

TE2003B

SoC Design: Computer organisation & architecture
The processor

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
February – June 2021



References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

J. Yiu, *The definitive guide to ARM Cortex-M0 and Cortex-M0+ processors*, Second edition, Elsevier, 2015.

A basic Microprocessor (μP)

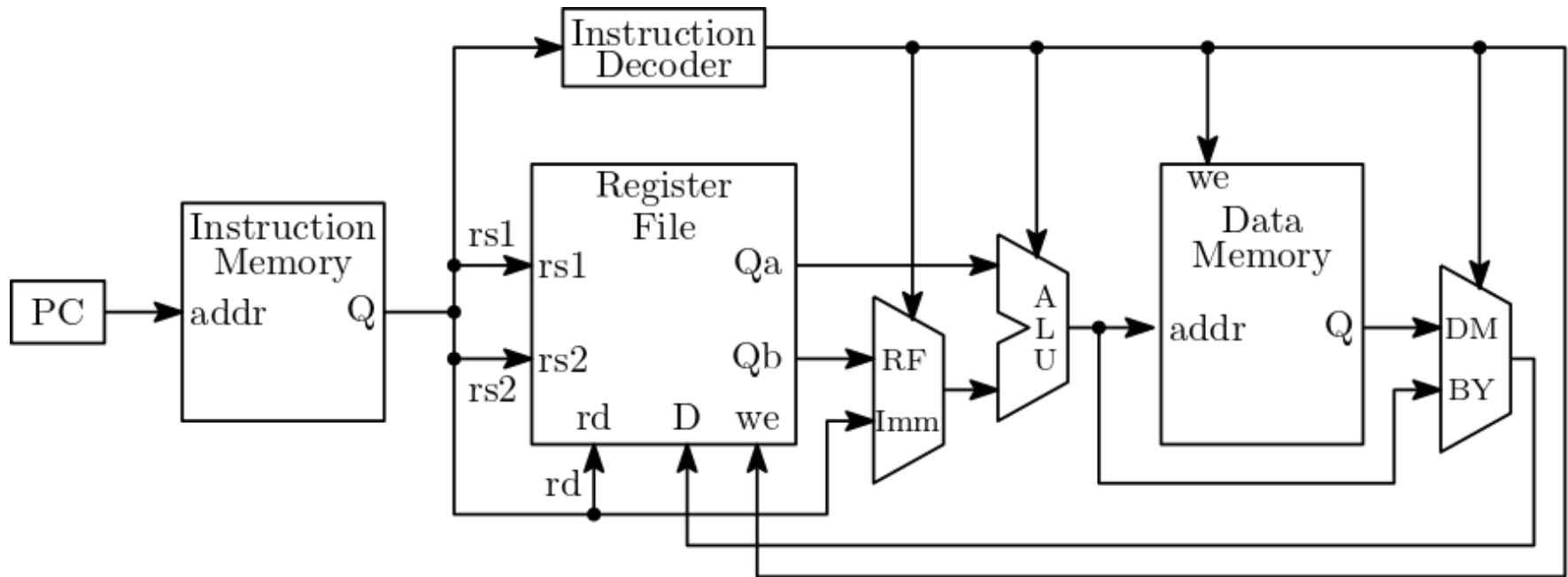


Figure 1: A generic μP block diagram

Instruction cycle

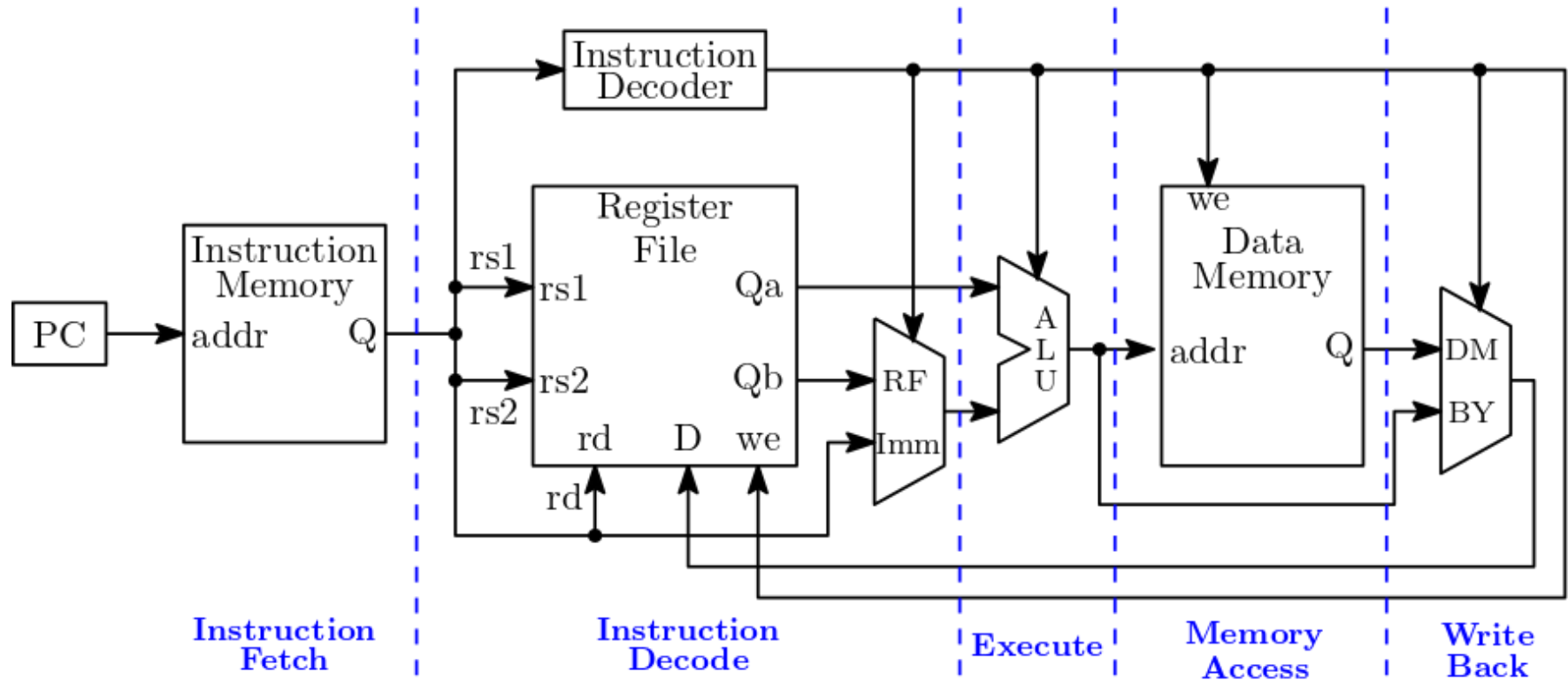


Figure 2: Instruction cycle

Stages of a basic μ P instruction cycle

1. **Instruction Fetch (IF).** Instructions are read from Instruction Memory (IM).
2. **Instruction Decode (ID).** Type of operation and operands are defined.
3. **Execute (EXE).** Operands are used in order to perform arithmetic or logical operations.
4. **Memory Access (MEM).** Data is read/written from/to Data Memory (DM).
