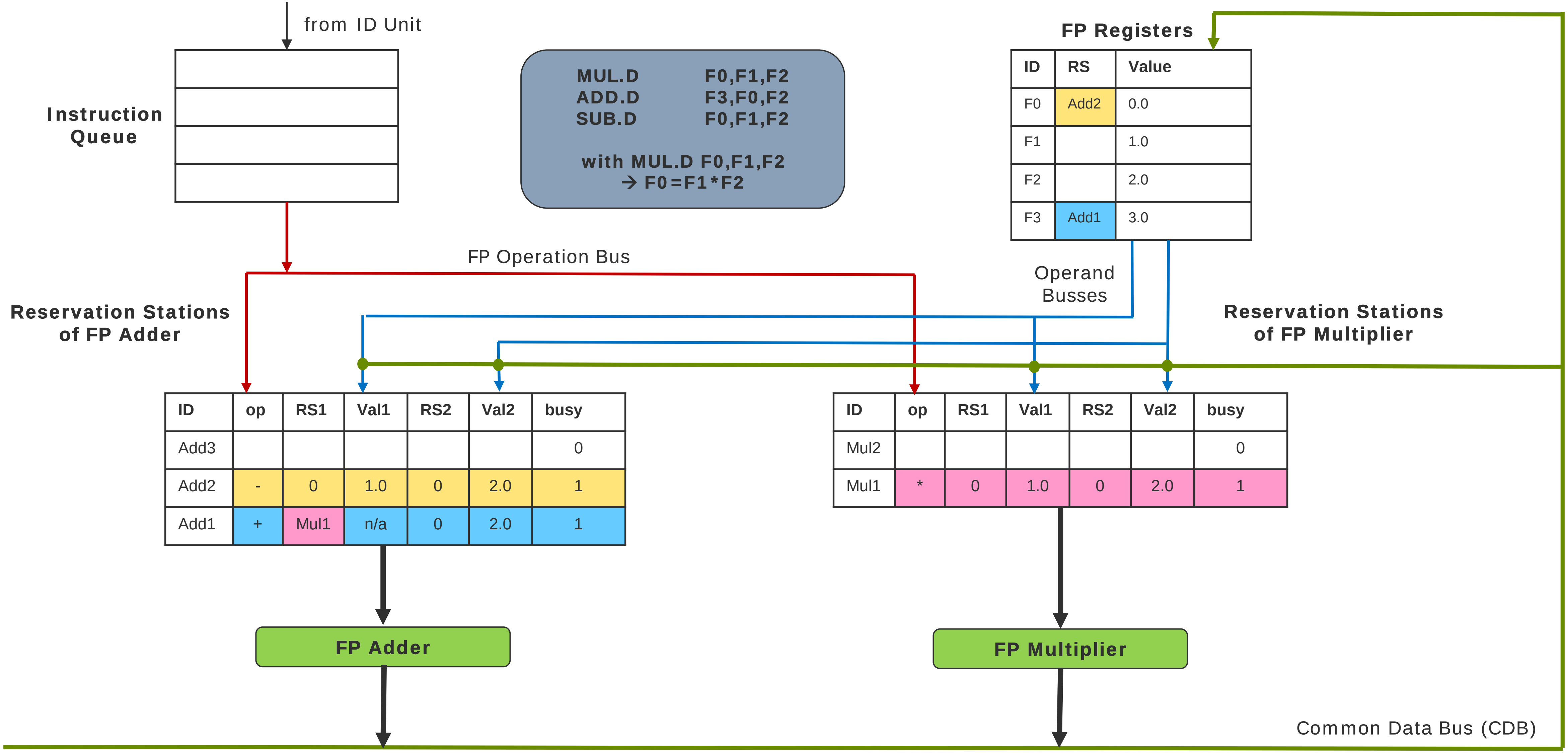
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1	+	Mul1	n/a	0	2.0	1

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1	*	0	1.0	0	2.0	1

Example: Dynamic Scheduling for a FP-Unit using Tomasulo's Algorithm



from ID Unit

Instruction Queue

```
MUL.D     F0,F1,F2
ADD.D     F3,F0,F2
SUB.D     F0,F1,F2
```

with MUL.D F0,F1,F2
→ F0=F1*F2

FP Registers

ID	RS	Value
F0	Add2	0.0
F1		1.0
F2		2.0
F3	Add1	3.0

FP Operation Bus

Operand Busses

Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2	-	0	1.0	0	2.0	1
Add1	+	Mul1	n/a	0	2.0	1

