Real-Time and Embedded Systems @ SIT

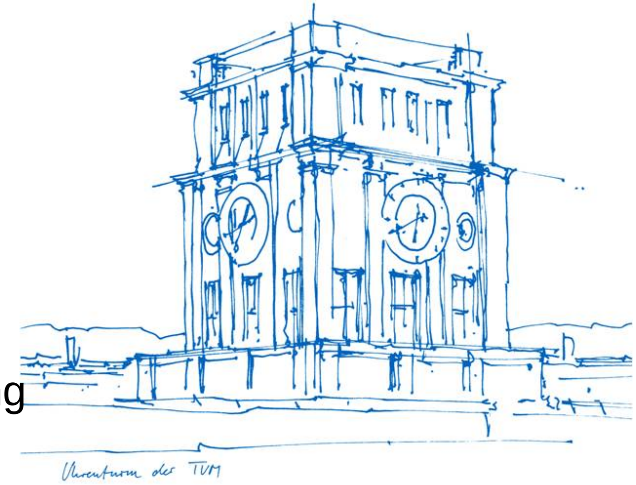
Embedded Processors, Caches and On-Chip Buses

Daniel Mueller-Gritschneder

Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Real-Time Computer Systems



Outlook

- How does the assembly code look like?
- How does assembly code look like for the RISC-V processor?
- How is this code executed on an embedded RISC-V processor?
  - Single-cycle processor
  - Pipelined processor
- How can we estimate the execution time?

- How do on-chip buses work?
- How does the HW/SW interface look like?

RISC-V Instruction Set Architecture

What is an Instruction Set Architecture (ISA)?

- The ISA describes:
  - the processor state organization (registers)
  - what instructions a processor executes.
    - How the instructions are encoded in machine code.
    - How the assembly of the instructions look likes.
  - Some more processor behavior (exceptions, …)
- A cross-compiler can generate the assembly code for a certain processor using the ISA (ISA is the interface between compiler and hardware)

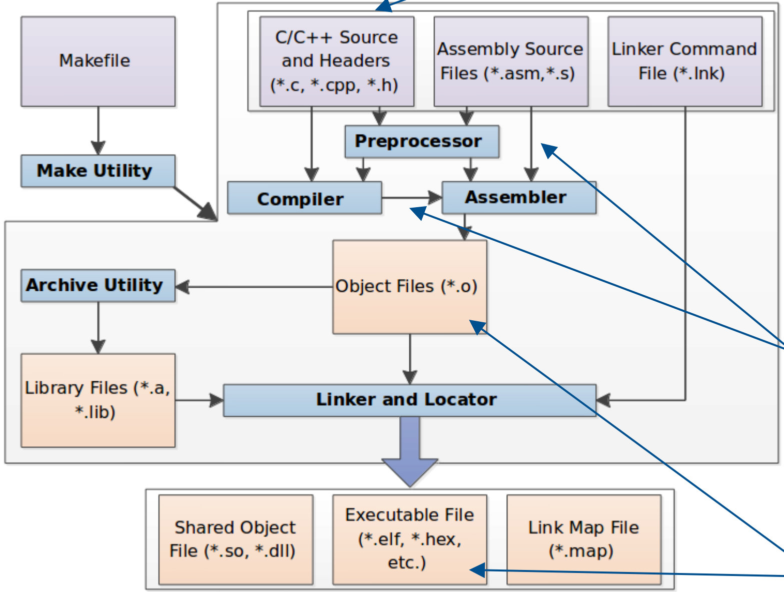
- Instructions are the „*words*" of the processor and the ISA is its „*vocabulary*"

Assembly Instruction Built-up

- Example instruction: `ADD x1,x2,x3`
  - ADD: Addition
  - x1: Destination register
  - x2,x3: Operand Registers
  - Behavior: Regs[x1] ← Regs[x2] + Regs[x3]

- We need to define:
  - The processor has a state Regs consisting of registers
  - Each register has an ID, e.g., here x1,x2,x3
  - Regs[ID] references the value stored in register ID
  - The assembly format of the ADD instruction, e.g., the first register ID is the destination, the second and third ID are the operands

Compilation


Makefile
Make Utility
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

C-code: a=a+4

Assembly-code: `ADDI x10,x10,4`

Machine code: `0x 00 45 05 13`

Processor Terminology

- **Instruction set architecture (ISA):** Set of instructions that can be executed by the processor
- **Microarchitecture:** A processor model that describes the structure of the processor with the target ISA, e.g. nr. of pipeline stages
- **Implementation:** The processor implemented on a chip, e.g. in 90nm technology
- **ARM terminology**:
  - Architectural Revision := ISA: ARMv7M
  - Implementation := Microarchitecture, e.g. ARM Cortex-M3

Why RISC-V?

- Many embedded processors use the ARM ISA.
- ARM is an IP vendor and does not sell chips.
- Semiconductor companies such as NXP, TI, Qualcomm, Infineon buy ARM processor IPs and integrate them into their Micro-Controllers or System-on-chip (SoCs).

- RISC-V is an open ISA, which can be easily used in academia
- Invented by UC Berkeley.
- RISC-V is a very hot topic in industry and more and more RISC-V SoCs are becoming available.

Registers of RISC-V

- RISC-V has 32 registers
- Each register can have different width, we look at RV32 with 32-bit width
- Each register has two IDs (x0-x31) and an ABI name that indicates its role

Register	ABI Name	Description	Saver
x0	Zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0,fp	Saved register/frame pointer	Callee
x9	s1	Saved Register	Callee
x10-11	a0-1	Function arguments	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Application Binary Interface (ABI)

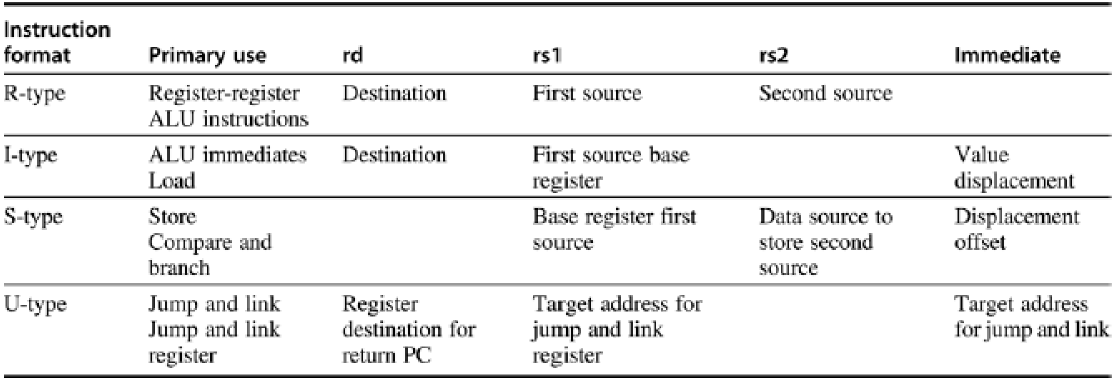
- Specifies rules for register usage in passing arguments and results for function calls
  - Callee-saved registers: If function foo1 (caller) calls foo2 (callee), then foo2 is not allowed to modify this value (it needs to save it and restore it before returning to foo1
  - Caller-saved registers: If function foo1 (caller) calls foo2 (callee), then foo1 needs to save this register before calling foo2 if it wants to keep the value in it because foo1 is allowed to modify it

- Assigns aliases for registers x1-x31 (see table previous slide)

RISC-V Instruction Types

- Four types of instructions with different encoding in machine code

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type



Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

RISC-V Instructions

- The RISC-V ISA is structured into several instructions groups
  - We look at RV32IM
- 32-bit Integer Instruction RV32I
  - Integer Register-Register Instructions (R-type)
    - Runs an arithmetic or logical operation on registers
    - Both operands are values in registers
  - Integer Register-Intermediate Instructions (I-type)
    - Second operand is a immediate (constant) value
    - Immediate is encoded strangely in the machine code
  - Control Transfer Instructions
    - Unconditional jumps
    - Conditional Branches
  - Load Store Instructions
    - Move data between memory and registers
    - Load-store Architecture: Operations on registers only
- 32-bit Integer Multiplication RV32M
  - Integer Multiplication Instructions
  - Integer Division Instructions

Integer Register-Register Instructions 1

- Instruction: `Instr xd,xm,xn`
- Instr: Assembly instruction name
- xd: Destination register
- xm,xn: Operand Registers
- Behavior: Regs[xd] ← Regs[xm] OP Regs[xn]

- Addition: `ADD a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] + Regs[a3]
- Subtraction: `SUB a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] - Regs[a3]

- Signed compare: `SLT a1,a2,a3`
  - Behavior: **if** (Regs[a2] < Regs[a3]) Regs[a1] ← 1 **else** Regs[a1] ← 0
- Unsigned compare: `SLTU a1,a2,a3`
  - Behavior: **if** (Regs[a2] < Regs[a3]) Regs[a1] ← 1 **else** Regs[a1] ← 0

Integer Register-Register Instructions 2

- Logical AND: `AND a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] & Regs[a3]
- Logical OR: `OR a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] | Regs[a3]
- Logical XOR: `XOR a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] ^ Regs[a3]

- Shift Left Logical: `SLL a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] << Regs[a3] (we shift 0s in from the right)
- Shift Right Logical: `SRL a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] >> Regs[a3] (we shift 0s in from the left)
- Shift Right Arithmetic: `SRA a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] >> Regs[a3] (if MSB is 1, we shift 1s in from the left, else 0s)

Integer Register-Immediate Instructions

- Format:

- Addition with immediate (constant value): Example: `ADDI a1,a2,3`
  - Behavior: Regs[a1] ← Regs[a2] + 3

- Further instructions as before but with immediate: `SLTI, SLTIU, SLLI, SRLI, SRAI`

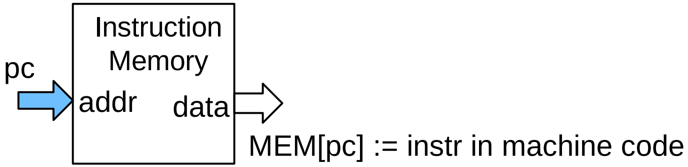
- There is no `SUBI:` Use addition with negative immediate: `ADDI a1,a2,-3`

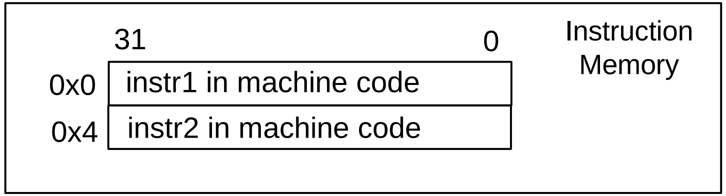
- No operation: `NOP`
  - Behavior: Does nothing

- Move: Example: `MV a1,a2`
  - Behavior: Regs[a1] ← Regs[a2]
  - Is a so-called pseudo instruction: The processor translates it to `ADDI a1,a2,0`

Program Counter and Instruction Memory

- The control transfer instructions change the program counter (pc)
- The pc tells from which address the next instruction should be fetched
- In RV32, each instruction is 32 bit or 4 byte
- The address is in bytes, so to jump from the one instruction to the next, the pc has to increment by 4.
- MEM[pc] is the instruction in the instruction memory stored at the address pc





- When there are Control Transfer Instructions, the PC is modified such that it jumps to another location different from the next instruction (PC+4) to implement
  - if, else, switch blocks
  - loops
  - function calls
  - function returns

Control Transfer Instructions - Jumps

- Unconditional Jump: `J offset`
  - Behavior: pc ← pc + (offset<<1)
  - Example: `J 8` has behavior: pc ← pc + (8<<1) = pc + 16
  - But we usually do not put the offset, but a symbol, e.g., start of loop that we want to jump to, e.g., pc = loop_start by writing `J loop_start`

- Unconditional Jump and Link: Example: `JAL ra,8`
  - Behavior: pc ← pc + (8<<1) = pc + 16
    Regs[ra] ← pc + 4
  - Here, we want to jump to a function. In order to be able to return, we save the next instruction address (pc+4) in the return address (ra) register regs[ra] = regs[x1]
  - So again here we do not put the offset but the symbol of the function: `JAL foo1`

- Unconditional Jump Register: Example: `JR ra`
  - Behavior: pc ← Regs[ra]
  - This is usually used for function return. The return addressed is saved in register ra.
  - A pseudo-instruction is `RET // equal to JR ra`

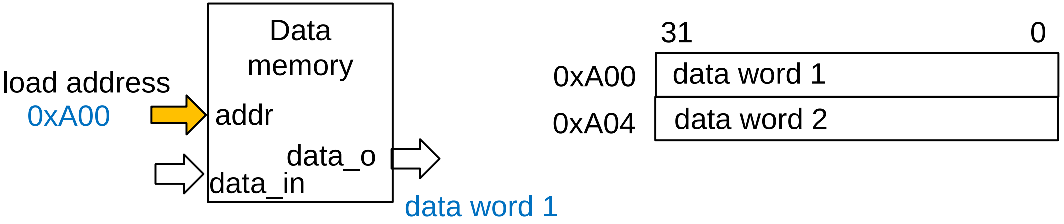
Control Transfer Instructions - Branches

- Conditional branch equal zero: `BEQ a1,a2, loop_start`
  - Behavior: **if** regs[a1] == regs[a2]) pc = loop_start **else** nothing

- Further branch instructions
  - not equal: `BNE a1,a2, loop_start`
  - Lesser than: `BLT a1,a2, loop_start`
  - Unsigned lesser than: `BLTU a1,a2, loop_start`
  - Greater or equal than: `BGE a1,a2, loop_start`
  - Unsigned greater of equal than: `BGEU a1,a2, loop_start`

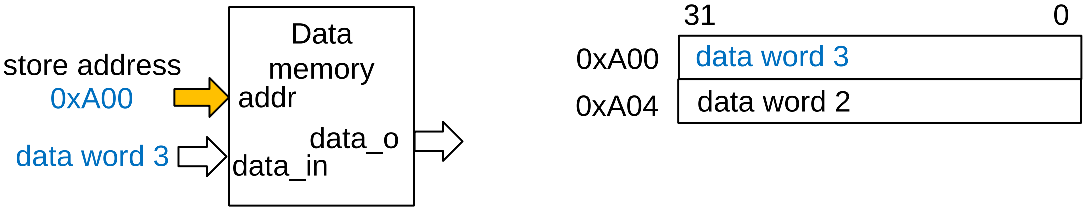
Data Memory

- Load and store instructions access the data memory (data, stack or heap)
- MEM[addr] is the value in the data memory stored at the address addr
- The load instruction fetches a value on data memory



	31	0
0xA00	data word 1	
0xA04	data word 2	

- The store instructions saves a value in data memory



	31	0
0xA00	data word 3	
0xA04	data word 2	

Load and Store Instructions

- Load word: `LW a1,80(a2)`
  - Behavior: Regs[a1] ← MEM[80 + Regs[a2]]
  - We set `(a2)`to indicate that the value in a2 is used as an address, 80 is the offset
  - Loads a word (4 byte)

- Store word: `SW a1,80(a2)`
  - Behavior: MEM[80 + Regs[a2]] ← Regs[a1]
  - We set `(a2)`to indicate that the value in a2 is used as an address, 80 is the offset
  - Stores a word (4 byte)

- Other instructions used to store half words (2 byte) or bytes

Integer Multiplication Instructions

- Signed-signed Multiplication
  - Multiplying two 32bit values can result in a value of up to 64bit
  - `MUL rdl,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1]*regs[rs2] // only the lower 32bit
  - `MULH rdh,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1]*regs[rs2] // only the higher 32bit
  - Example:

    -
      ```
       MUL a3,a1,a2
       MULH a4,a1,a2
      ```
    - Behavior: [regs[a4] regs[a3]] = regs[ra1]*regs[a2] // full 64 bit

- Unsigned-unsigned multiplication `MULU, MULHU`
- Unsigned-signed multiplication `MULSU, MULHSU`

Integer Division Instructions

- Signed-signed Division
  - `DIV rdl,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1] / regs[rs2]
  - `REM rdh,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1] modulo regs[rs2] // remainder
  - Example: `DIV a3,a1,a2`
    Behavior: regs[a3] = regs[ra1] / regs[a2]

- Unsigned-unsigned division `DIVU, REMU`
- Unsigned-signed division `DIVSU, REMSU`

Writing RISC-V Assembly Code

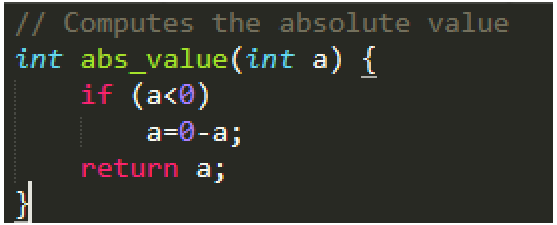
Writing a function in assembly

- Start with a symbol with the function name `foo1:`
- The first function parameter is in a0, second in a1, …
- The return value should be in a0 before returning
- For returning use the RET instruction

- Guideline: Use tx registers for temporary local variables

Writing a small assembly function 1

- Example C-Code 1



```c
// Computes the absolute value
int abs_value(int a) {
    if (a<0)
        a=0-a;
    return a;
}
```

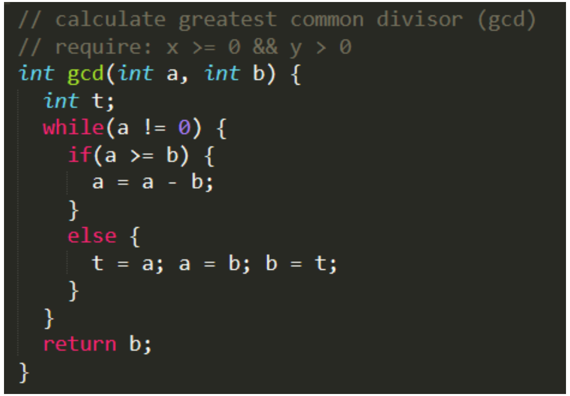
- RISC-V Code
  - According to ABI a is given to the function in register a0
  - The function should also return a in register a0

```
abs_value:
     BGE a0,zero,abs_value_return // if a>=0
     SUB a0,zero,a0   // a=0-a
abs_value_return:
     RET                // JR ra
```

function
return

Writing a small assembly function 2

- Example C-Code 2



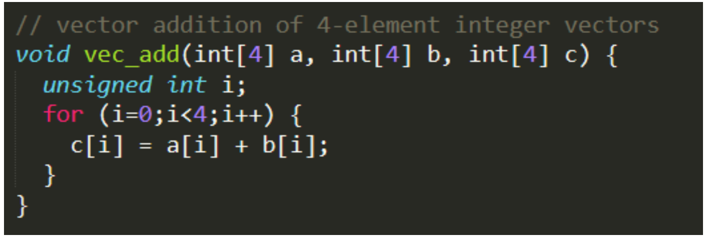
```c
// calculate greatest common divisor (gcd)
// require: x >= 0 && y > 0
int gcd(int a, int b) {
  int t;
  while(a != 0) {
    if(a >= b) {
      a = a - b;
    }
    else {
      t = a; a = b; b = t;
    }
  }
  return b;
}
```

RISC-V Code

```
// a: a0, b: a1, t: t0
gcd:
  BEQZ a0, gcd_done       // while(a!=0)
  BLT a0, a1, gcd_else    // a < b -> else
  SUB a0, a0, a1          // a = a-b
  J gcd                   // while loop
gcd_else:
  MV t0, a0               // t = a
  MV a0, a1               // a = b
  MV a1, t0               // b = t
  J gcd                   // while loop
gcd_done:                 // now a1 contains the gcd
  MV a0, a1               // move to a0 for returning
  RET                     // return (jr ra)
```

Writing a small assembly function 3

- Example C-Code 3



```c
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
// base address of a: a0,
// base address of b: a1,
// base address of c: a2,
// i: t0,  constant 4: t3
vec_add:
  ADDI t0,zero,0       // i=0
  ADDI t3,zero,4       // t3=4
vec_add_for:
  LW t1,(a0)    // t1 = a[i]
  LW t2,(a1)    // t2 = b[i]
  ADD t1,t1,t2  // t1 = a[i] + b[i]
  SW t1,(a2)    // c[i] = t1
  ADDI a0,a0,4  //next element is base address + 4
  ADDI a1,a1,4  //next element is base address + 4
  ADDI a2,a2,4  //next element is base address + 4
  ADDI t0,t0,1  // i++
  BLTU t0,t3,vec_add_for // for (i < 4)
  RET   // void return
```

Embedded Processor RISC Architecture

Single-Cycle

Processor Classification 1

- Reduced Instruction Set Computer (RISC)
  - Many registers
  - Three-address instructions
  - Simple addressing modes
  - Relative simple ISA
  - Examples: MIPS, PowerPC, ARM, RISC-V
- Complex Instruction Set Computer (CISC)
  - Few registers
  - two-address instructions
  - Variety of addressing modes
  - Several register classes
  - Variable length instructions
  - Example: x86
- Stack-based computers:
  - Operands pushed on stack
  - Performing operations on top of stack

Classification 2

- General-purpose computer:
  - ISA for general use

- Application-specific instruction set processor (ASIP)
  - ISA tailored for class in certain application domain

- Digital signal processor (DSP)
  - ISA optimized to process digital signals, e.g. video, audio.