5. **Write Back (WB).** Results from EXE or MEM stages are written back into Register File (RF).

Design conventions

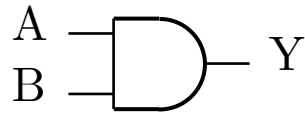
Design conventions

- μ P components might be divided into two categories.
- **Datapath.** Elements that process or manipulate data and addresses.
 - Multiplexers (MUX).
 - Arithmetic and Logic Unit (ALU).
 - Registers.
 - Memories.
- **Control.** Signals and logic elements that direct and dictate how datapath elements should respond and operate.
 - Enable.
 - MUX selectors.
 - ALU op.

Combinational elements

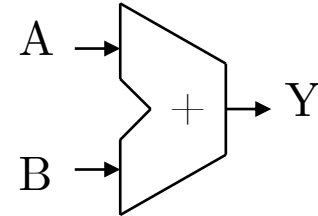
AND

$$Y = A \& B$$



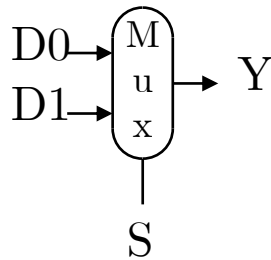
Adder

$$Y = A + B$$



MUX

$$Y = S ? D1 : D0$$



ALU

$$Y = F(A, B)$$

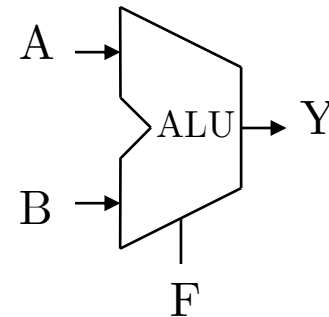


Figure 3: Examples of combinational elements

Sequential elements

- Register: stores data in a circuit
 - Uses a Clock (Clk) signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1