Reservation Stations of FP Multiplier

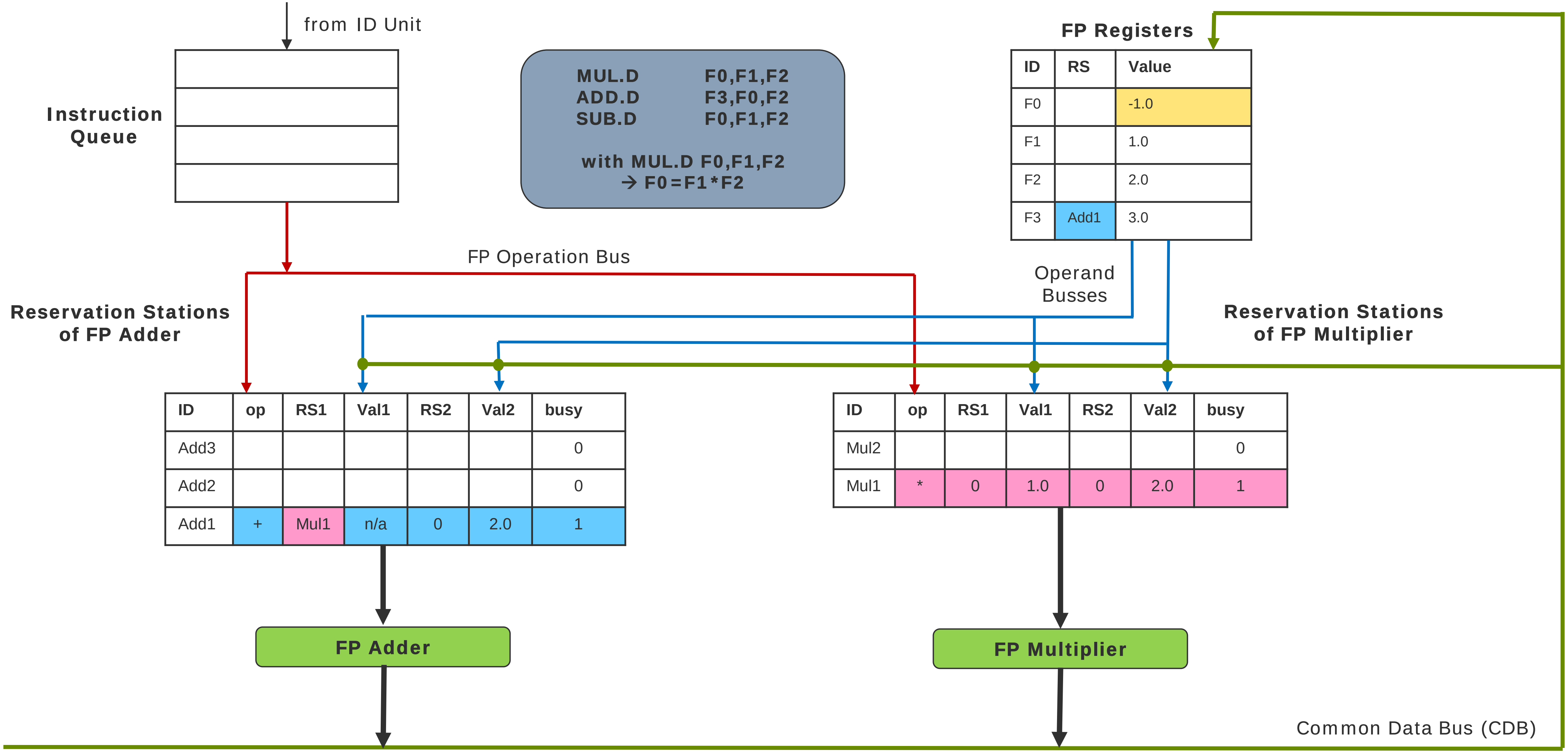
ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1	*	0	1.0	0	2.0	1

FP Adder

FP Multiplier

Common Data Bus (CDB)

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



```
MUL.D    F0,F1,F2
ADD.D    F3,F0,F2
SUB.D    F0,F1,F2
```