Classification 3

- Von-Neumann architecture: Program and data on same bus
- Harvard architecture: Program and data on separate buses

Processor Architectures

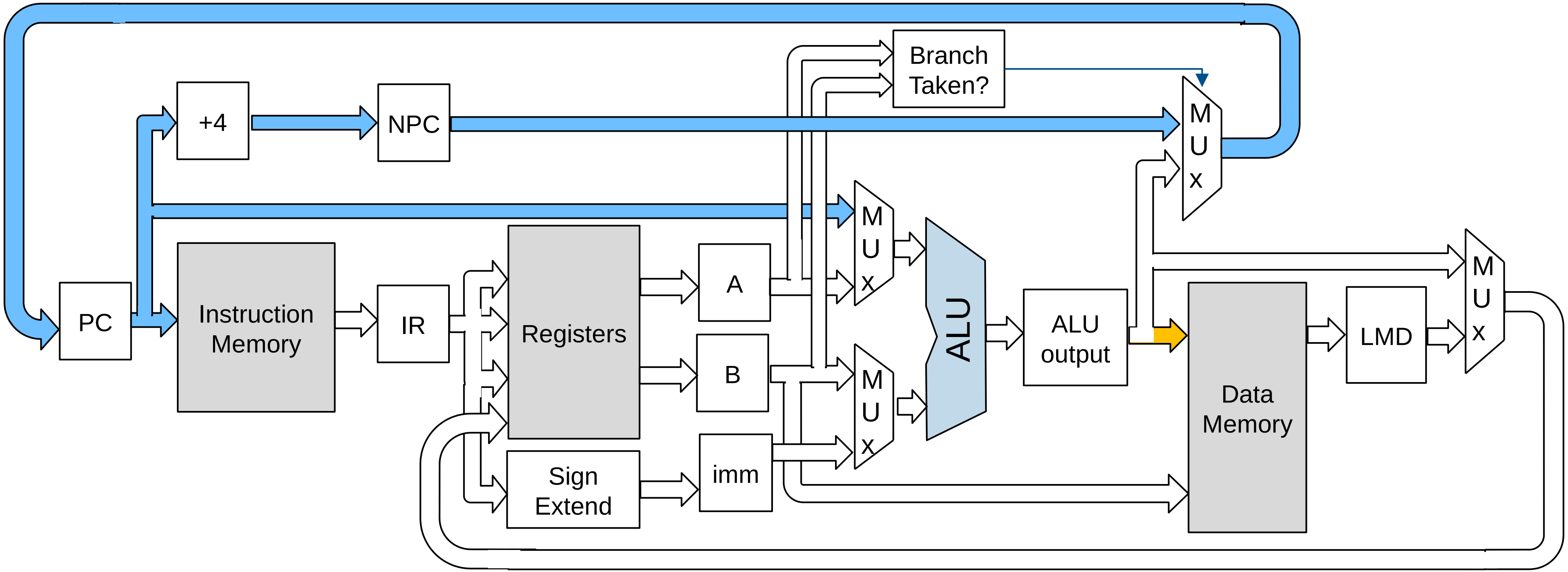
- Single-cycle processors:
  - Execute every instruction in one cycle
  - Usually only low frequencies because maximum clock cycle is determined by longest instruction.

- Multi-cycle processors:
  - Some instruction take several cycles.
  - Allows higher frequency than single-cycle execution

- Pipelined processors:
  - Instructions are split into several stages
  - When one instructions leaves a stage, the next instruction can enter it and does not have to wait for its predecessor to finish.
  - Allows higher clock frequencies than without pipeline

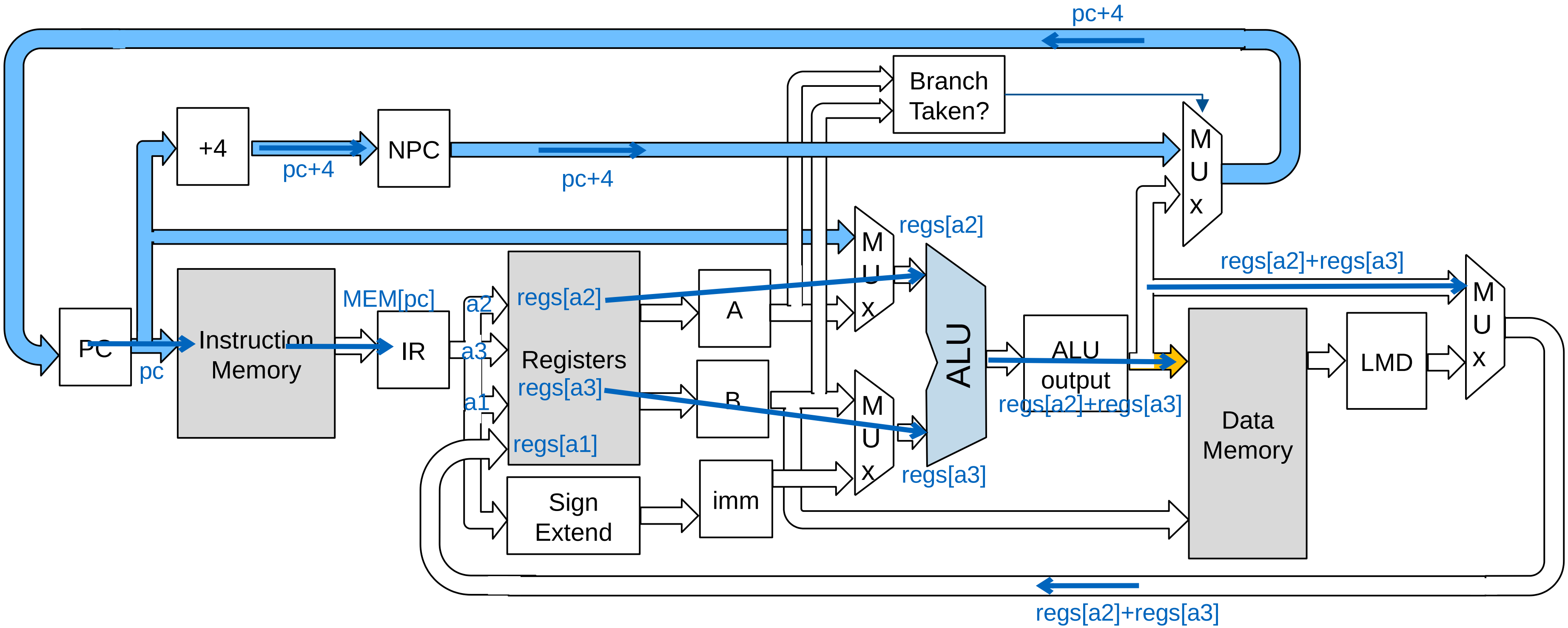
Single-cycle Processor (1/5)

- Simple architecture to support RISC-V RV32I instructions (not RV32M)



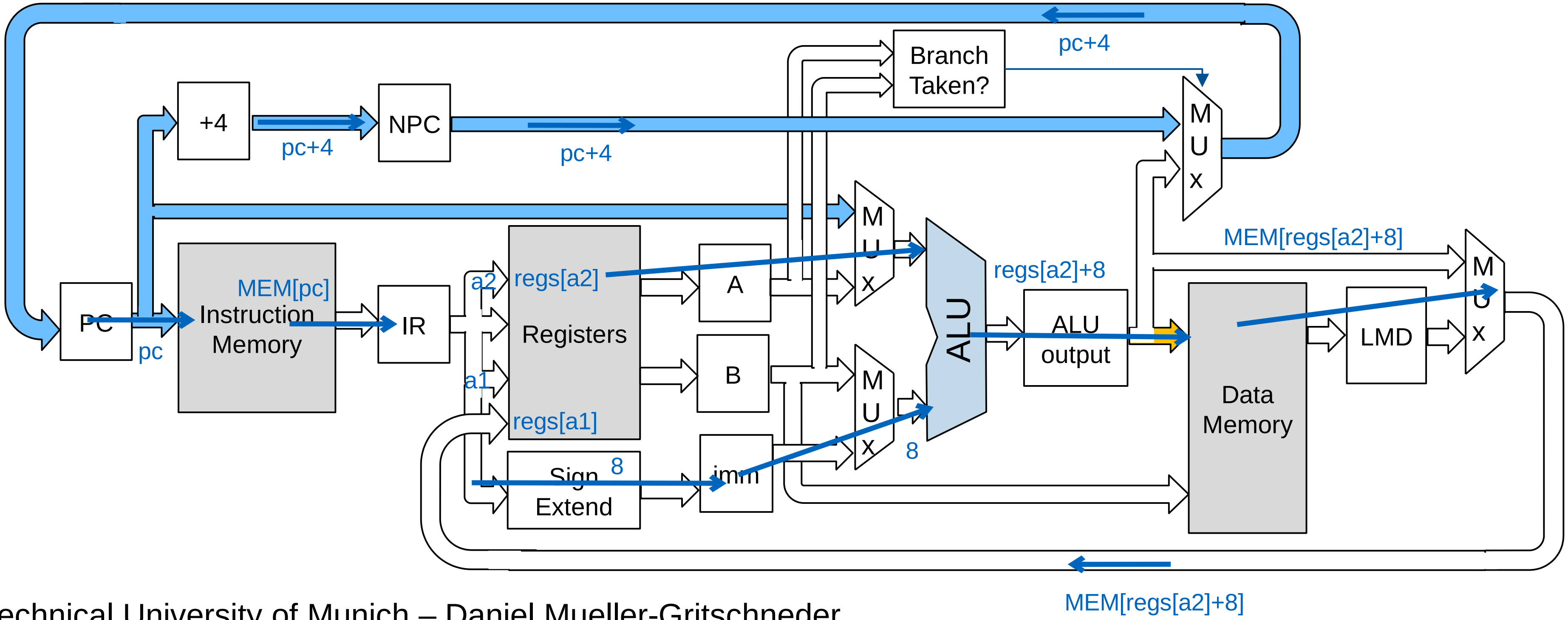
Single-cycle Processor (2/5)

- Example: `ADD a1,a2,a3`



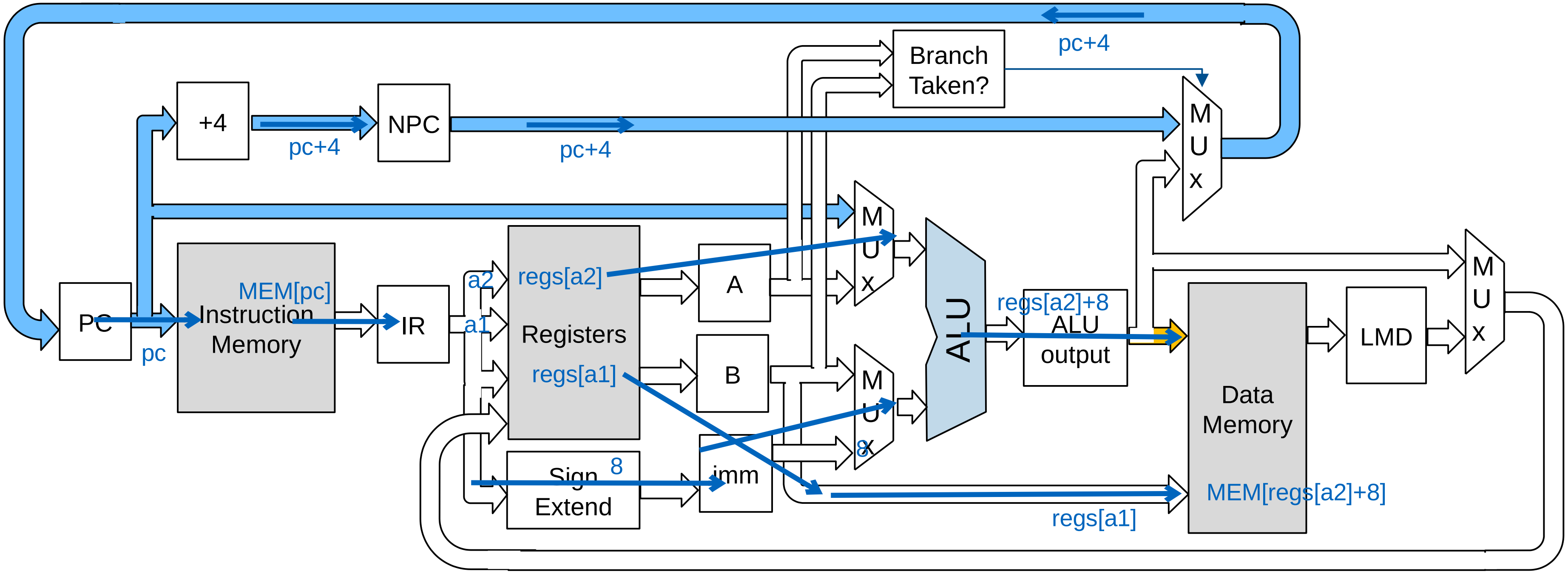
Single-cycle Processor (3/5)

- Example: `LW a1,8(a2)`



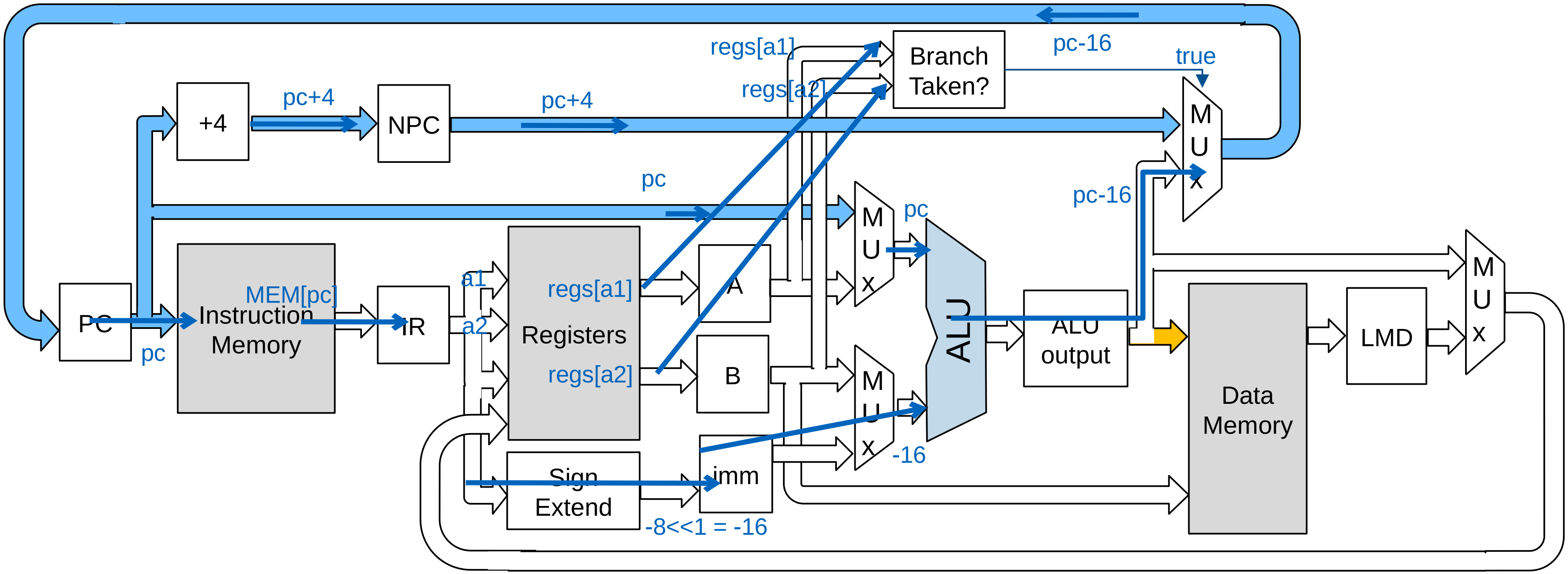
Single-cycle Processor (4/5)

- Example: `SW a1,8(a2)`



Single-cycle Processor (5/5)