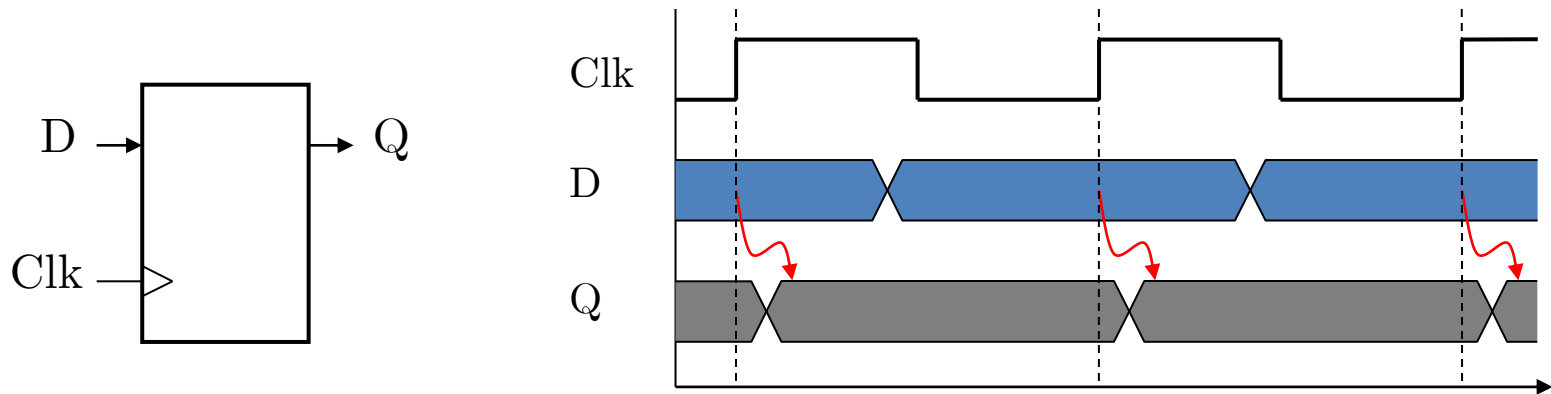


Figure 4: Clocking methodology

Sequential Elements

- Register with Write Enable (WE)
 - Updates on Clk edge if and only if write enable input is 1.
 - Used when stored value is required later.

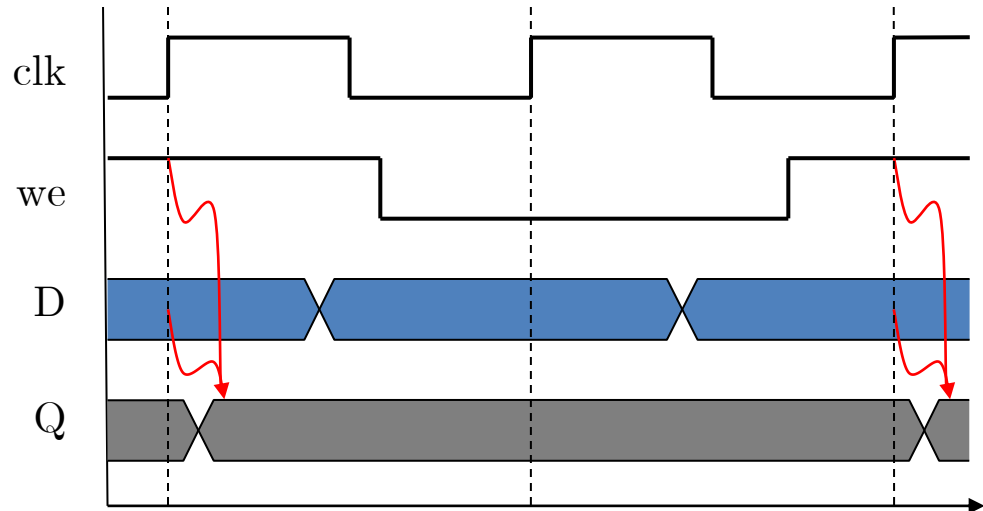
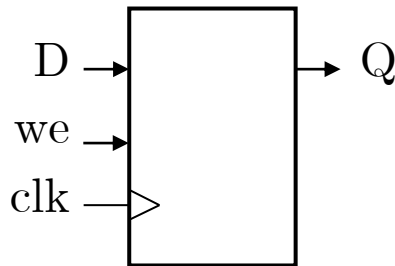


Figure 5: A register with WE

ARMv6-M design

ARM architecture

- Advanced RISC Machines (ARM).
- ARM does not fabricate Integrated Circuits (ICs).
- ARM licenses (sells) μ P cores in the form of Intellectual Property (IP).
- ARM architecture has several variants, which are identified as ARMvX.
 - For example: Freescale's KL25Z μ C (Microcontroller) is based on ARM's Cortex-M0+ (read as M zero plus), which is based on ARMv6-M ISA.