with MUL.D F0,F1,F2
→ F0=F1*F2

FP Registers

ID	RS	Value
F0		-1.0
F1		1.0
F2		2.0
F3	Add1	3.0

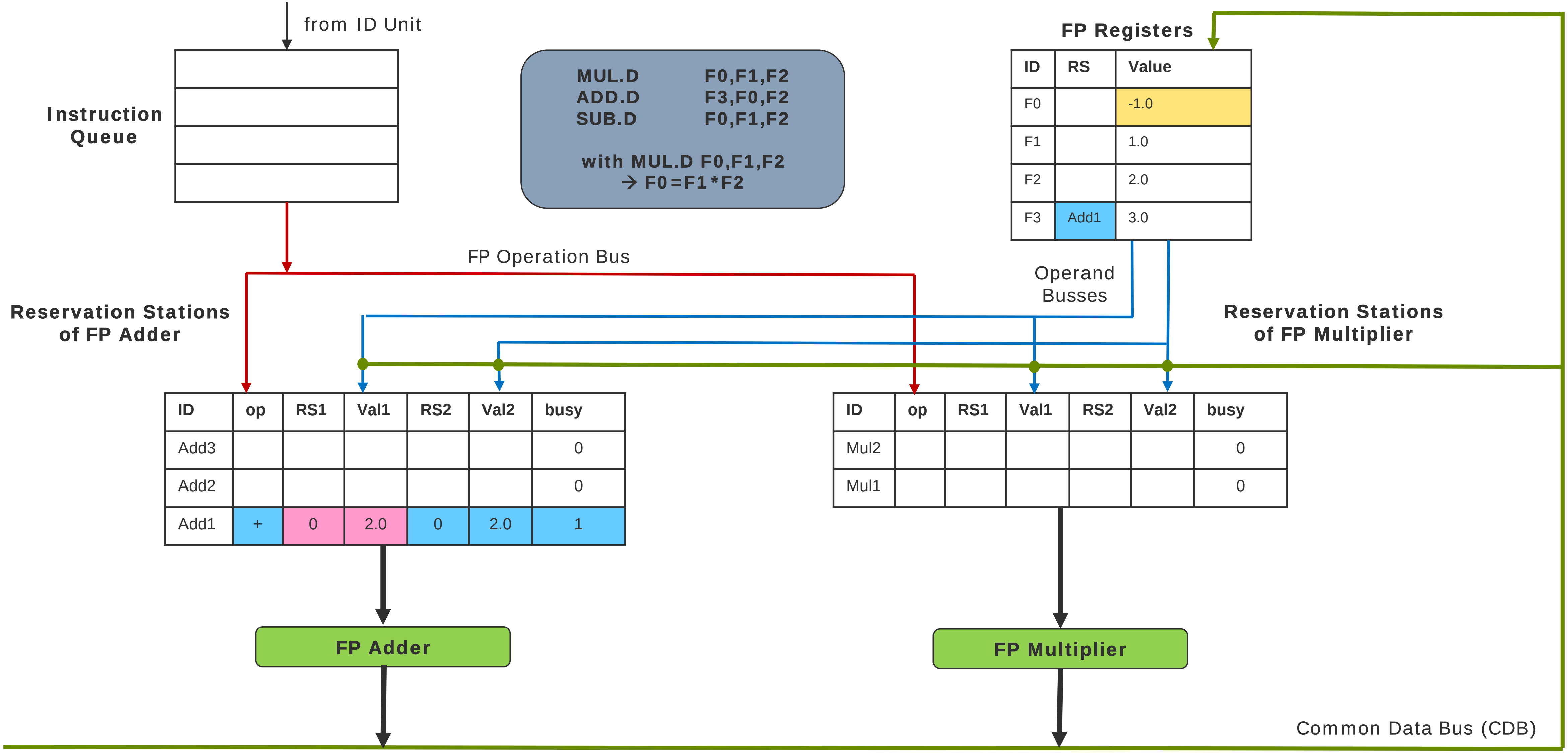
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1	+	Mul1	n/a	0	2.0	1

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1	*	0	1.0	0	2.0	1

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



```
MUL.D     F0,F1,F2
ADD.D     F3,F0,F2
SUB.D     F0,F1,F2
```