- Example (for branch is taken a1==a2): `BEQ a1,a2,-8`



Embedded Processor RISC Architecture

Pipelined

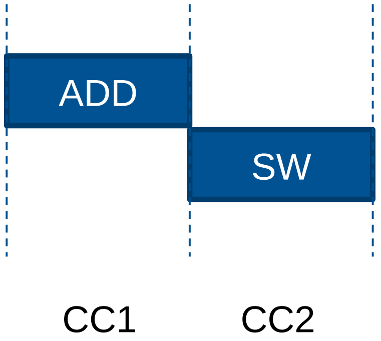
Single-Cycle vs. Pipelined Execution

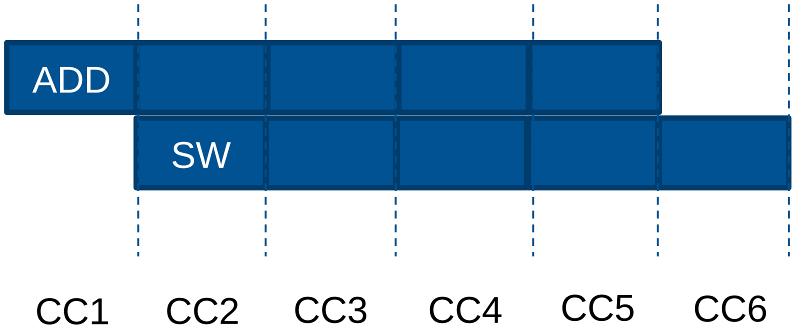
Single-Cycle

Pipelined

- Break Instruction into stages
- When an instruction leaves a stage, the next one can use it.
- Higher clock frequency

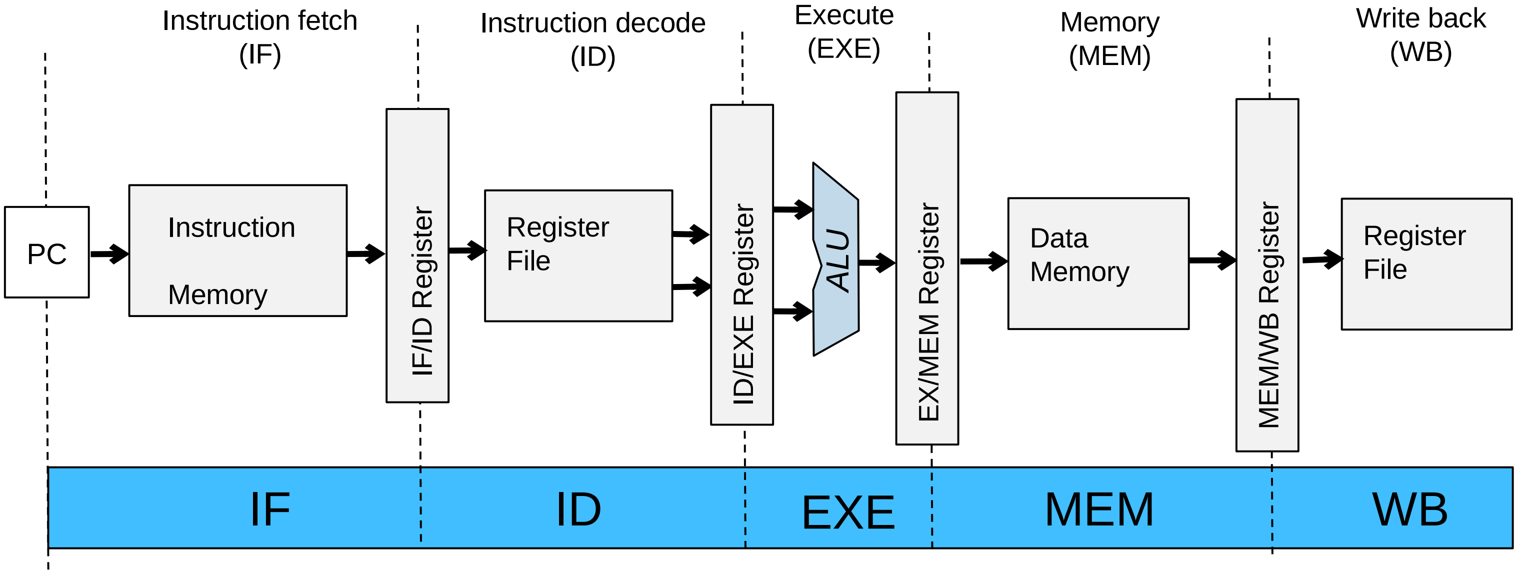
```
ADD a1,a2,a3
SW  a1,8(a2)
```





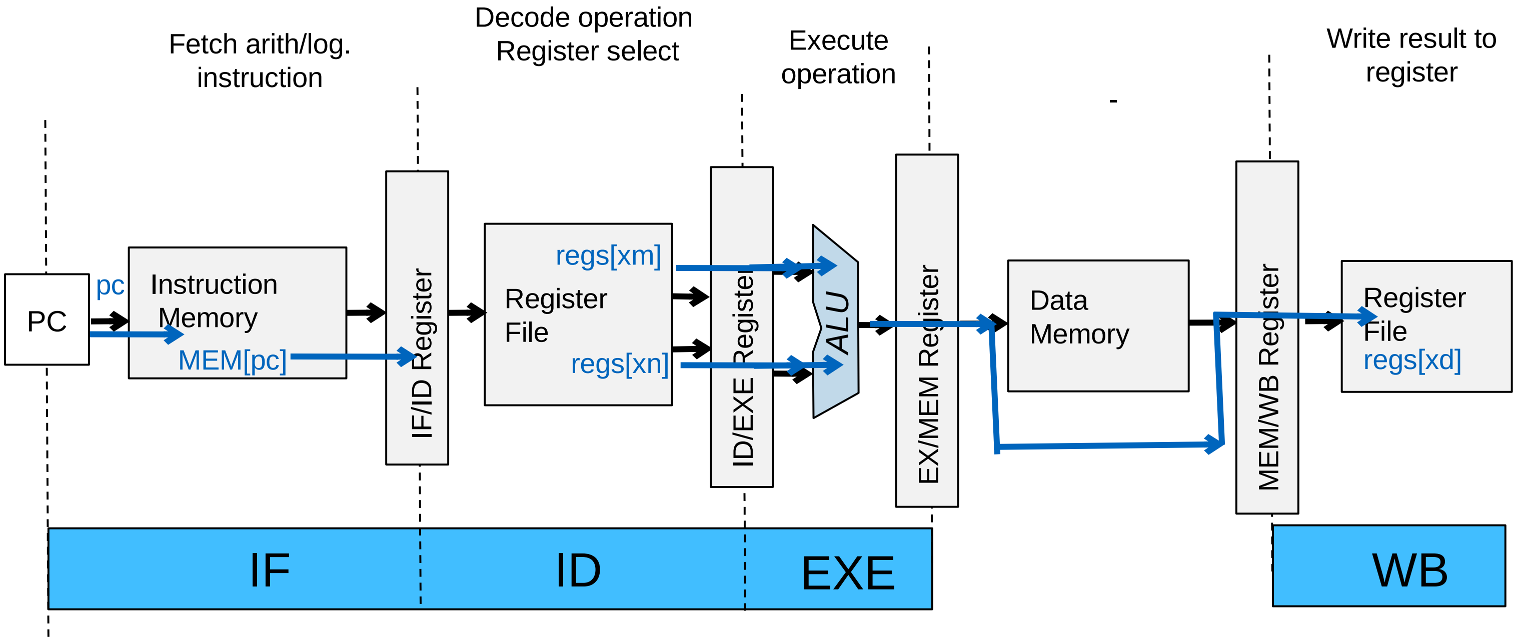
Standard 5-stage RISC Pipeline (1/5)

- Instructions pass through five pipeline stages.
- Registers save values between the stages
- Flow of data between stages:



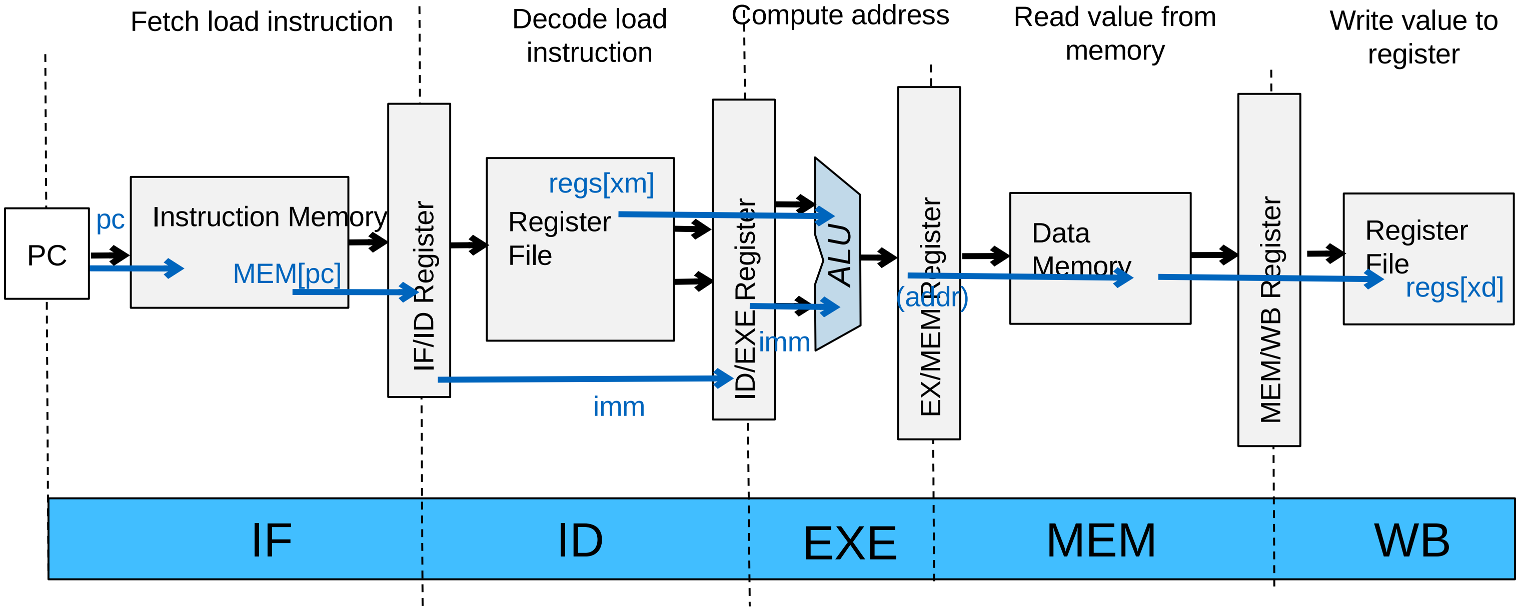
Standard 5-stage RISC Pipeline (2/5)

- Instructions for Arithmetic/Logic Operation



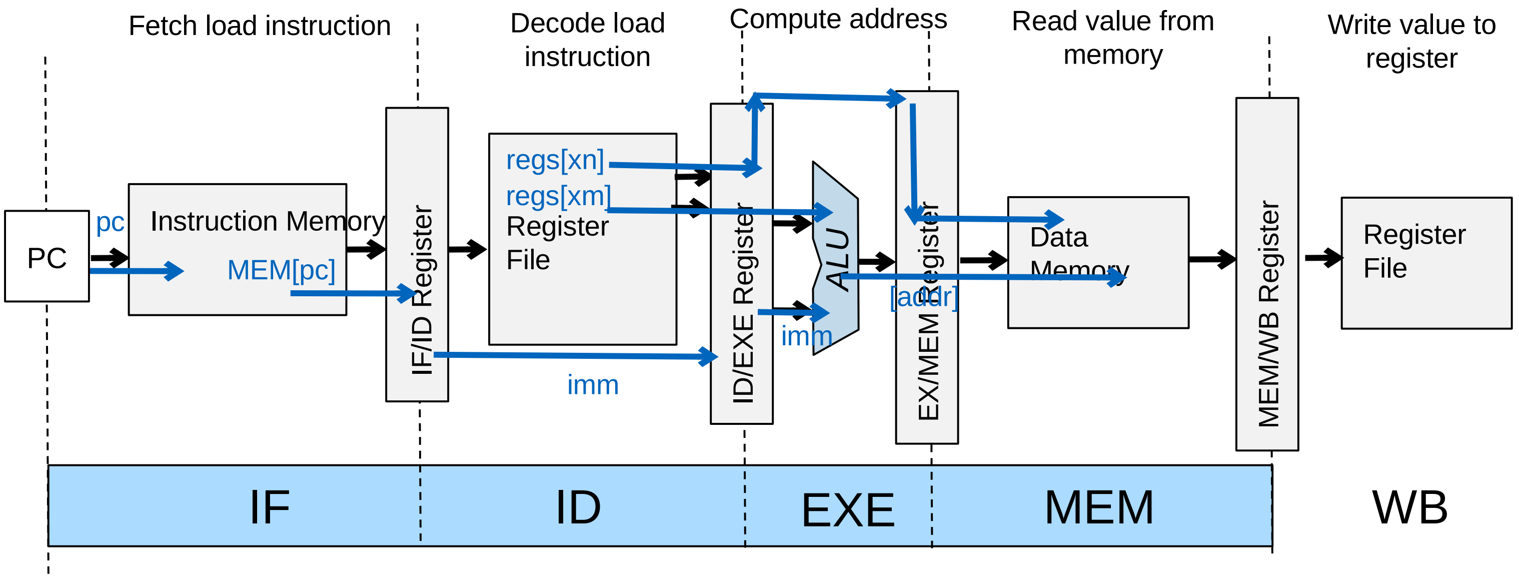
5-stage RISC Pipeline (3/5)

- Load instruction



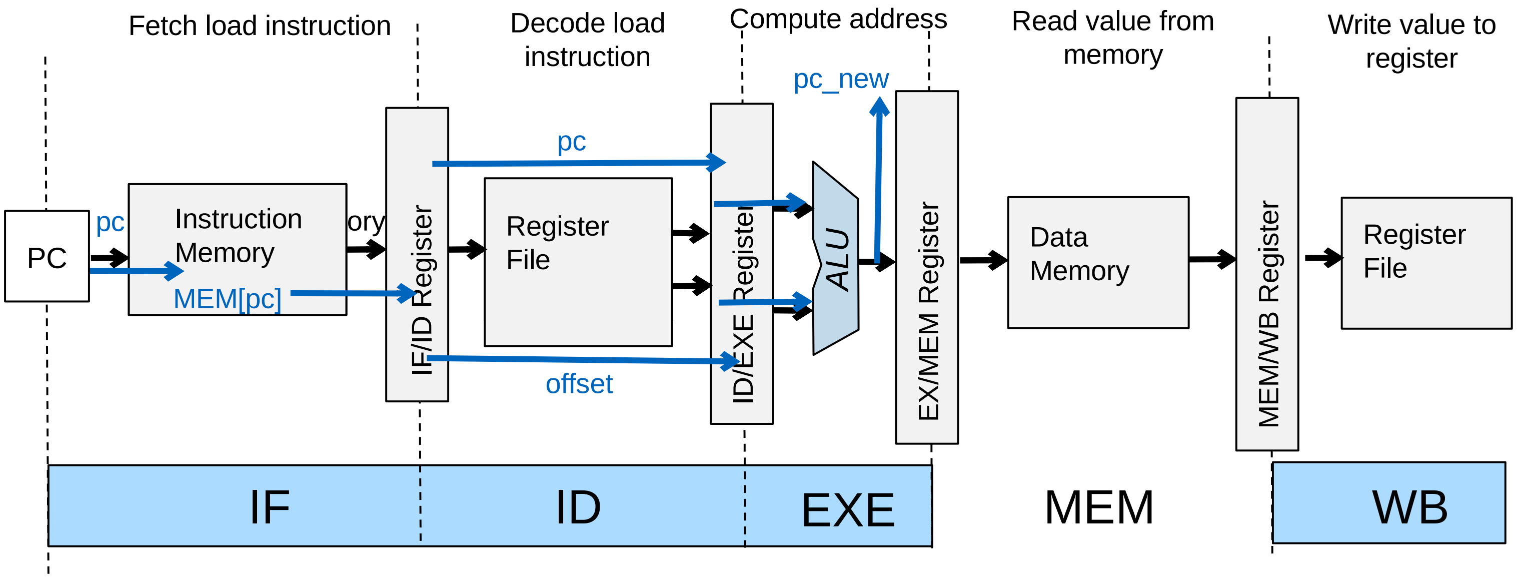
5-stage RISC Pipeline (4/5)

- Store instruction



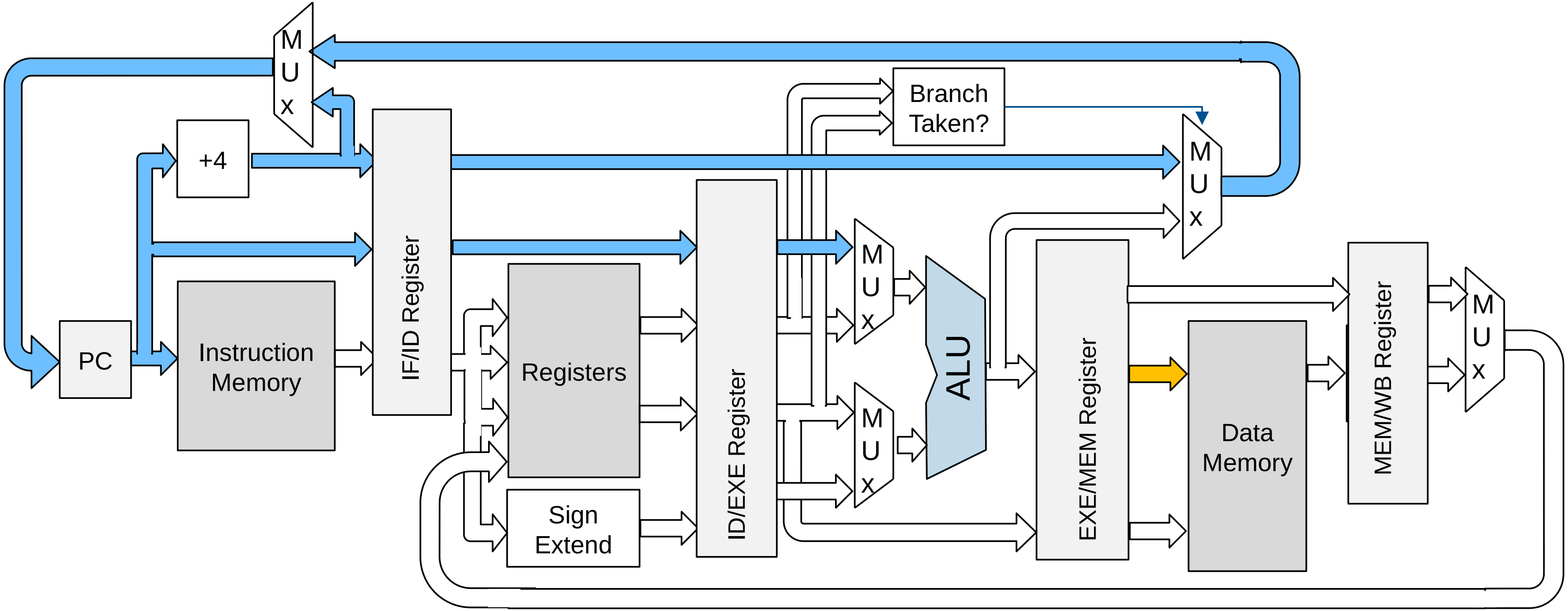
5-stage RISC Pipeline (5/5)