ARM core families

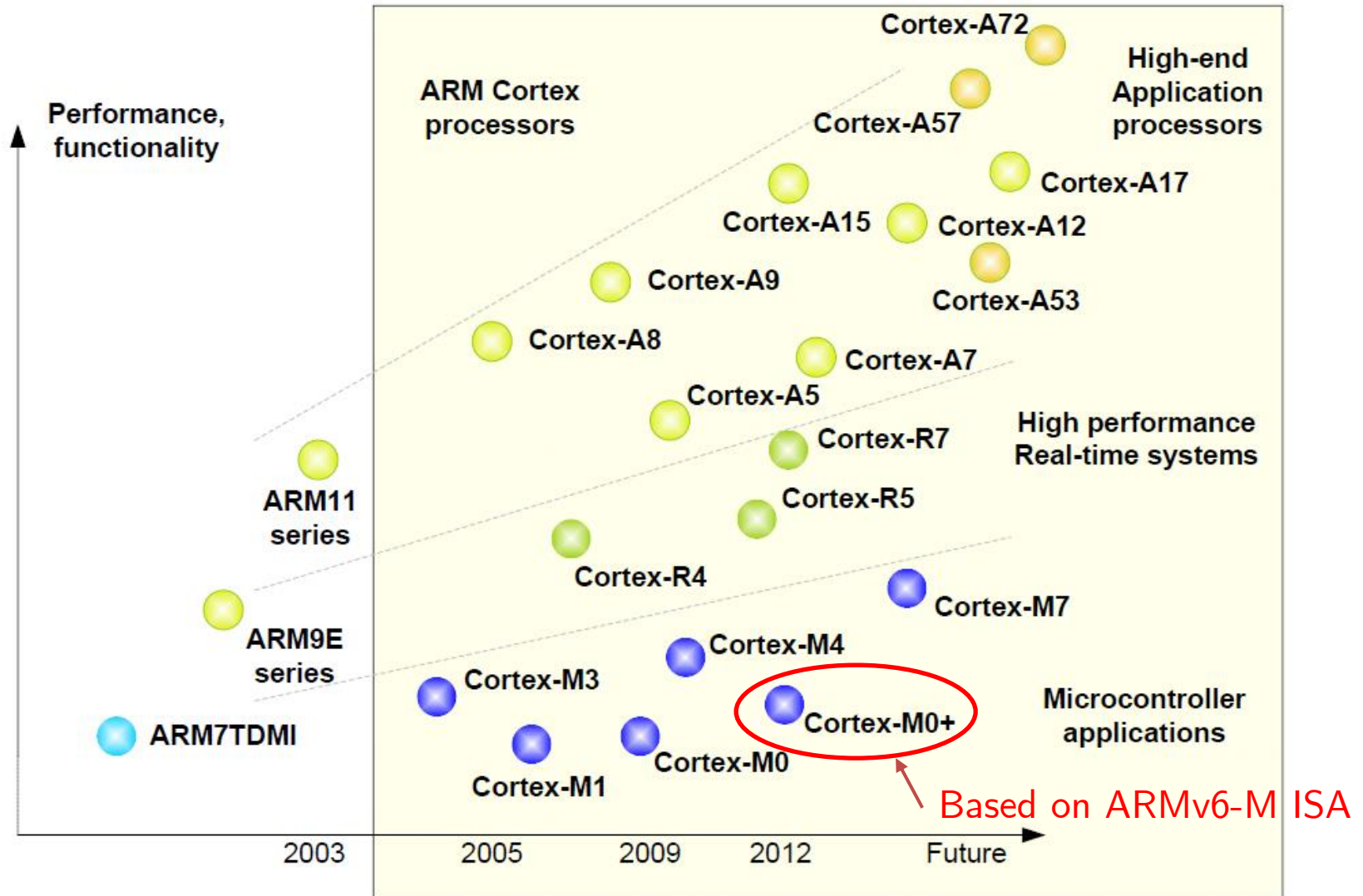


Figure 6: ARM core families

Cortex-M0+ main characteristics

- 32-bit RISC processor
- 16 32-bit registers
 - Some of these registers have a special purpose
- 56 instructions
 - 50 16-bit instructions
 - 6 32-bit instructions
- Stack Pointer (SP)
- Designed for low-power applications