with MUL.D F0,F1,F2
→ F0=F1*F2

FP Registers

ID	RS	Value
F0		-1.0
F1		1.0
F2		2.0
F3	Add1	3.0

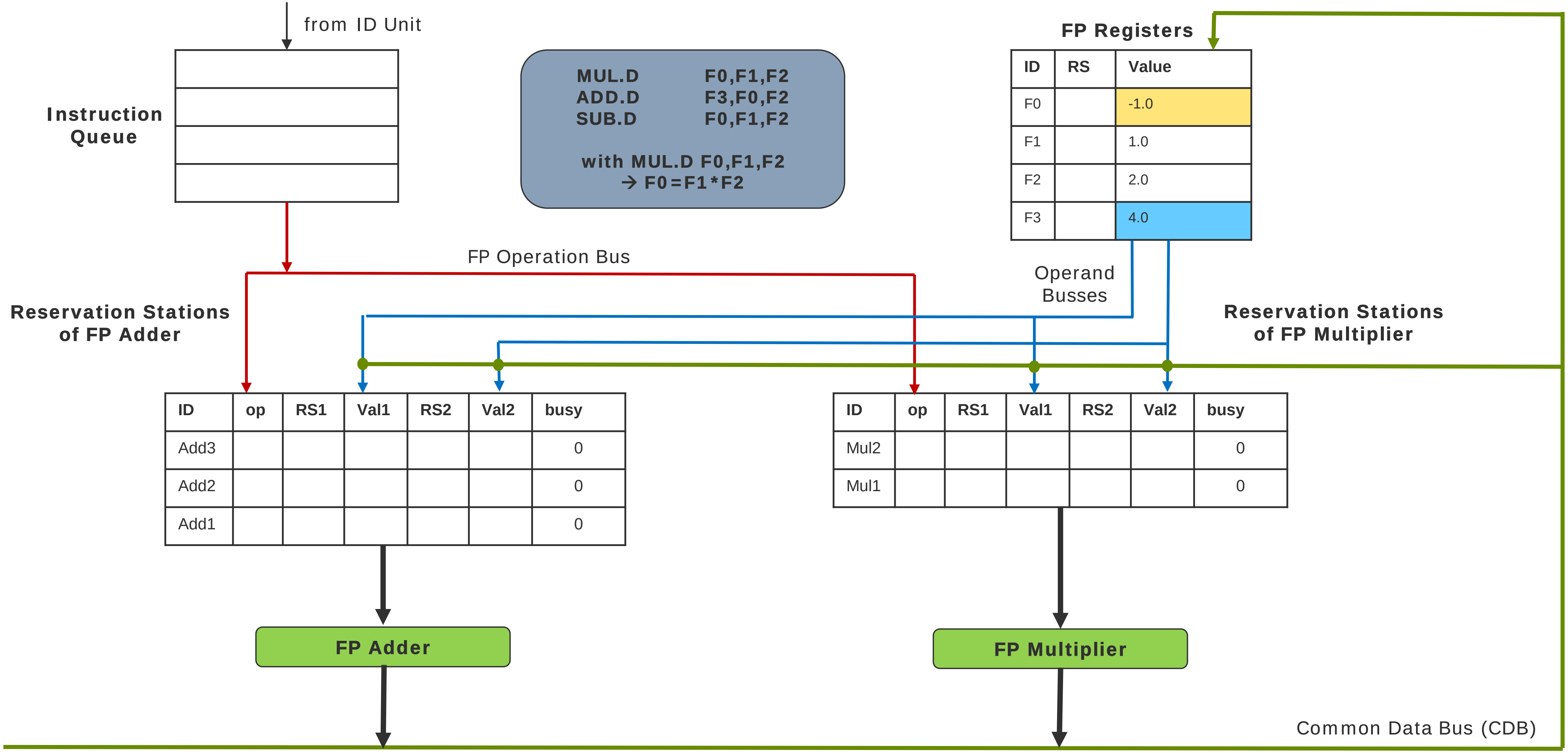
Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1	+	0	2.0	0	2.0	1

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm



```
MUL.D    F0,F1,F2
ADD.D    F3,F0,F2
SUB.D    F0,F1,F2
```