- Branch instruction



Pipelined Processor (5-stages)

- We introduce registers between the stages

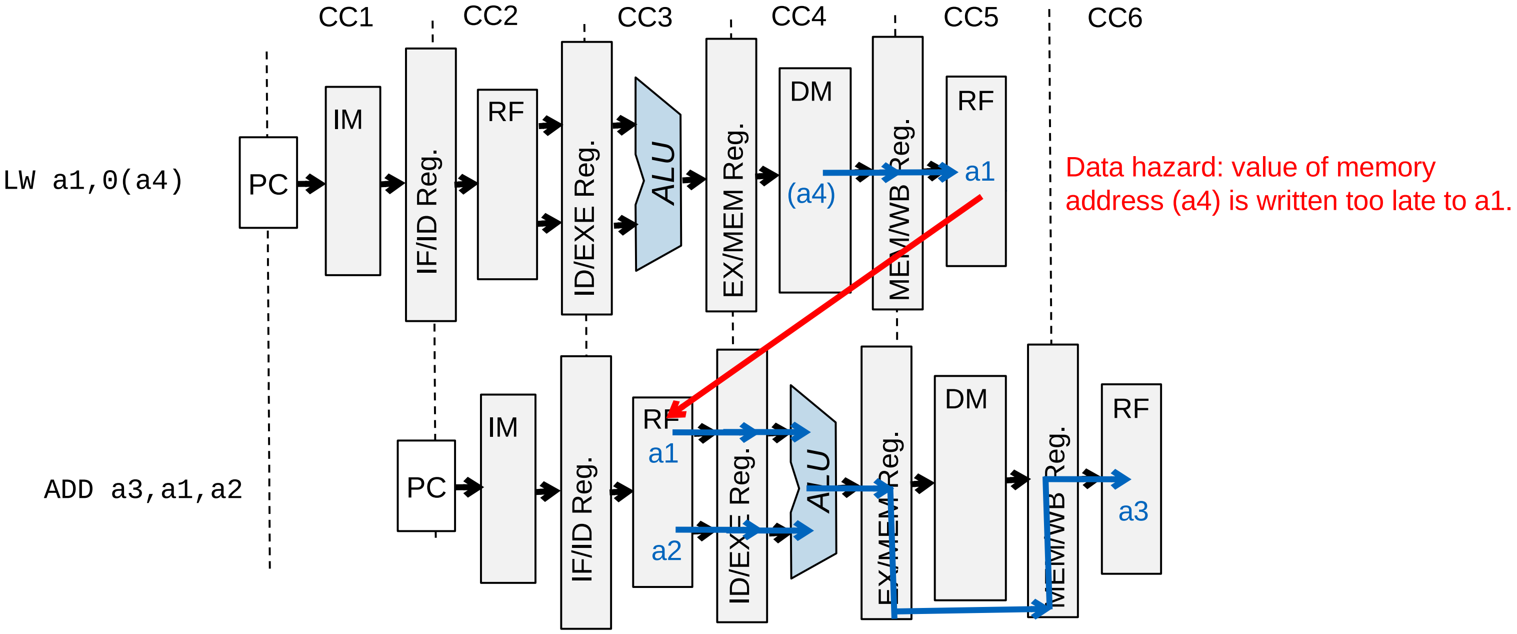


Pipeline Hazards

- Pipelined execution may insert additional cycles during execution due to
  - **Structural hazards**: Some resources are required in two stages of execution so two instructions may not use them at the same time
  - **Data hazards**: Some instruction wants to use a value that has not yet finished to be computed.
  - **Control hazard**: The result of a jump instruction is not yet computed.

Data Hazard after Load (1/3)

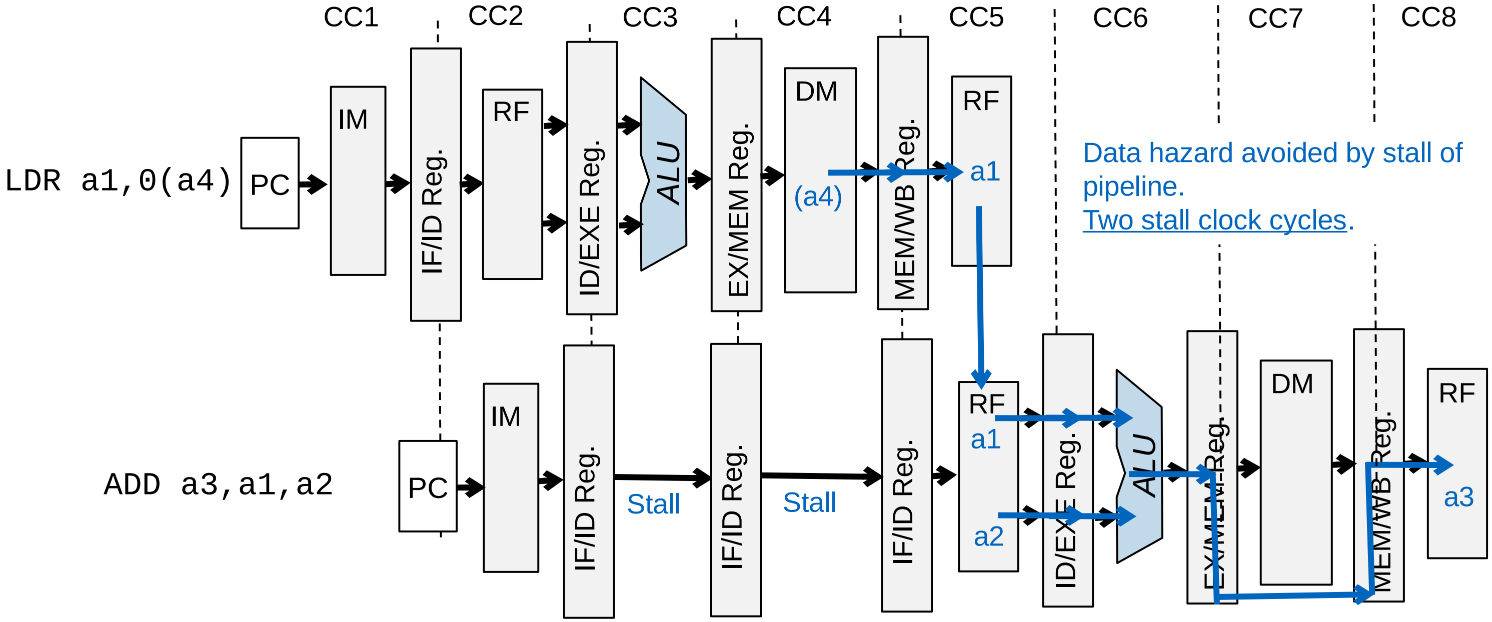
- Example for ALU after LDR data hazard



Data hazard: value of memory address (a4) is written too late to a1.

Data Hazard after Load (2/3)

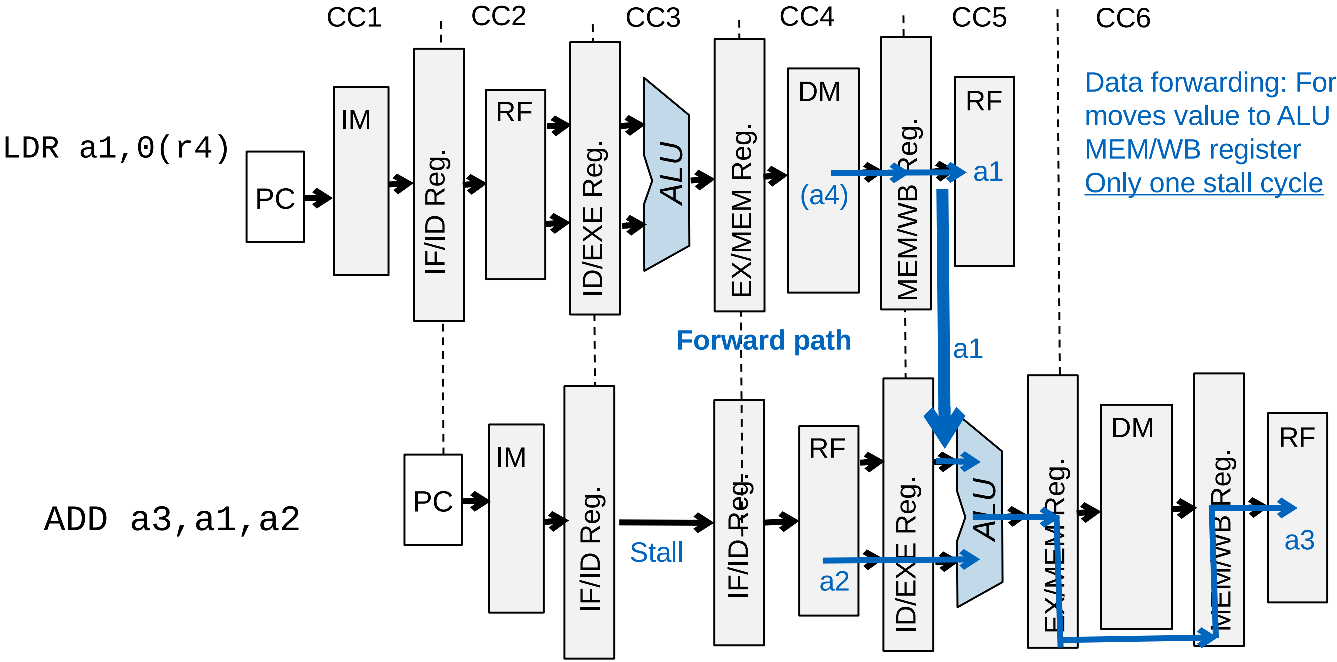
- Example for ALU after LDR data hazard



Data hazard avoided by stall of pipeline.
Two stall clock cycles.

Data Hazard after Load (3/3)

- Example for **Forwarding** for ALU after LDR data hazard

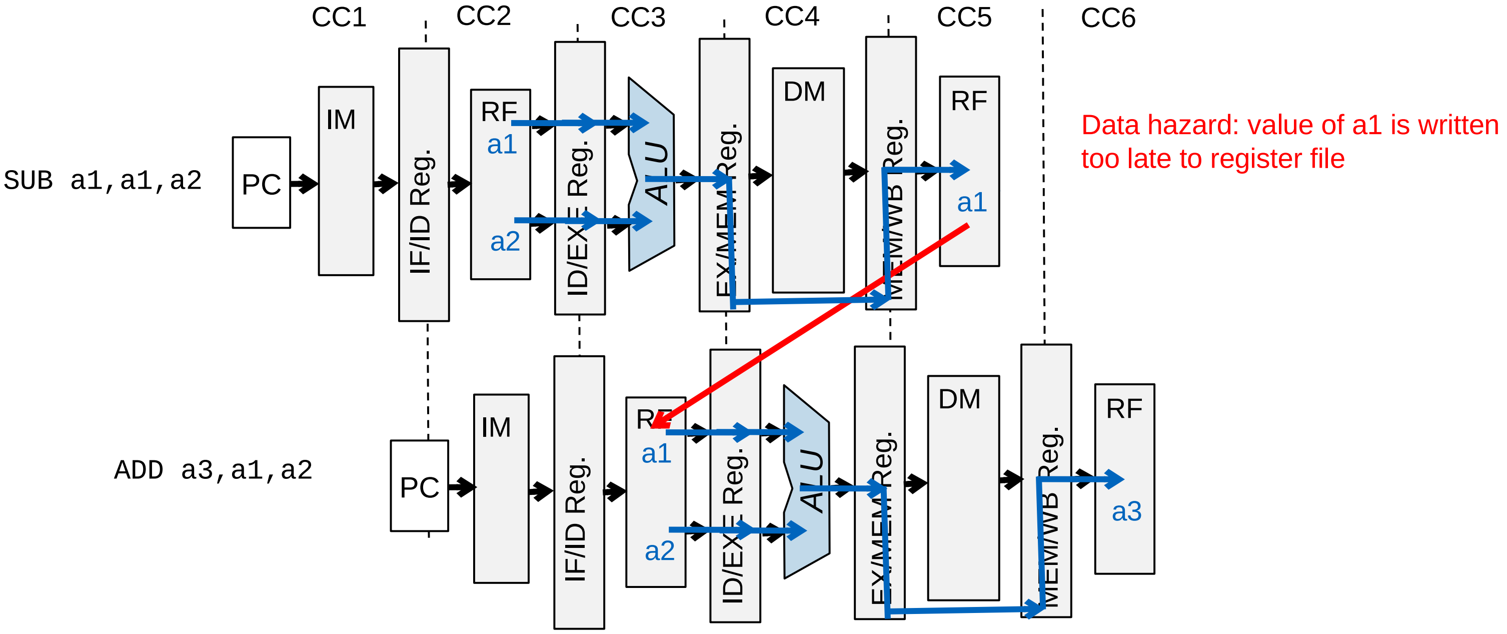


Data forwarding: Forwarding unit moves value to ALU from MEM/WB register

Only one stall cycle

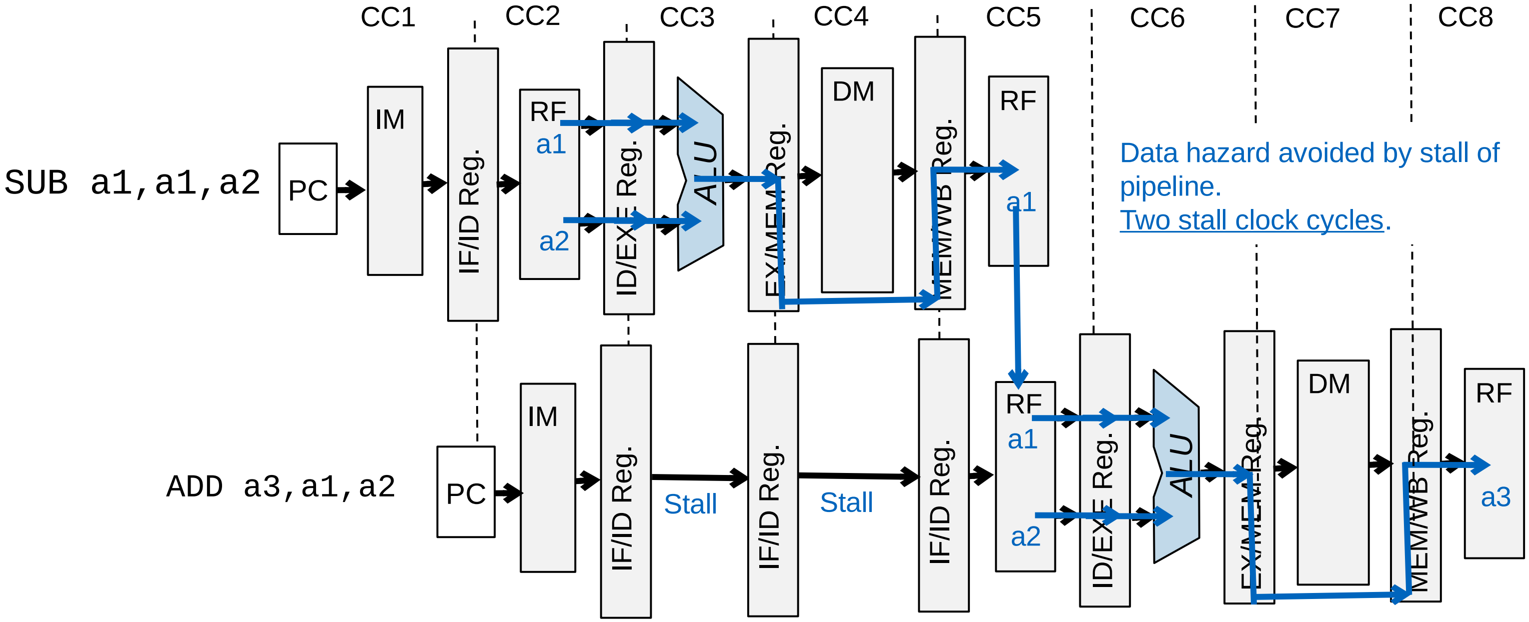
Data Hazard after ALU Operation (1/3)

Example for ALU after ALU data hazard



Data Hazard after ALU Operation (2/3)

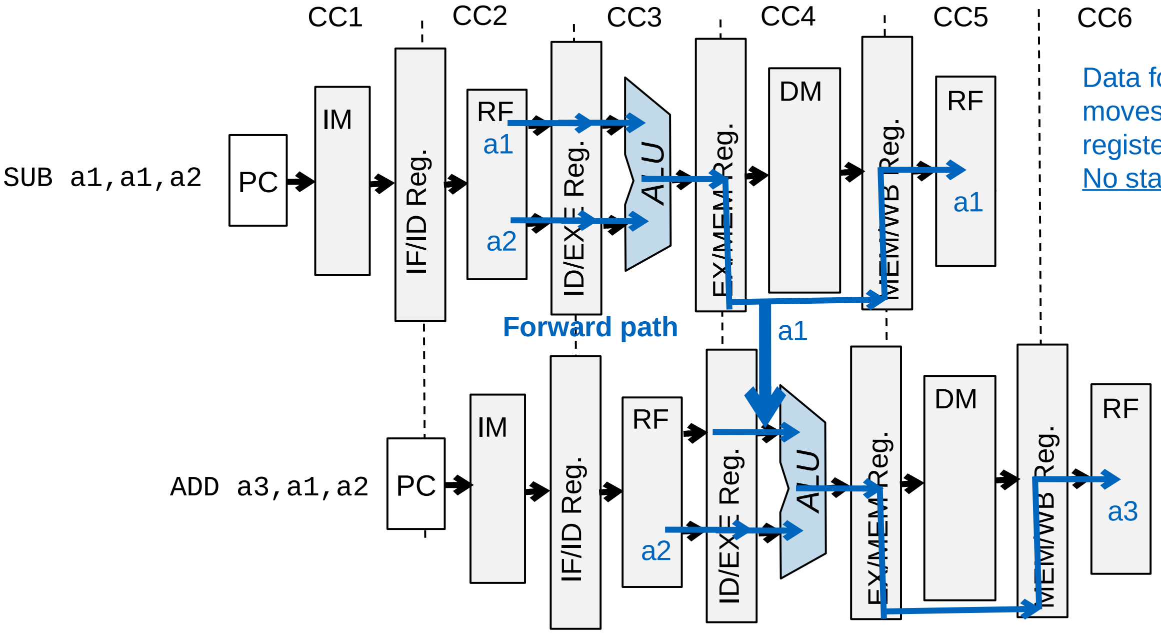
- Example for ALU after ALU data hazard



Data hazard avoided by stall of pipeline.
Two stall clock cycles.