with MUL.D F0,F1,F2
→ F0=F1*F2

FP Registers

ID	RS	Value
F0		-1.0
F1		1.0
F2		2.0
F3		4.0

Reservation Stations of FP Adder

ID	op	RS1	Val1	RS2	Val2	busy
Add3						0
Add2						0
Add1						0

Reservation Stations of FP Multiplier

ID	op	RS1	Val1	RS2	Val2	busy
Mul2						0
Mul1						0

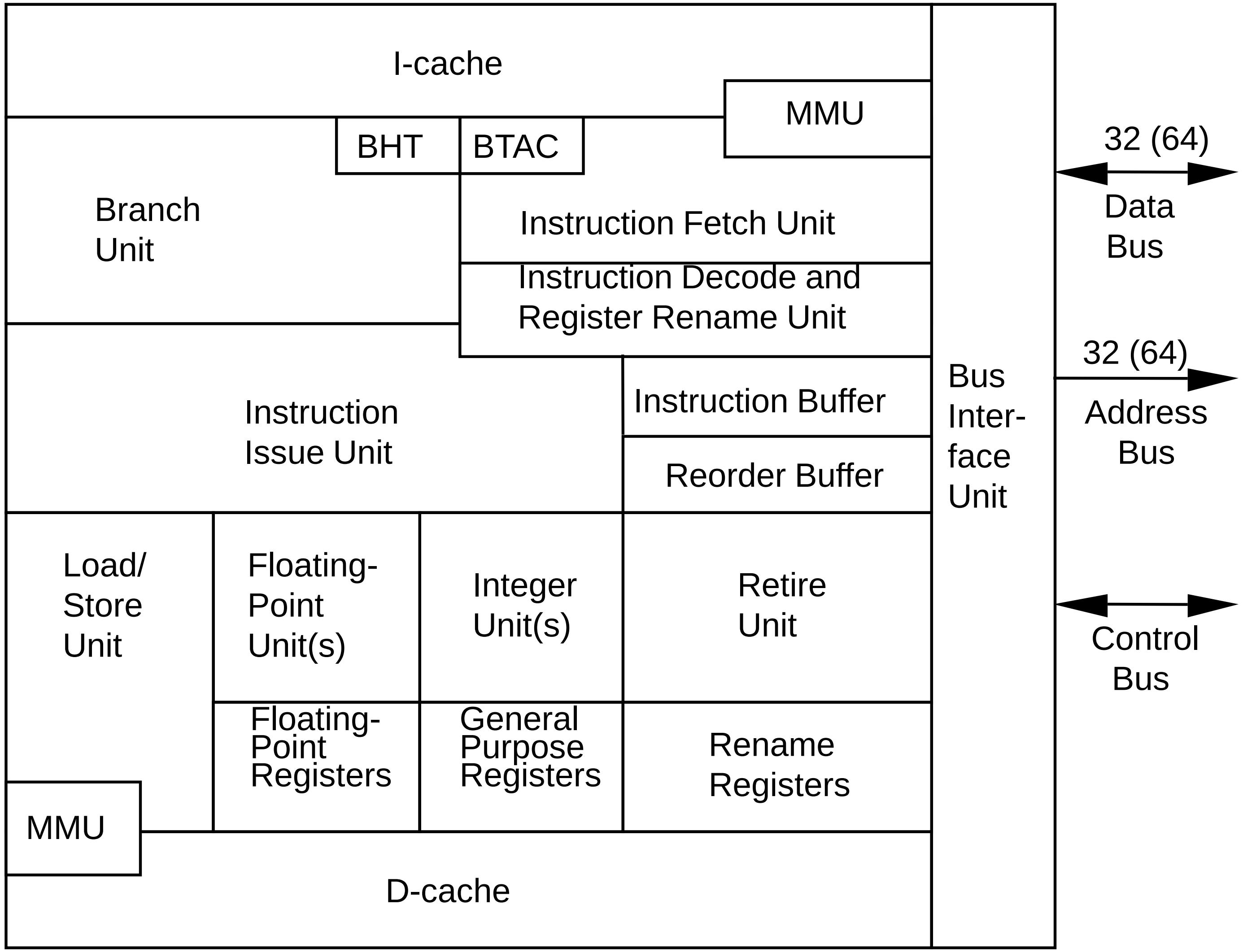
Example: Dynamic Scheduling for a FP-Unit using Tomasulo’s Algorithm

Possible Extensions:

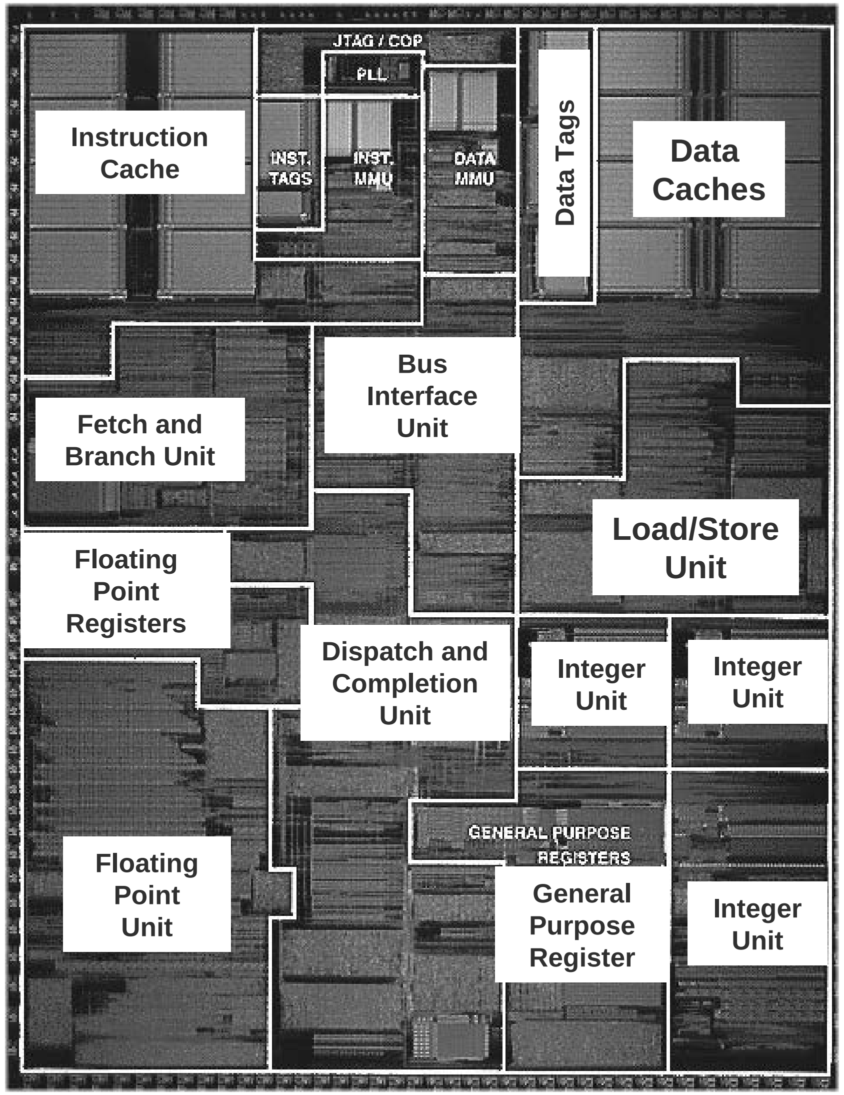
- out-of-order Issuing
- Multiple-Issue
- Speculative Execution using the Reorder Buffer