Data Hazard after ALU Operation (3/3)

- Example for forwarding for ALU after ALU data hazard

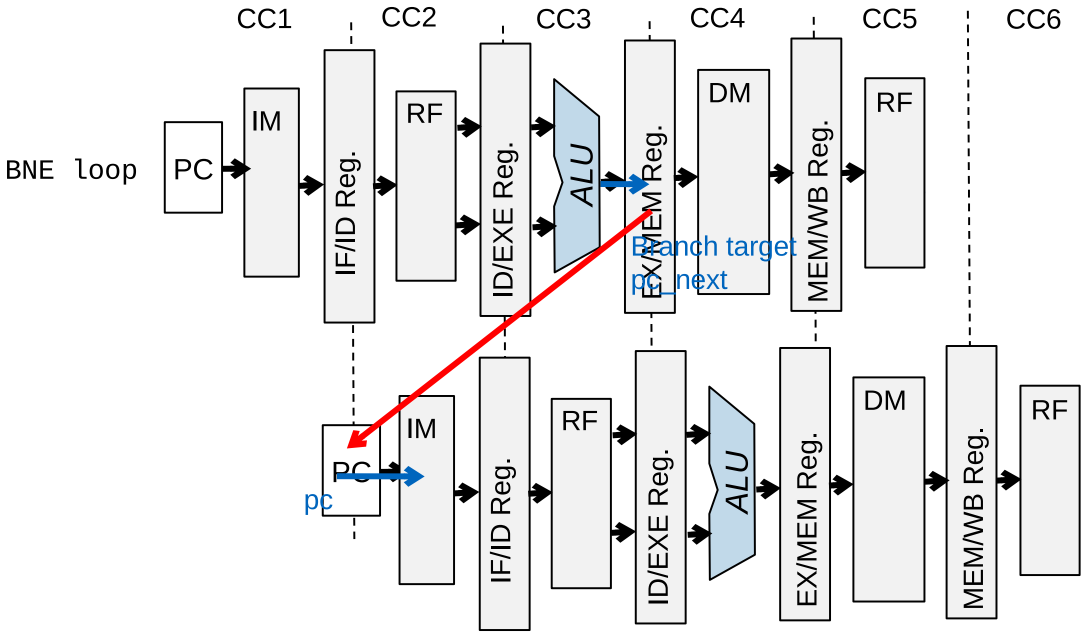


Data forwarding: Forwarding unit moves value to ALU from Ex/MEM register

No stall cycle

Control Hazard (1/2)

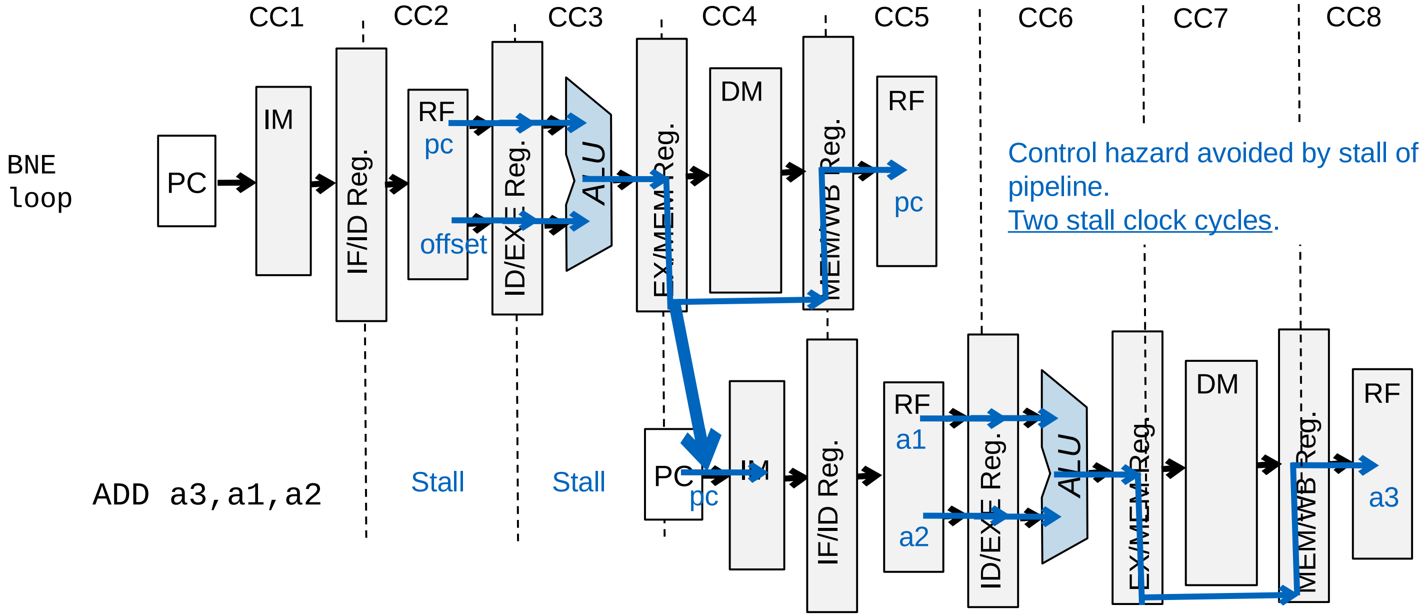
- Control hazard for conditional branches and jumps



Control hazard: Instruction fetched before new pc value is known.

Control Hazard (2/2)

- Control hazard for conditional branches and jumps

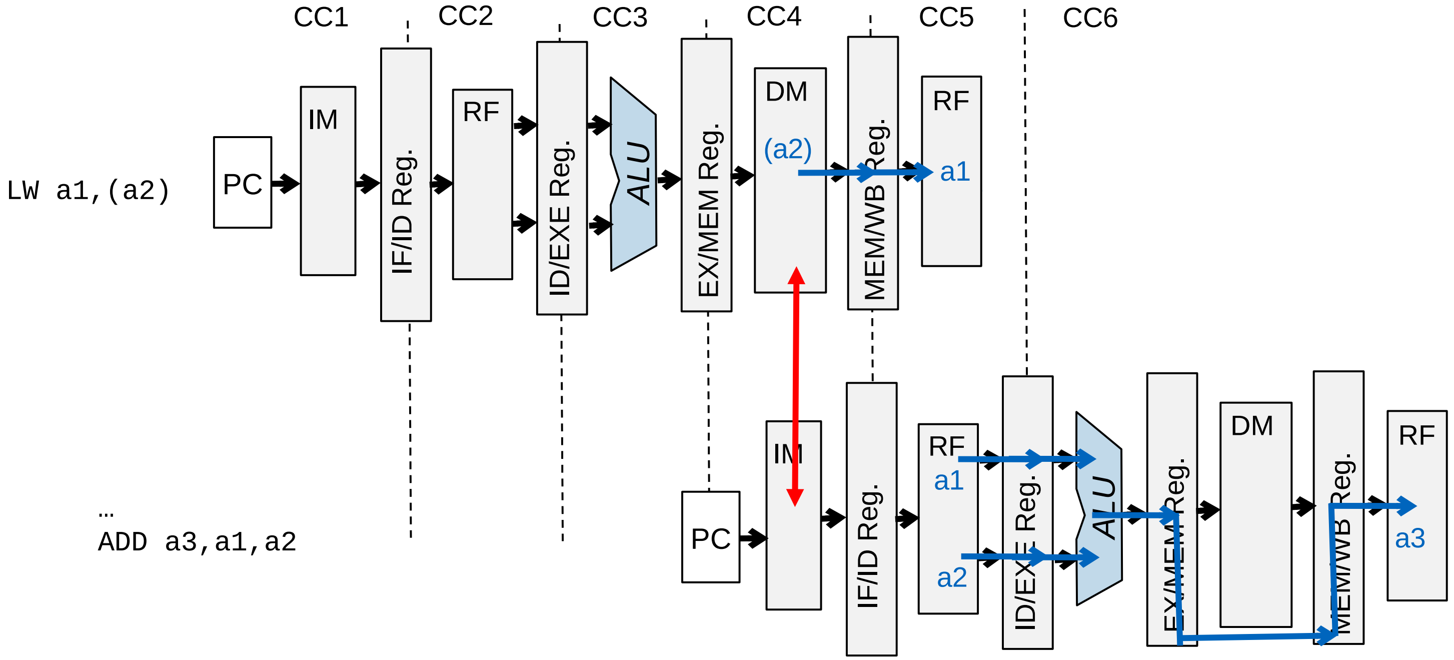


Control hazard avoided by stall of pipeline.
Two stall clock cycles.

Structural Hazard

Example: Von-Neumann architecture

Read data value and read instruction cannot be done in same clock cycle



Improving Pipeline Performance

- Data forwarding:
  - A forwarding unit moves data between pipeline registers.
  - reduces the number of clock cycles for stalls due to data hazards.
- Speculative execution:
  - Branch prediction:
    - A branch prediction unit predicts if branch is taken or not.
    - The predicted branch is started to be executed.
    - If the prediction was right, no stall cycle due to control hazard
    - If the prediction was wrong, instructions are flushed from the pipeline. New instructions are fetched. Additional cycles equal to stall cycles for control hazard.

Estimating Execution Time

Execution Time (1/4)

- Execution Time:

$$\boxed{t_{exe} = I \cdot CPI \cdot t_{clock} = \frac{I \cdot CPI}{f_{clock}} = \frac{I}{f_{clock} \cdot IPC}}$$

- Number of executed instructions: I
- Average number of cycles per instruction: CPI
- Average number of instructions per cycle: IPC
- Clock period: t_{clock}
- Clock frequency: f_{clock}

Execution Time (2/4)

Effects:

- Impact factors:

	Nr. Instr.	CPI	f_{clock}	Example of influences
Algorithm	x	(x)		Choice of source program instructions
Programming language	x	x		Efficiency of language
Compiler	x	x		Code Optimizations, Register Allocation
ISA	x	x	x	Type of machine instructions
Microarchitecture		x	x	Pipeline, Forwarding,
Implementation			x	technology
(Memory System)		x		Load/Store waiting times

Execution Time (3/4)

CPI different for different instruction classes

Execution time

$$\boxed{t_{exe} = (\sum_i I_i \cdot CPI_i) \cdot t_{clock}}$$

Number of executed instructions of type i: I_i

Average number of cycles per instruction of type i: CPI_i

Clock period: t_{clock}

Execution Time (4/4)

Example: $f_{clock} = 50\text{MHz}$

Instr class i	1	2	3
CPI_i	1	1.5	1.7
Instruction Count I_i	10000	20000	10000

Number of clock cycles:

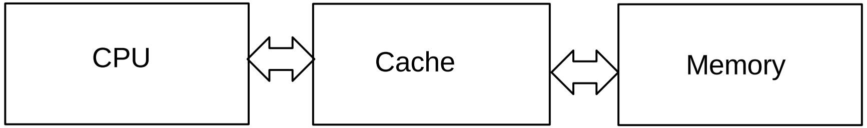
$$cc_I = 10000 \cdot 1 + 20000 \cdot 1.5 + 10000 \cdot 1.7 = 57000$$

Execution time:

$$t_{exe} = \frac{57000}{50\text{MHz}} = 1.14\text{ms}$$

Memory Systems (1/3)

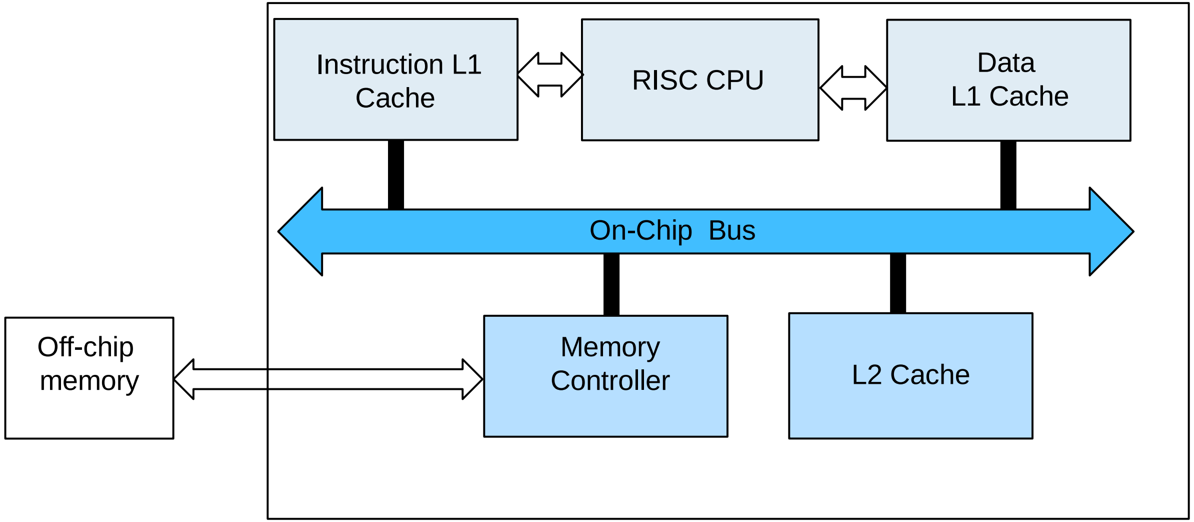
- Memory access times higher than clock frequency of CPU.
- Load/Store operations lead to stalls in program execution until value is read or written to memory.
- Addition of small fast memory that holds data most often requested by CPU (Caches).



Type	SRAM	DRAM
Access times (2004)	0,5-5ns	50-70ns
Cost per GB (2004)	4000-10000$	100-200$

Memory Systems (2/3)

Example with a memory hierarchy



Memory Systems (3/3)

- Block: Minimum size of information to be transferred between levels of the memory hierarchy.

- Hit rate: r_{hit} Fraction of memory accesses found in the cache
- Miss rate: r_{miss} Fraction of memory accesses not found in the cache $r_{hit} = 1 - r_{miss}$

- Hit time: t_{hit} Time to access the cache including time to check whether the access is a hit or a miss
- Miss penalty: t_{miss} Additional time to fetch block from lower level of memory hierarchy including time to access the block in case of a cache miss.

Cache Classification

- Cache organization:
  - Direct-mapped caches: Cache with one single location in which each block can be placed.
  - Set-associative cache: Cache with fixed nr. of locations (at least 2) in which block can be placed.
  - Fully-associative: Block can be placed in any location into the cache.
- Cache consistency:
  - Write-through: always write modified block from cache to memory on a store
  - Write-back: write modified block from cache to memory when block is replaced in the cache.
- Split-cache: Different cache for program and one for data.
- Split-memory: Different memory for program and data.

Execution Time with Memory System (1/3)

- Wait cycles to access a memory: $n_{wait,m} = \lceil \frac{t_{access,m} - t_{clock}}{t_{clock}} \rceil$

  Example: Cache + memory: $t_{access,cache} = t_{hit}$

  $t_{access,mem} = t_{hit} + t_{miss}$

  Memory-stall clock cycles: $cc_{stall,m} = M_m \cdot n_{wait,m}$

  Number of accesses to this memory: M_m

  Example: Only memory: $M_{mem} = I_{load} + I_{store}$

  Example: Cache +memory: $M_{cache} = r_{hit} \cdot (I_{load} + I_{store})$

  $M_{mem} = r_{miss} \cdot (I_{load} + I_{store})$

  Execution time with memory system:

$$t_{exe,memsys} = t_{clock} \cdot (\sum_i I_i \cdot CPI_i + \sum_m cc_{stall,m})$$

Execution Time with Memory System (2/3)

Example: $f_{clock} = 50\text{MHz}$ $t_{clock} = 20\text{ns}$

CPU:			
Instr class i	1	2	3
CPI_i	1	1.5	1.7
I_i	10000	20000	10000

$$cc_I = 10000 \cdot 1 + 20000 \cdot 1.5 + 10000 \cdot 1.7 = 57000$$

	M_m	$r_{miss,m}$	$t_{miss,m}$	$n_{miss,m}$	$cc_{stall,m}$
Instruction fetch	40000	2%	520ns	25	20000
Load Instructions	8000	4%	520ns	25	8000
Store Instructions	5000	4%	520ns	25	5000

$$t_{exe} = \frac{57000+20000+8000+5000}{50\text{MHz}} = 1.8\text{ms}$$

Execution time increased by 58%.

Without Memory System: $t_{exe} = \frac{57000}{50\text{MHz}} = 1.14\text{ms}$

Execution Time with Memory System (3/3)

Example with double frequency: $f_{clock} = 100\text{MHz}$ $t_{clock} = 10\text{ns}$

CPU:			
Instr class i	1	2	3
CPI_i	1	1.5	1.7
I_i	10000	20000	10000

$$cc_I = 10000 \cdot 1 + 20000 \cdot 1.5 + 10000 \cdot 1.7 = 57000$$

	M_m	$r_{miss,m}$	$t_{miss,m}$	$n_{miss,m}$	$cc_{stall,m}$
Instruction fetch	40000	2%	520ns	51	40800
Load Instructions	8000	4%	520ns	51	16320
Store Instructions	5000	4%	520ns	51	10200

$$t_{exe,M} = \frac{57000+40800+16320+10200}{100\text{MHz}} = 1.24\text{ms}$$

Execution time decreased by 31%.

For 50MHz: $t_{exe} = \frac{57000+20000+8000+5000}{50\text{MHz}} = 1.8\text{ms}$

Multi-issue processors (1/2)

- Multi-issue: Multiple instructions launched in one clock cycle.