→ Out of scope for this Bachelor’s module

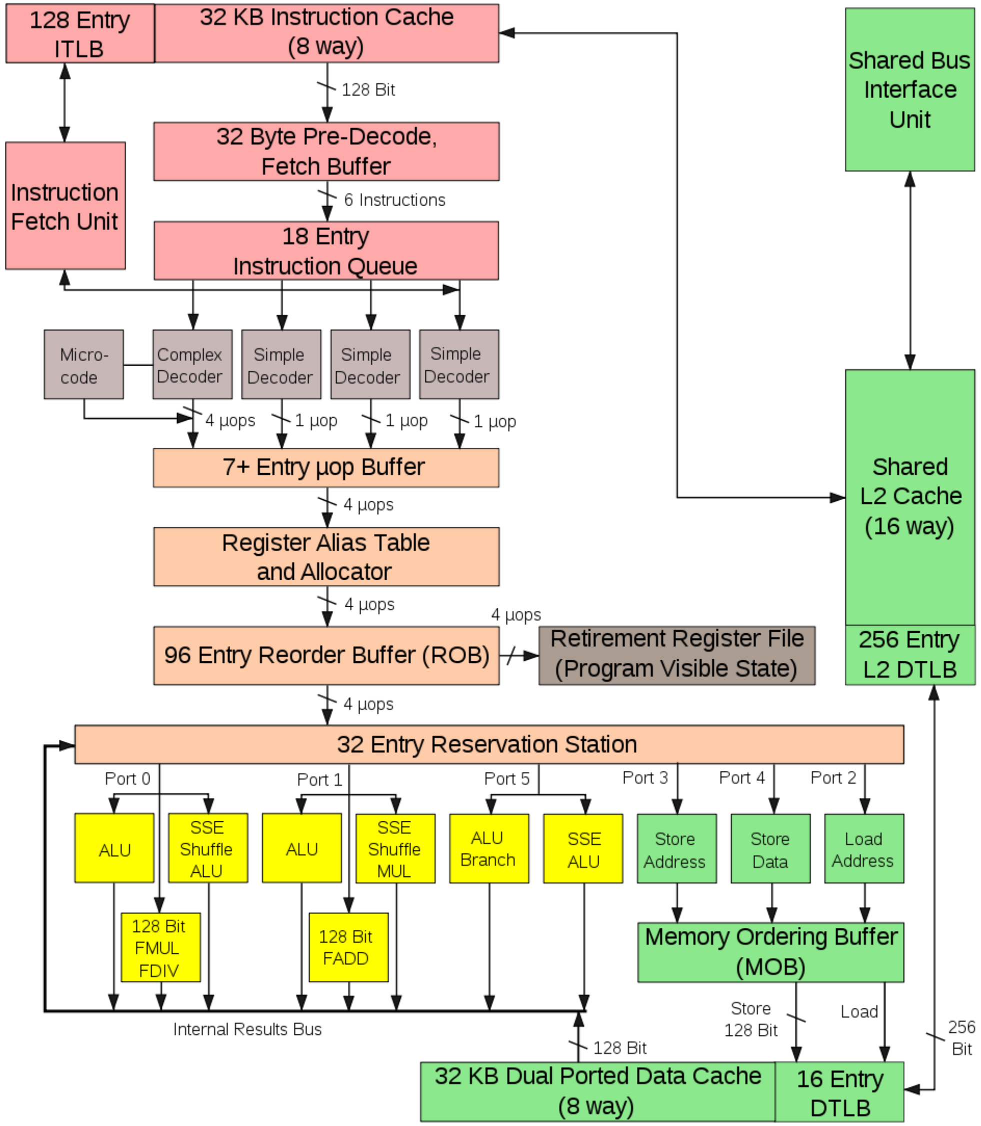
Example Components of a Superscalar Processor



Floor plan of the PowerPC 604



Example: Intel Core 2 Architecture



(Future) Processor Architecture Principles

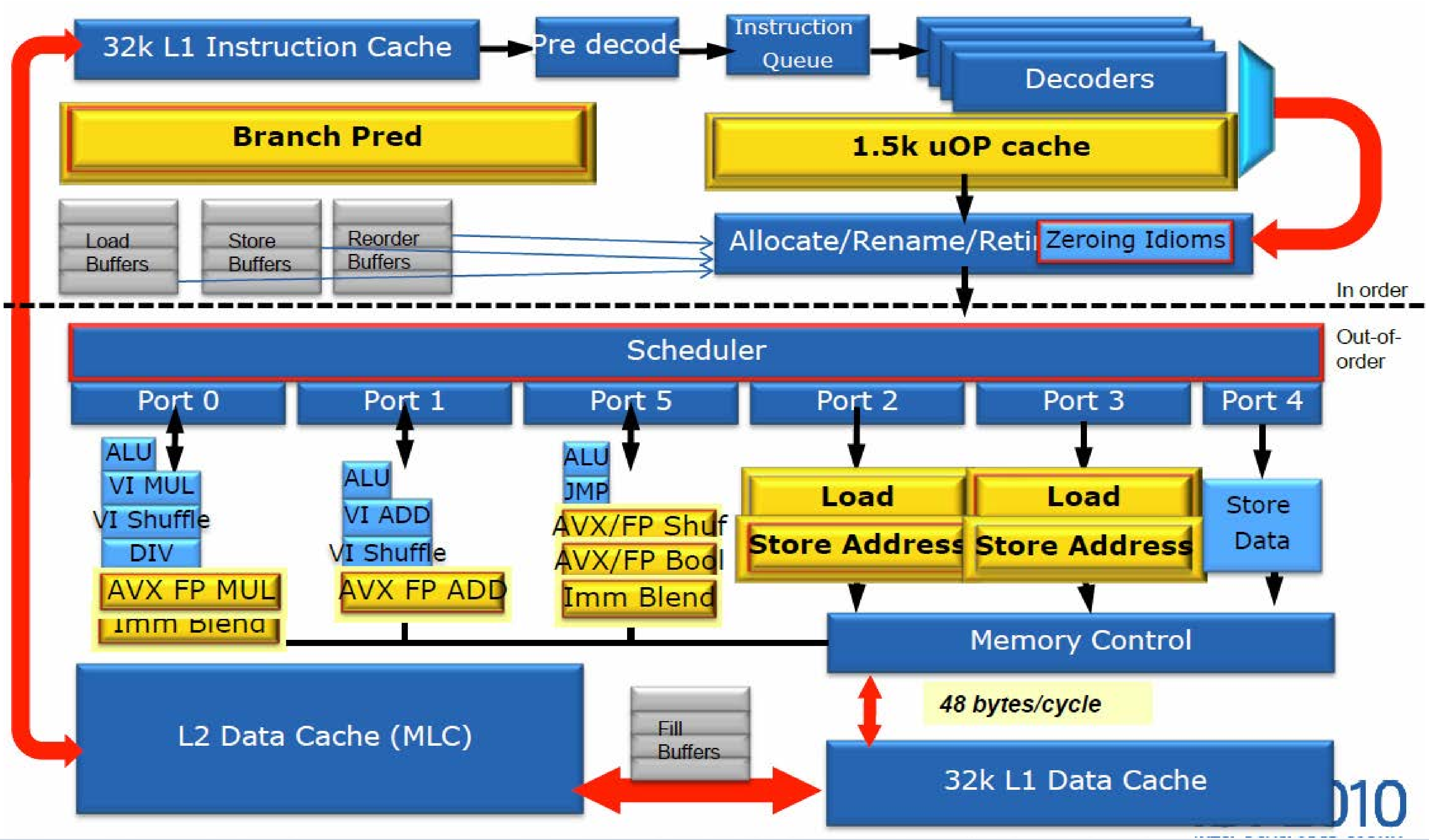
Speed-up of a single-threaded application