- Static multi-issue processors:
  - Very Long Instruction word (VLIW) processors
  - Instruction word encodes more than one instruction
  - Compiler decides which instructions can be executed in parallel
  - Example: IA-64

- Dynamic multi-issue processors
  - Superscalar processors
  - Decision which instruction is started in each clock cycle is made during code execution
  - Dynamic pipeline scheduling to avoid data and branch hazards

Multi-issue processors (2/2)

- Instruction-level parallelism
  - Instructions with no data/control dependencies can be executed in parallel
  - Parallelism can be seen from sequencing graph
  - Scheduling as in HW-Synthesis

- Software (loop) pipelining:
  - Increase instruction-level parallelism by executing iterations of a loop in parallel
  - Check for data dependencies in Do-across loops

- Thread parallelism:
  - Execute instructions from different threats in parallel

On-chip Buses

Memory-mapped Buses

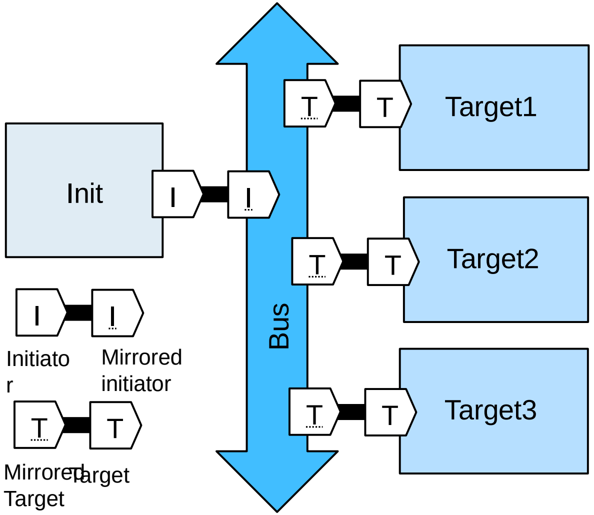
- Purpose:
  - Read or write a value from or to a certain address
  - Value can be data or peripheral control information
- Memory-mapped Bus has several (sub-)buses (group of signals) and a defined bus protocol
  - Address bus
  - Data bus for reading data
  - Data bus for writing data
  - Control signals: Indicate if access is read or write, bust length, ID, bus grant, …
- Modules on the bus can either act as initiators or targets
  - Typical initiators: CPUs, DSPs, DMAs, bus bridges, …
  - Typical targets: Memory, accelerators, interface peripheral, bus bridges, …

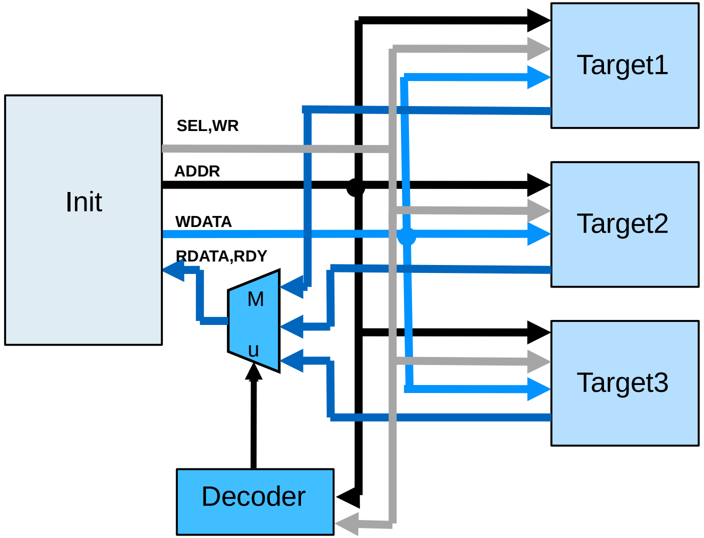
Classes of Memory-mapped Buses

- Single-initiator bus:
  - One initiator component can address different target components, which are mapped to different addresses.
- Shared bus:
  - There are several initiators on the bus.
  - An arbiter decides which initiator module is granted access to the bus
  - Only one initiator can access one slave via the bus at a time
- Layered bus:
  - There is more than one arbiter such that more than one initiator is granted access on the bus.
  - Only one target component on each layer can be accessed at a time
- Crossbar/ bus matrix
  - Each target component has its own arbiter
  - Each target component can be accessed by one initiator at a time

Single-Initiator Bus

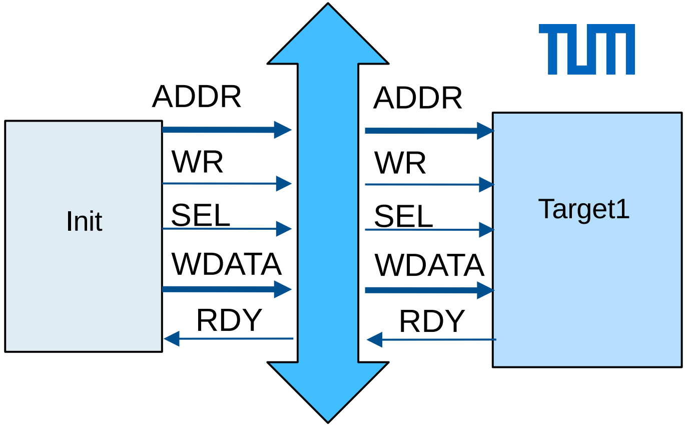
- Target knows if it is addressed by observing the address bus ADDR and can receive data on write data bus WDATA
- Decoder forwards the data from the addressed target by multiplexing it to the read data bus RDATA
- Additional control bus CTRL for signals related to bus protocol (e.g. WR, SEL, RDY)



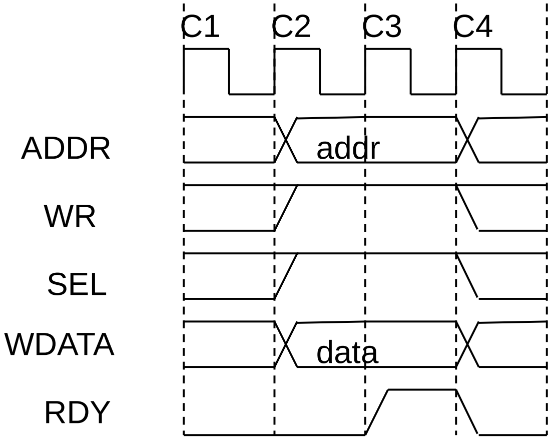


Simple Write Access

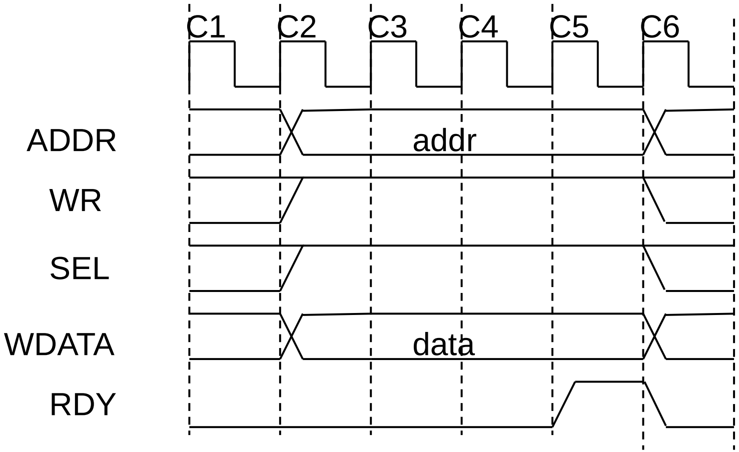
1. Initiator places address and data on the ADDR and WDATA bus
   Initiator indicates write by setting signal WR to high
   Initiator indicates that access is started by setting SEL signal to high
2. Target acknowledges write access by RDY signal



No wait cycles

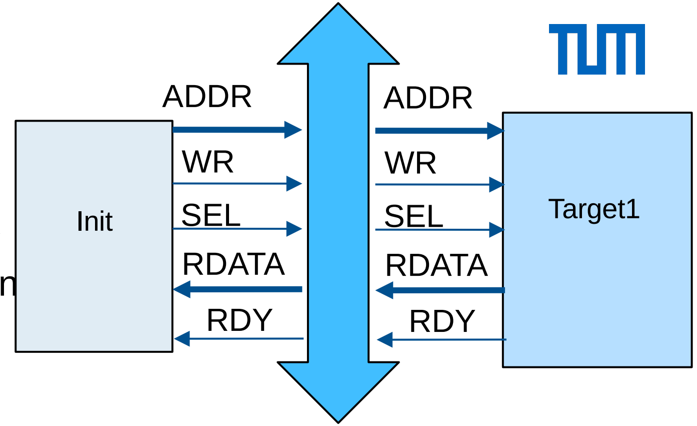


Two wait cycles

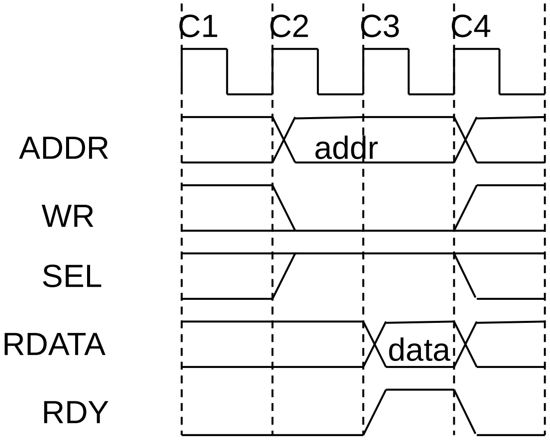


Simple Read Access

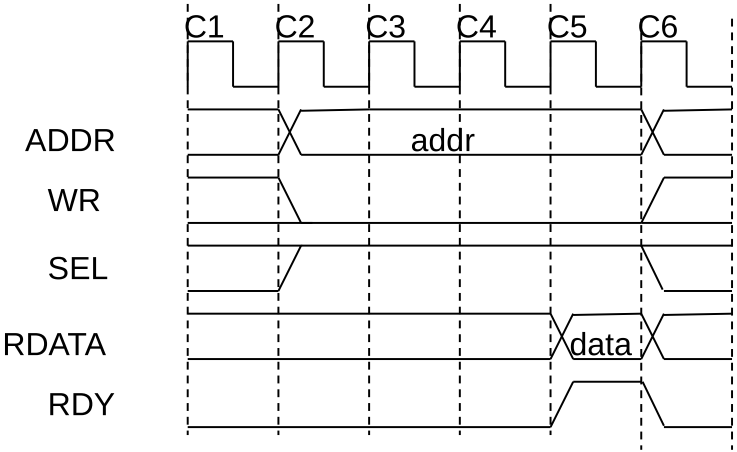
1. Initiator places address on the ADDR bus
   Initiator indicates read access by setting signal WR to low
   Initiator indicates that access is started by setting SEL sign
2. Target places data on RDATA bus
   Target acknowledges write access by RDY signal



No wait cycles



Two wait cycles

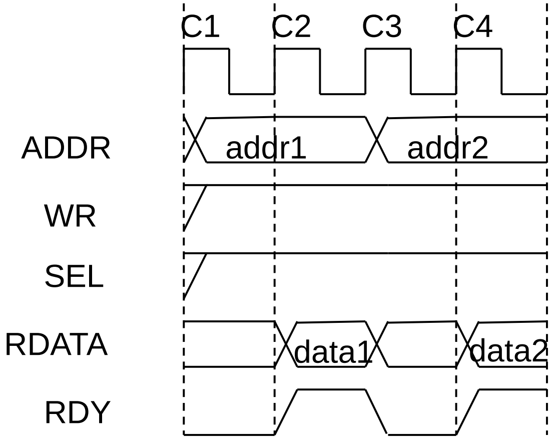


Performance of Simple Accesses

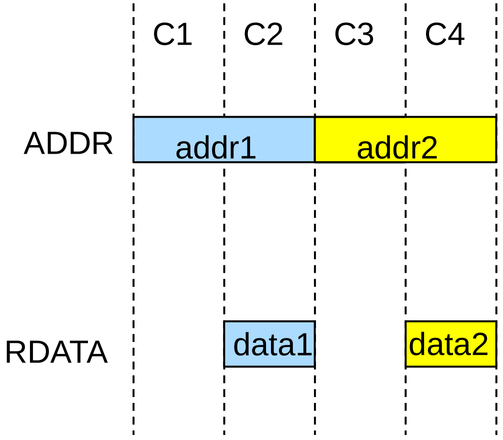
Each access takes minimally two cycles
Maximal bus bandwidth is:

$$BW_{bus} = 0.5 \cdot buswidth \cdot f_{bus}$$

Two read accesses



Two read accesses (bus access diagram)



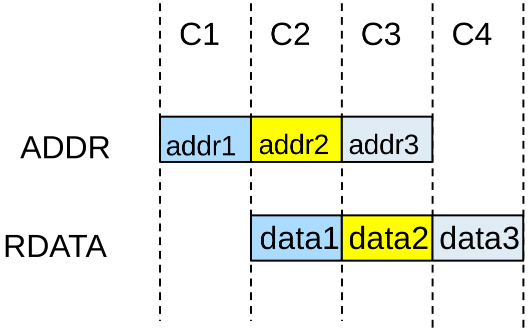
Pipelined Accesses

- The next address can be placed on the bus while the data is read
- Maximal bandwidth supported by bus is equal to:

$$BW_{bus} = buswidth \cdot f_{bus}$$

- Additional control signals and logic required to support pipelined accesses.

Three pipelined read accesses



C1 C2 C3 C4

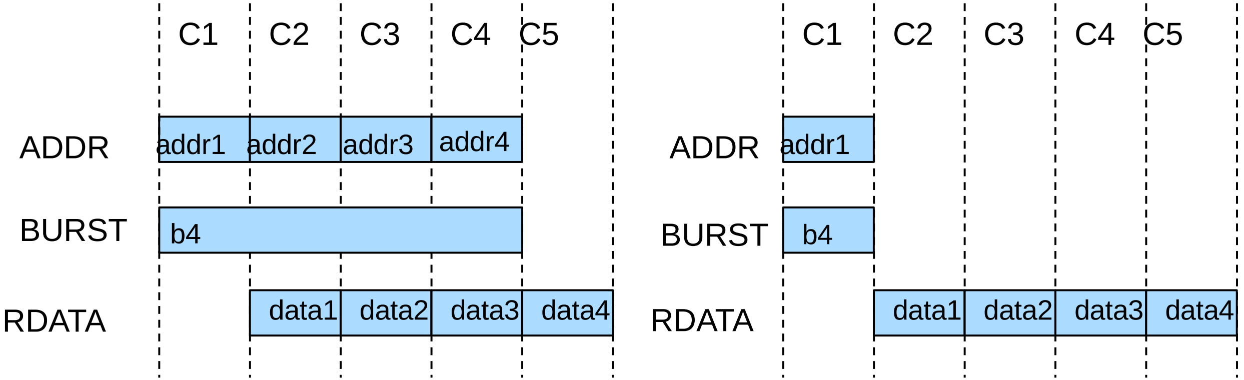
ADDR: addr1 addr2 addr3

RDATA: data1 data2 data3

Burst Accesses

- A burst accesses a consecutive row of addresses
- Version 1: the addresses for all accesses must be given and a control signal that indicates that this is a burst access of a certain size
- Version 2: Only the start address must be given and a control signal that indicates that this is a burst access of a certain size

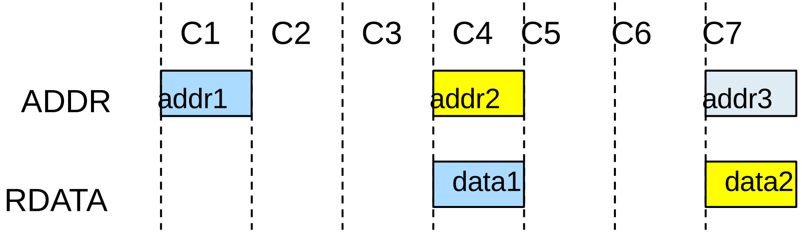
Four data values are returned for one start address (burst4)



Multiple Outstanding Transactions

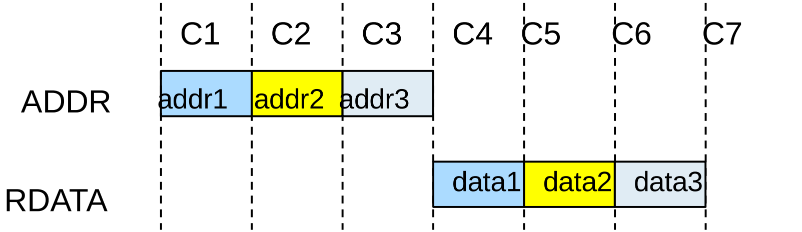
- A address may be placed on the bus before the data of the previous access has been read or be written.
- This improves performance in case of wait cycles.

No outstanding transactions (two wait cycles)



	C1	C2	C3	C4	C5	C6	C7
ADDR	addr1			addr2			addr3
RDATA				data1			data2

With multiple outstanding transactions (two wait cycles)

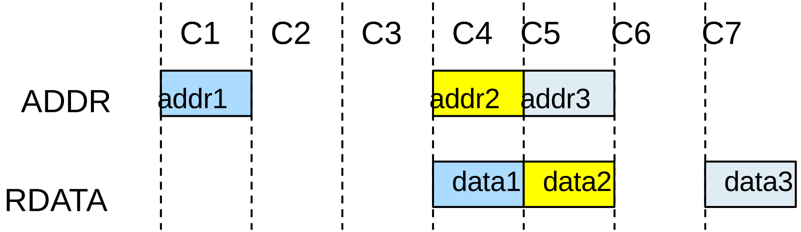


	C1	C2	C3	C4	C5	C6	C7
ADDR	addr1	addr2	addr3				
RDATA				data1	data2	data3	

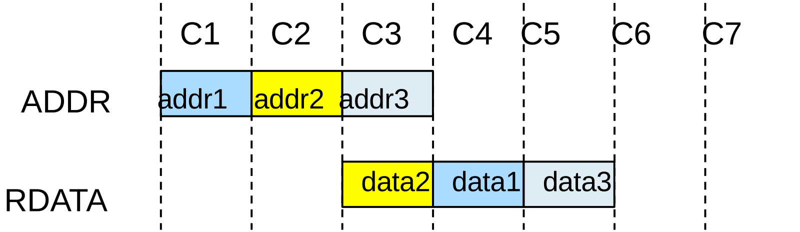
Out of order completion with interleaving

- A address may be placed on the bus before the data of the previous access has been read or be written.
- In case of wait cycles, the order of data reads may be changed

No out of order completion with interleaving

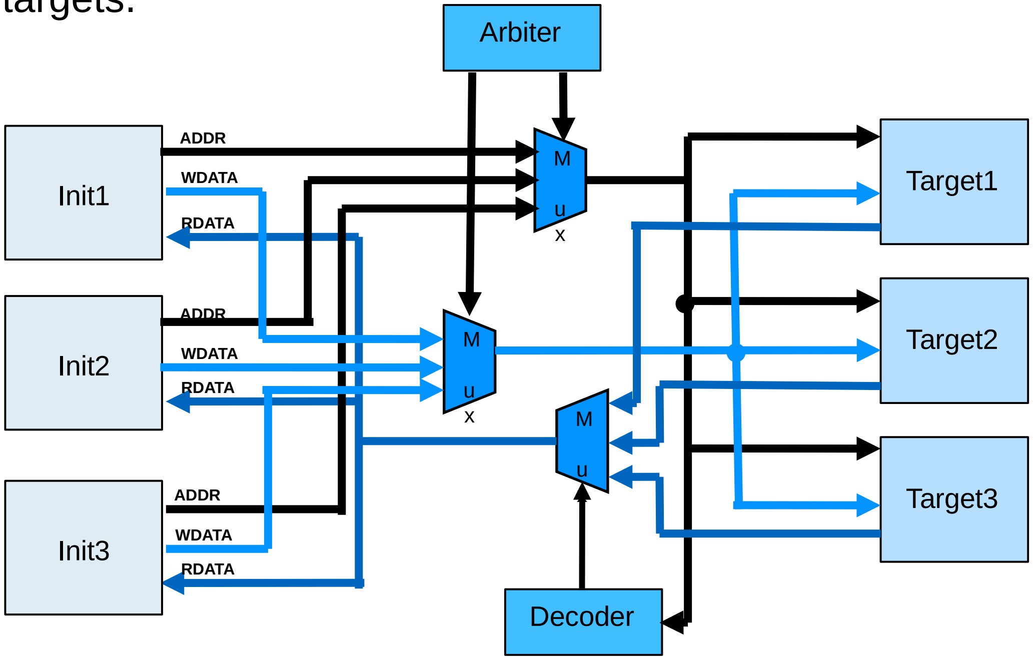


With out of order completion with interleaving



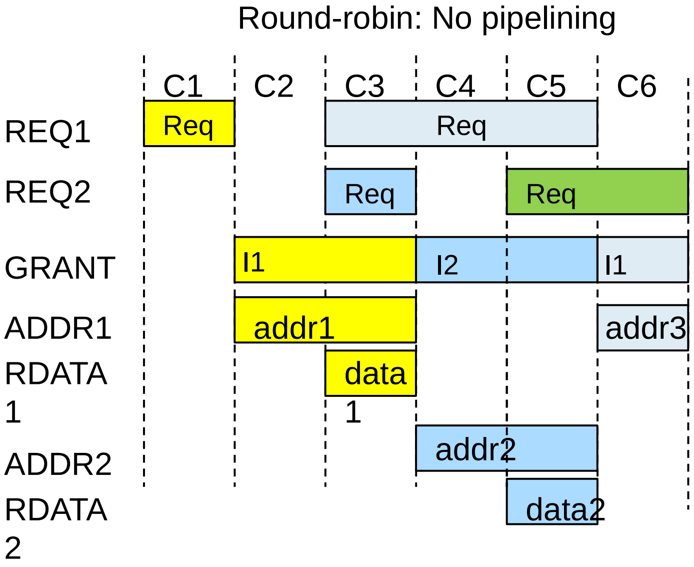
Shared Bus

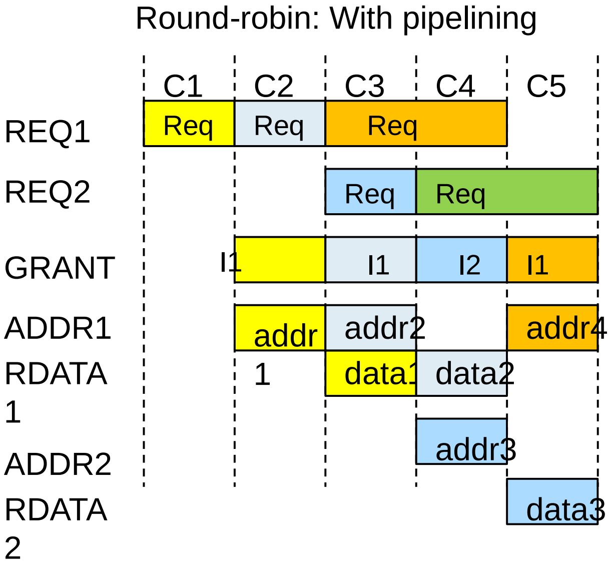
- Arbiter grants access to the initiator:
- Only the address and data of one initiator is forwarded to the targets.



Bus Arbitration

- The arbiter grants access to initiator that request the bus
- Round-robin: Access granted to initiators in pre-defined order that is repeated.
- FIFO: First initiator requesting the bus is granted access.
- Priority: Initiator with highest priority is granted access to the bus

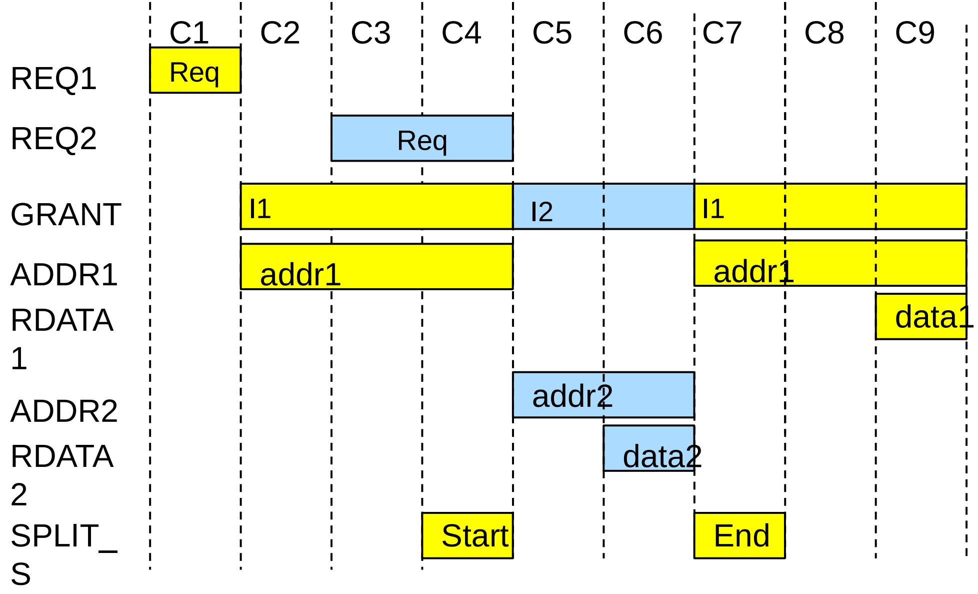




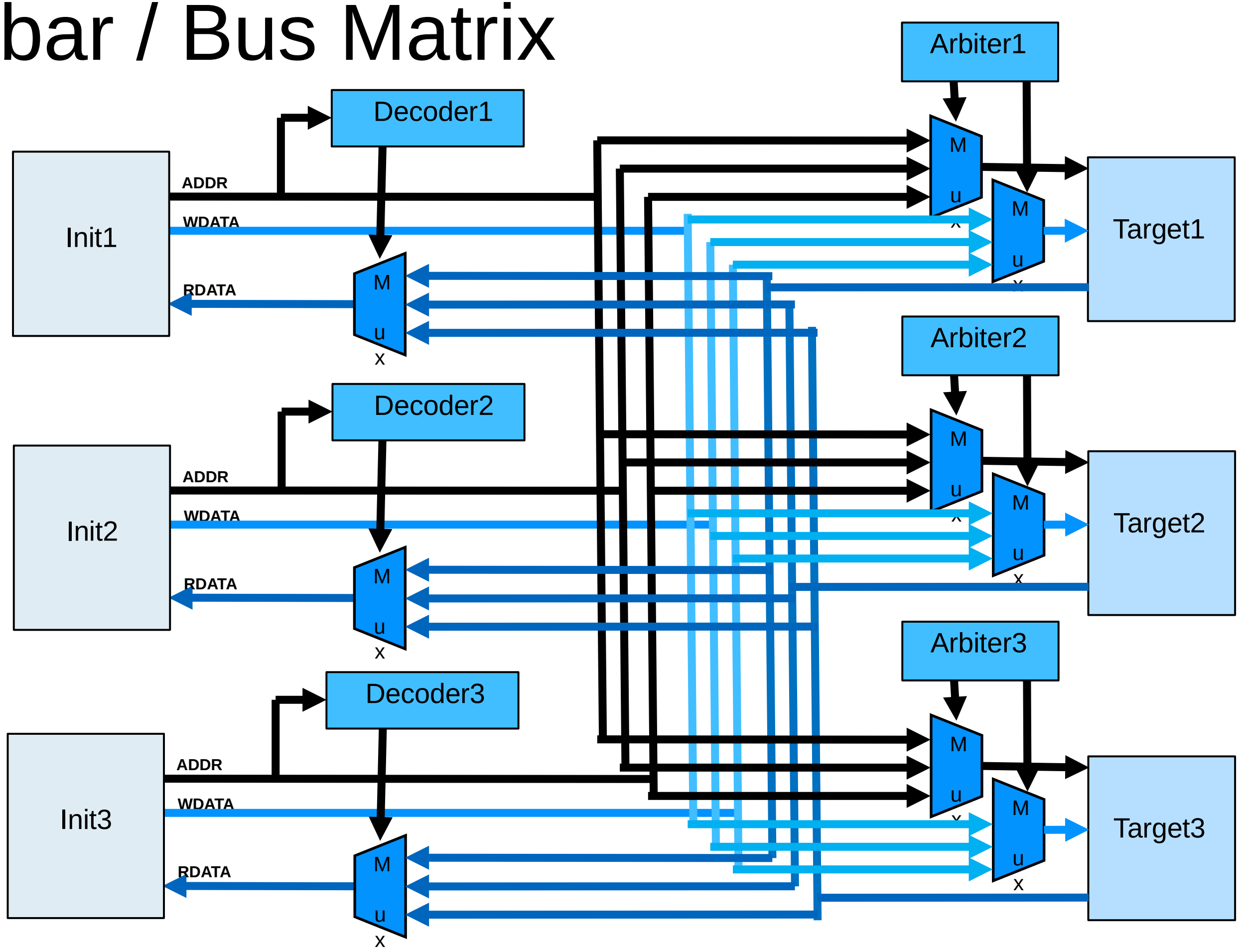
Split Accesses

Slave can allow a split of an access if it was many wait cycles

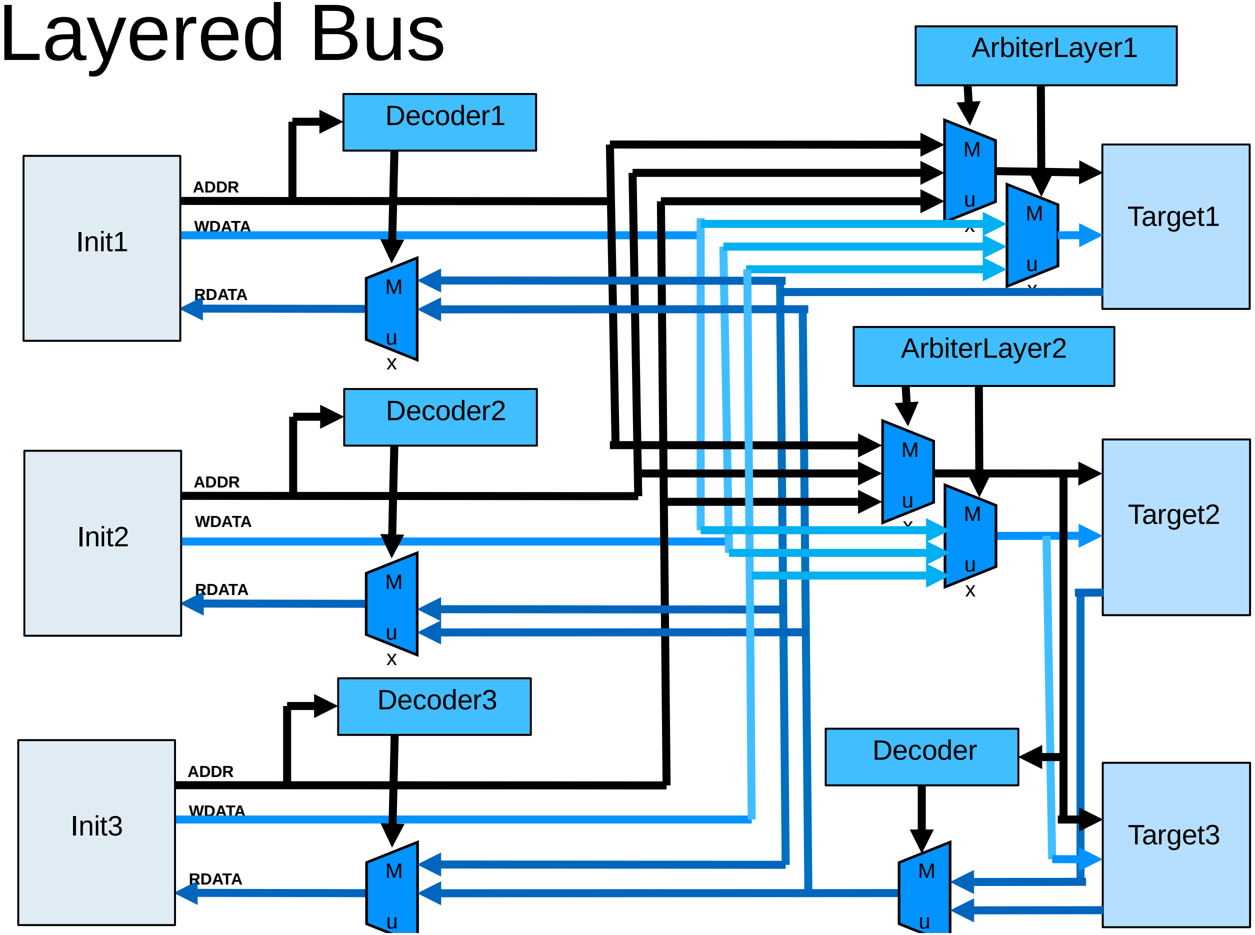
Access of initiator I1 is split by issuing a start of split by slave
I2 is granted the bus and access of initiator I2 is performed
Then access of initiator I1 is finished by issuing an end of split.



Crossbar / Bus Matrix



Layered Bus



- Targets are on different layers.
- initiator can connect to targets on different layers simultaneously.

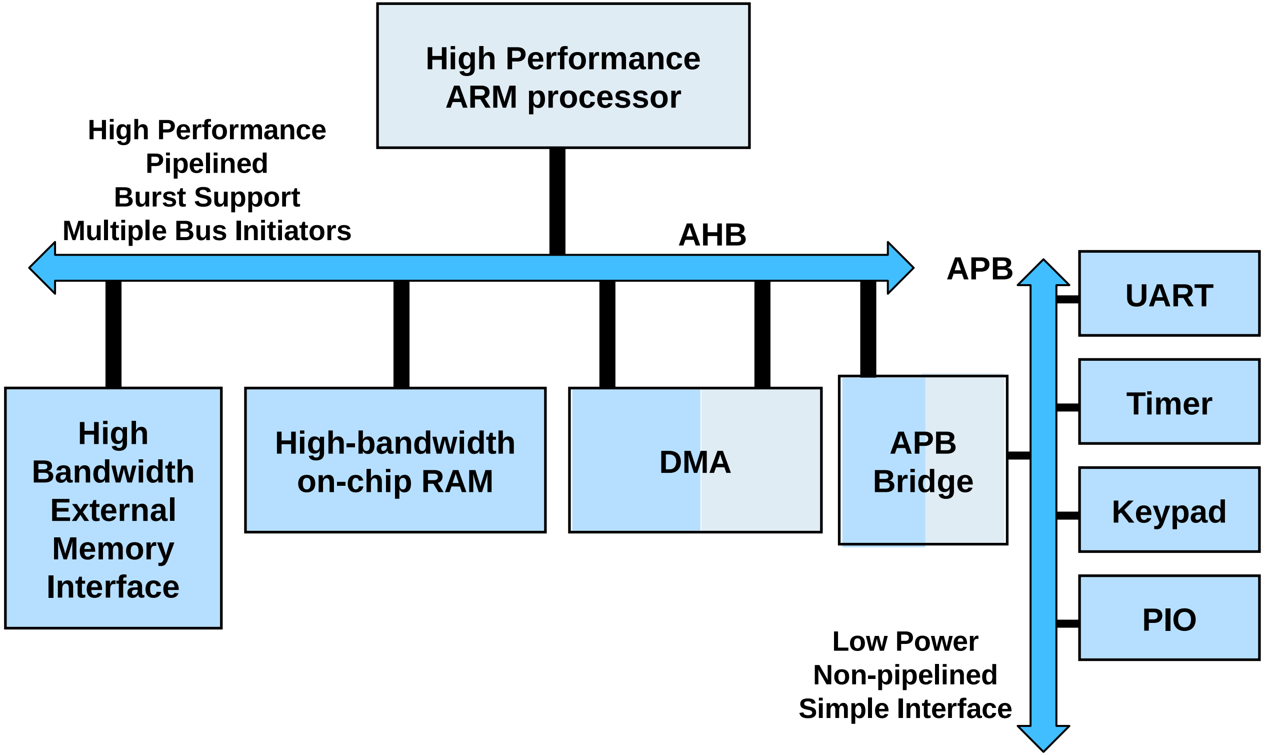
Bus Standards

- IBM Core Connect
  - PLB: Processor Local Bus
  - OPB: On-chip Peripheral Bus
- AMBA Bus
  - AHB: Advanced High Performance Bus
  - APB: Advanced Peripheral Bus
  - AXI: Advanced eXetendible Interface
- Open Core Protocol (OCP)
  - Socket-base On-chip Interface standard
  - Only defines the interface to the components, not the architecture of the bus.