- Simultaneous Multithreading
- Advanced superscalar
- Superspeculative
- Multiscalar processors

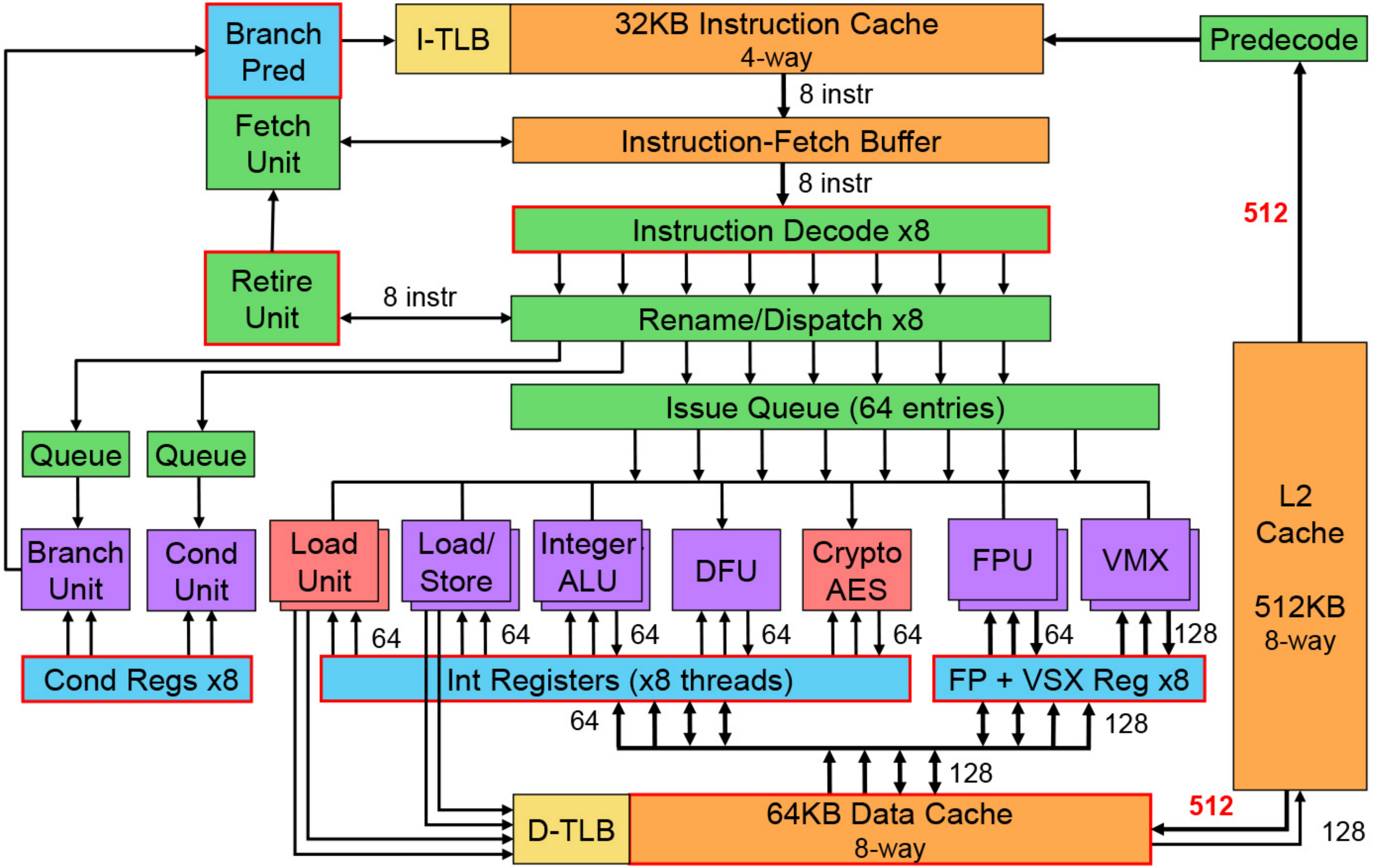
Speed-up of multi-threaded applications

- Chip multiprocessors (CMPs)
- Simultaneous multithreading

Example: Intel Sandy Bridge



Example: Power8



Questions & Tasks

- Check the architecture of your favorite CPU and try to find the different stages of the pipeline!
- What are the key features of Tomasulo’s algorithm?
- What is the role of the Common Data Bus?

Summary

Architecture of a microprocessor

- Control unit, execution unit, registers, interfaces, busses

Performance enhancement of computers

- Technology
- Structures

Pipelining

- Methods
- Hazards
- Branch prediction
- Vector pipelining

Superscalar processors

- Tomasulo