ARM AMBA Standard

- Different Versions e.g., AMBA 2,0, AMBA 3.0,…
- AHB: Advanced High Performance Bus
  - High performance
  - Pipelined operation
  - Multiple bus initiators
  - Burst transfers
  - Split transactions
- APB: Advanced Peripheral Bus
  - Low power
  - Simple Interface
  - Suitable for many peripherals
  - One initiator (APB Bridge)
- AXI: Advanced eXetendible Interface
  - Configurable channel-based specification

An Example AMBA System



HW/SW Interface

Software Hierarchy

- **Application Software**: Software Code of tasks to run on processor
- **System Software**:
  - Provides communication and runtime environment
  - Operating System (OS), Real Time Operating System (RTOS)
  - Device drivers
  - Hardware Abstraction Layer (HAL): Lowest level of SW that abstracts hardware behavior
- If there is no OS: Bare-metal application

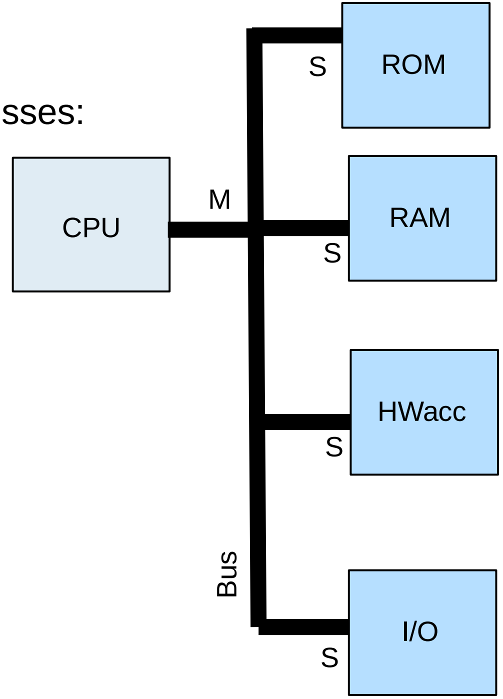
On-chip Communication

- Initiator to target:
  - A initiator such as CPU can access a target such as peripherals, memories, IOs, HW accelerators via the bus
  - Bus targets are assigned address regions in the memory space addressable by the bus initiators
- Initiator to initiator:
  - Shared memory region addressable by both initiators
  - Message passing with message buffers for both initiators
- Target to initiator:
  - A target and also an initiator can notify another initiator by issuing an interrupt
  - The initiator can poll an status register of a target

Processor Memory Map (1/2)

Example:

CPU has 32-bit addresses:
0x00 00 00 00
…
0xFF FF FF FF



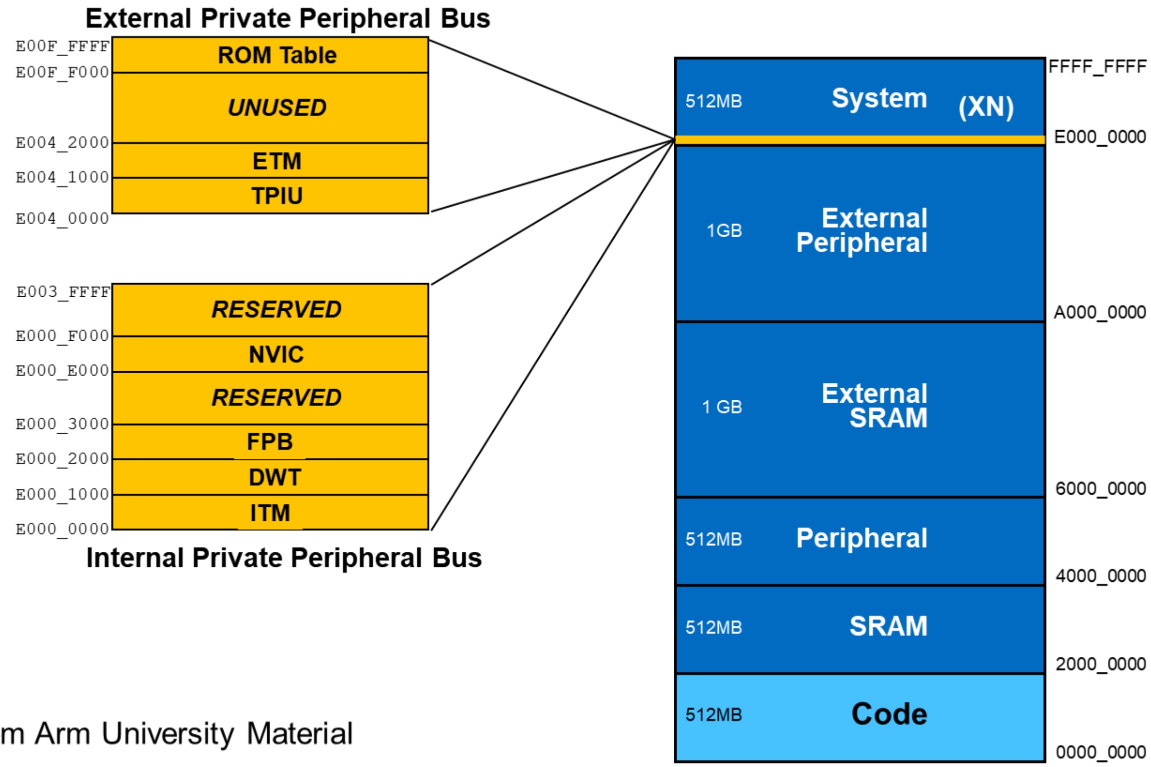
ROM (262k) mapped to:
0x00 00 00 00
…
0x00 03 FF FF

RAM (1MB) mapped to:
0x10 00 00 00
…
0x10 0F FF FF

HWacc (4 byte) mapped to:
0x50 00 00 00
…
0x50 00 00 10

I/O mapped to:
0x60 00 00 B0
…
0x60 00 00 B8

Example ARM Memory Map



From Arm University Material

Peripheral Register Interface (1/4)

- The addressable space of a peripheral is divided into registers with given address
- Each register can further be divided into bitfields

- Each register / bitfield may contain information regarding input data, output data, status and configuration of the peripheral

Peripheral Register Interface (2/4)

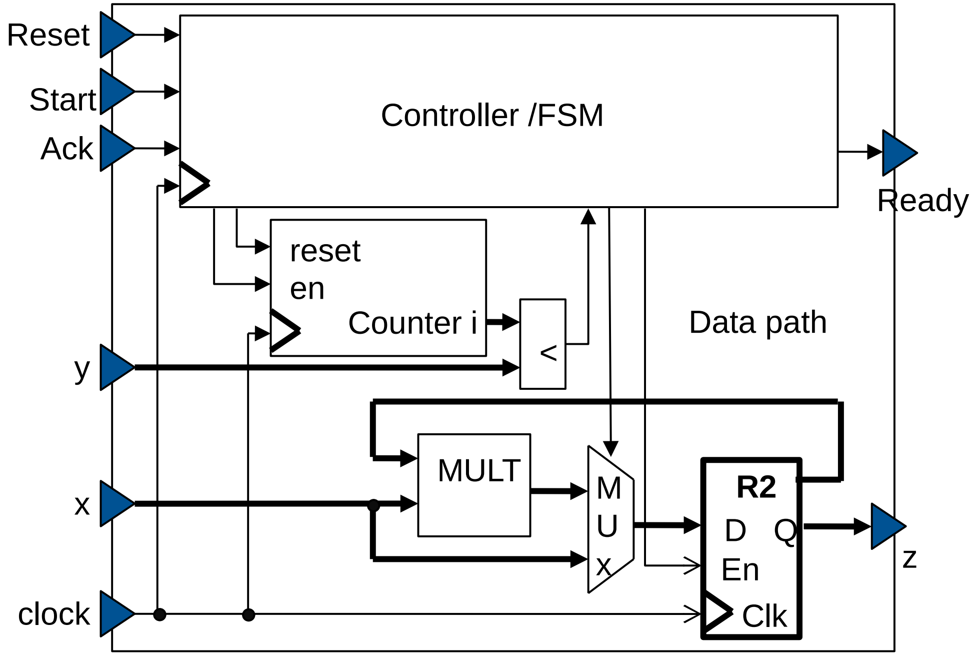
- Registers and bit fields can be:
    - Read only to CPU (data output and status registers)
    - Read / writeable by CPU (data input and configuration registers)
    - Certain registers may have two addresses
        - On one address register is read only
        - On another address register might be writable

- Access to these fields can trigger certain on-access actions:
    - Clear on read: Value is set to default after it was read accessed
    - Ignore read value: If a certain value is written for a bit field, it is ignored. Can be useful, if only one out of several bit fields in a register should be changed.
    - …

Peripheral Register Interface (3/4)

Example: HW accelerator for $z=x^y$

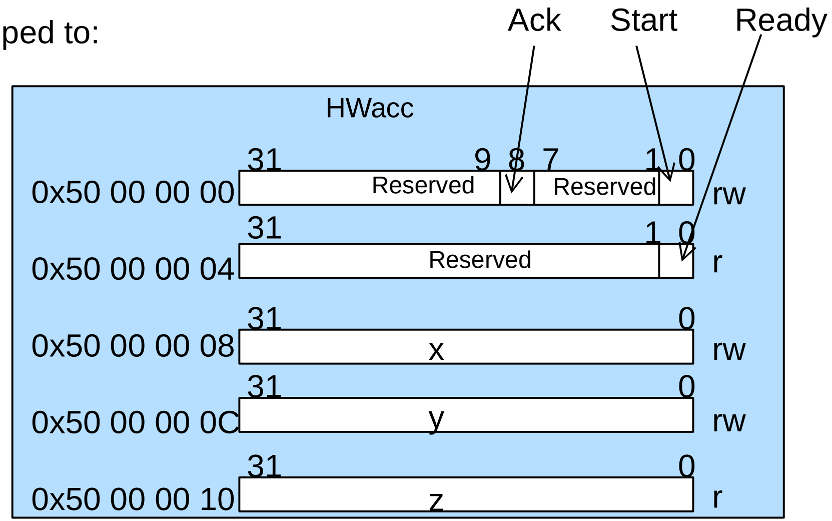
```
int power(int x, int y)
{
    int z=x;
    for (int i=1;i<y;i++)
    {
        z=z*x;
    }
    return z;
}
```



Peripheral Register Interface (4/4)

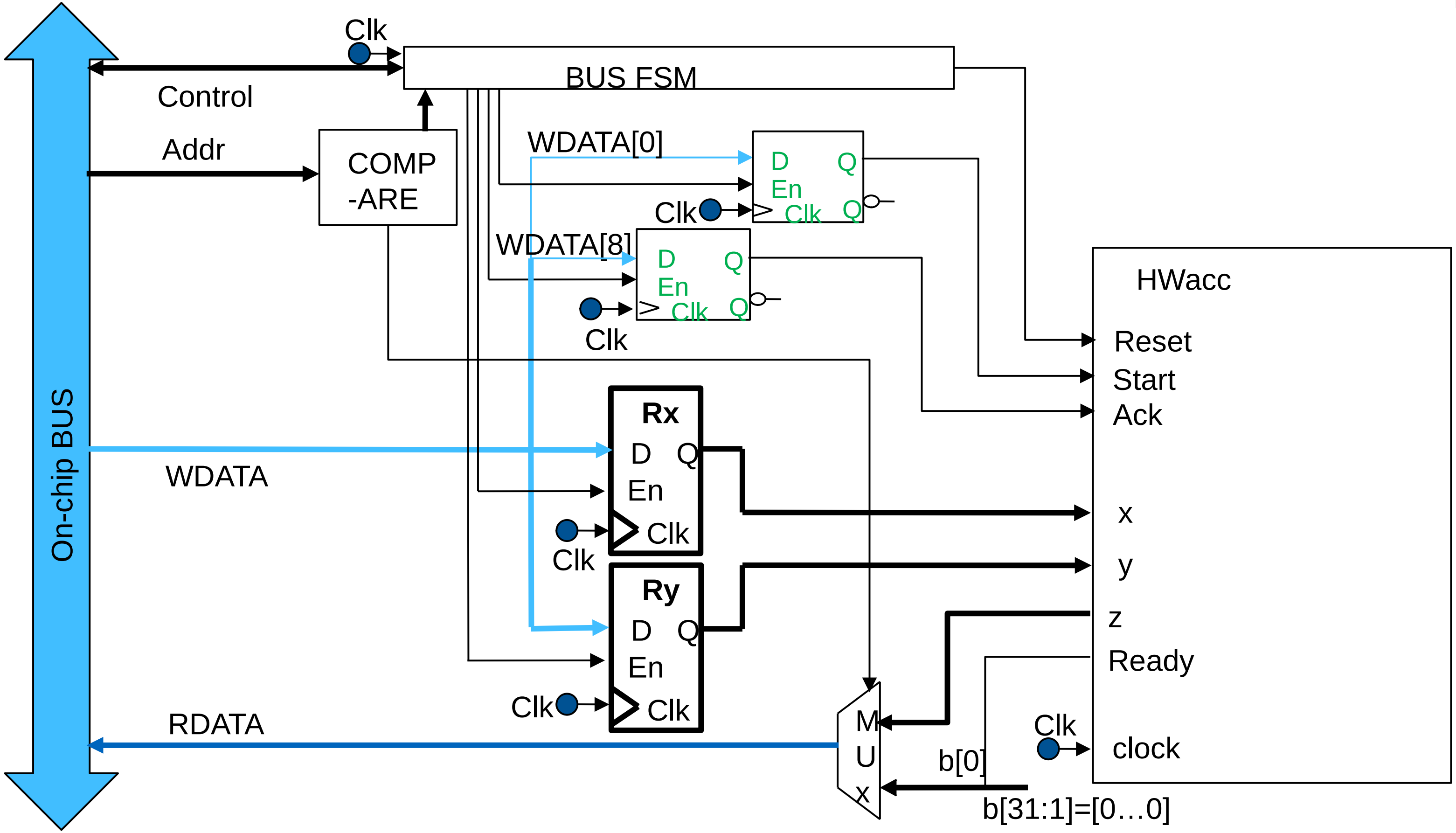
Example:

HWacc (4 byte) mapped to:



Address	Bits	Fields	Access
0x50 00 00 00	31–9, 8, 7–1, 0	Reserved (31–9), Ack (8), Reserved (7–1), Start (0)	rw
0x50 00 00 04	31–1, 0	Reserved (31–1), Ready (0)	r
0x50 00 00 08	31–0	x	rw
0x50 00 00 0C	31–0	y	rw
0x50 00 00 10	31–0	z	r

Peripheral Bus Interface



SW Representation of Register IF

- Several ways
  - Define pointers to all addresses of registers
  - Define a struct
  - **volatile** keyword: Data may be changed by other sources than the processor (important for compiler optimization)

Example:

```
struct Hwacc_data {
uint32_t config;
uint32_t status;
uint32_t x;
uint32_t y;
uint32_t z;
}
#define HWACC_ADDRESS 0x50000000;
// Initialize with
volatile Hwacc_data* Hwacc_data_ptr = (Hwacc_data*) HWACC_ADDRESS;
```

Access functions (Write)

- Write functions for input data

```
void Hwacc_write_x(int x)
{
    Hwacc_data_ptr->x = x;
}
void Hwacc_write_y(int y)
{
    HWacc_data_ptr->y = y;
}
```

Access functions (Get/Set Function)

Get/Set functions for configuration of peripheral
Example:

```
void Hwacc_set_start()
{
    uint32_t config = Hwacc_data_ptr->config;
    config = config | 0x0000 0001;
    Hwacc_data_ptr->config = config;
}
void Hwacc_set_ack()
{
    uint32_t config = Hwacc_data_ptr->config;
    config = config | 0x0000 0100;
    Hwacc_data_ptr->config = config;
}
```

```
void Hwacc_unset_start()
{
    uint32_t config = Hwacc_data_ptr->config;
    config = config & 0xFFFF FFFE;
    HWacc_data_ptr->config = config;
}
void Hwacc_unset_ack()
{
    uint32_t config = Hwacc_data_ptr->config;
    config = config & 0xFFFF FEFF;
    Hwacc_data_ptr->config = config;
}
```

Access functions (Ready/Read)

- Ready/Read function for Status and Output data

```
bool Hwacc_isready()
{
   bool ready=false;
   int status = Hwacc_data_ptr->status;
   status = status & 0x0000 0001;
   if (status) ready=true;
   return ready;
}

int Hwacc_read_z()
{
    int z = HWacc_data_ptr->z;
    return z;
}
```

Simple Driver Function

- Driver function with ready status polling

```
volatile Hwacc_data* Hwacc_data_ptr = (Hwacc_data*) HWACC_ADDRESS;
int Hwacc_power(int x, int y)
{
    Hwacc_write_x(x);
    Hwacc_write_y(y);
    Hwacc_set_start();
    Hwacc_unset_start();
    int z;
    bool ready = Hwacc_isready();
    // Polling
    while (! ready)
    {
          ready = Hwacc_isready();
    }
    int z = Hwacc_ready_z(x);
    Hwacc_set_ack();
    Hwacc_unset_ack();
    return z;
}
```

```
int power(int x, int y)
{
    int z=x;
    for (int i=1;i<y;i++)
    {
          z=z*x;
    }
    return y;
}
```

Simple Driver Function

- Driver function with interrupt service routine (ISR)

```
volatile Hwacc_data* Hwacc_data_ptr = (Hwacc_data*) HWACC_ADDRESS;
int z;
bool z_valid;
void hwacc_power_start(int x, int y)
{
    Hwacc_write_x(x);
    Hwacc_write_y(y);
    Hwacc_set_start();
    Hwacc_unset_start();
    z_valid = false;
}

void Hwacc_ISR() {
    z_valid = Hwacc_isready();
    if (z_valid) {
          z = Hwacc_ready_z(x);
          Hwacc_set_ack();
          Hwacc_unset_ack();
    }
}
```

```
int power(int x, int y)
{
    int z=x;
    for (int i=1;i<y;i++)
    {
          z=z*x;
    }
    return y;
}
```

Polling vs. Interrupts

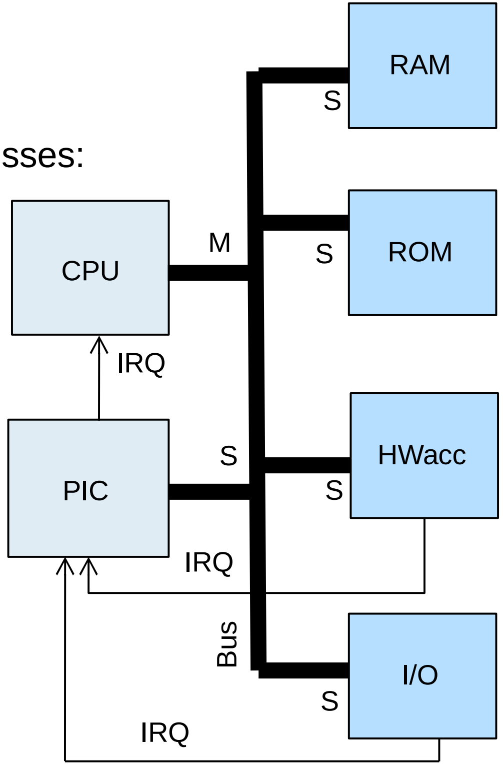
- If the CPU wants to know the status of an peripheral, it needs to access a status register, e.g., it can poll a status register in fixed time periods to see if a computation finished.

- Alternatively the peripheral can issue an interrupt to the CPU to signal a certain event, e.g., that it finished a certain computation
- ARM has two types of interrupts:
  - Standard: IRQ
  - Fast: FIQ

Interrupt Lines & Interrupt Controller (PIC)

Example:

CPU has 32-bit addresses:
0x00 00 00 00
…
0xFF FF FF FF



RAM (1MB) mapped to:
0x010 00 00 00
…
0x10 0F FF FF

ROM (262k) mapped to:
0x00 00 00 00
…
0x00 03 FF FF

HWacc (4 words) mapped to:
0x50 00 00 00
…
0x50 00 00 10

I/O mapped to:
0x60 00 00 B0
…
0x60 00 00 B8

Recap

Recap

- How does the assembly code look like?
- How does assembly code look like for the RISC-V processor?
- How is this code executed on an embedded RISC-V processor?
    - Single-cycle processor
    - Pipelined processor
- How can we estimate the execution time?

- How do on-chip buses work?
- How does the HW/SW interface look like?