Cortex-M0+ memory map

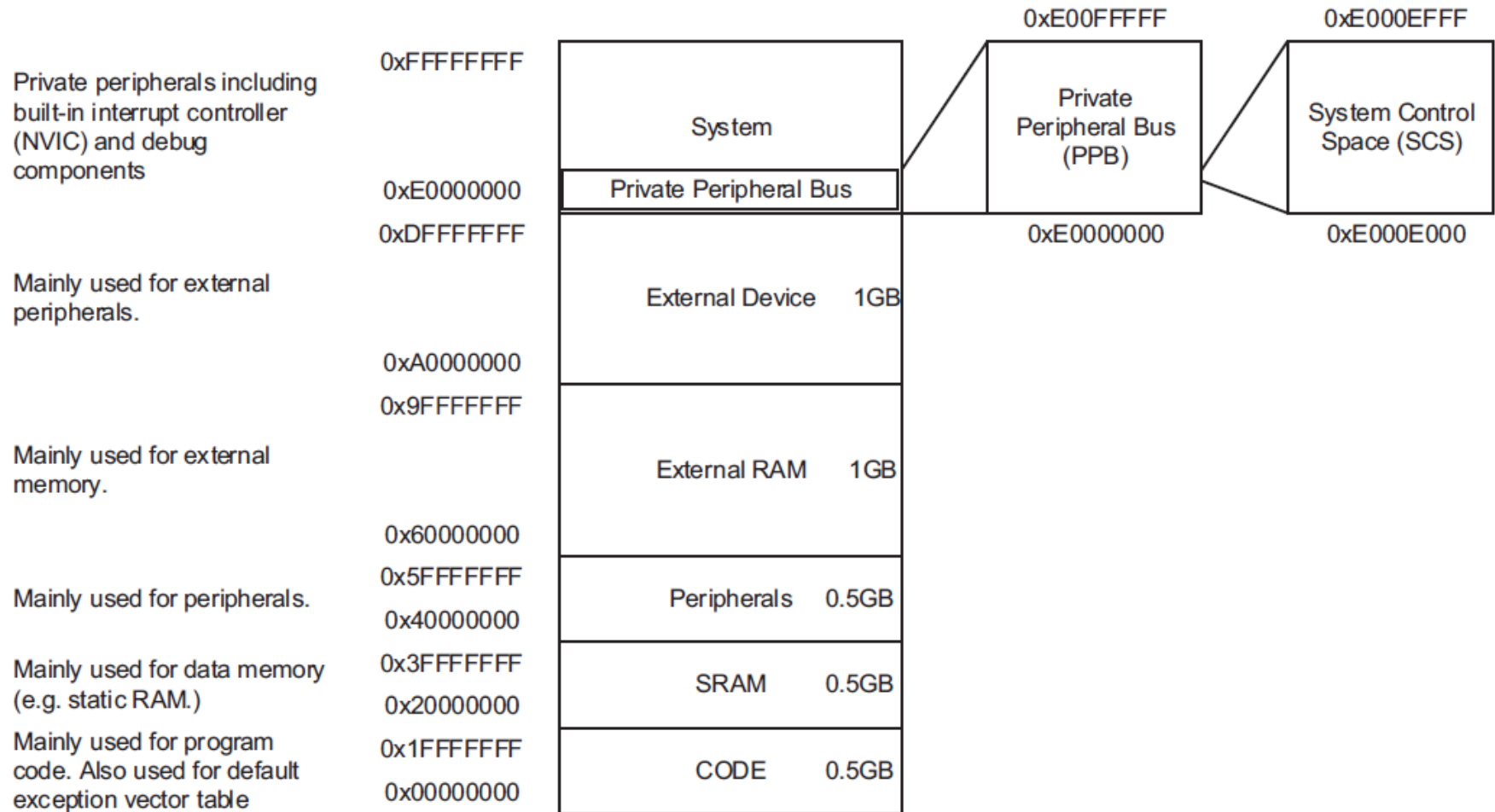


Figure 7: Cortex-M0+ memory map

Cortex-M0+ registers

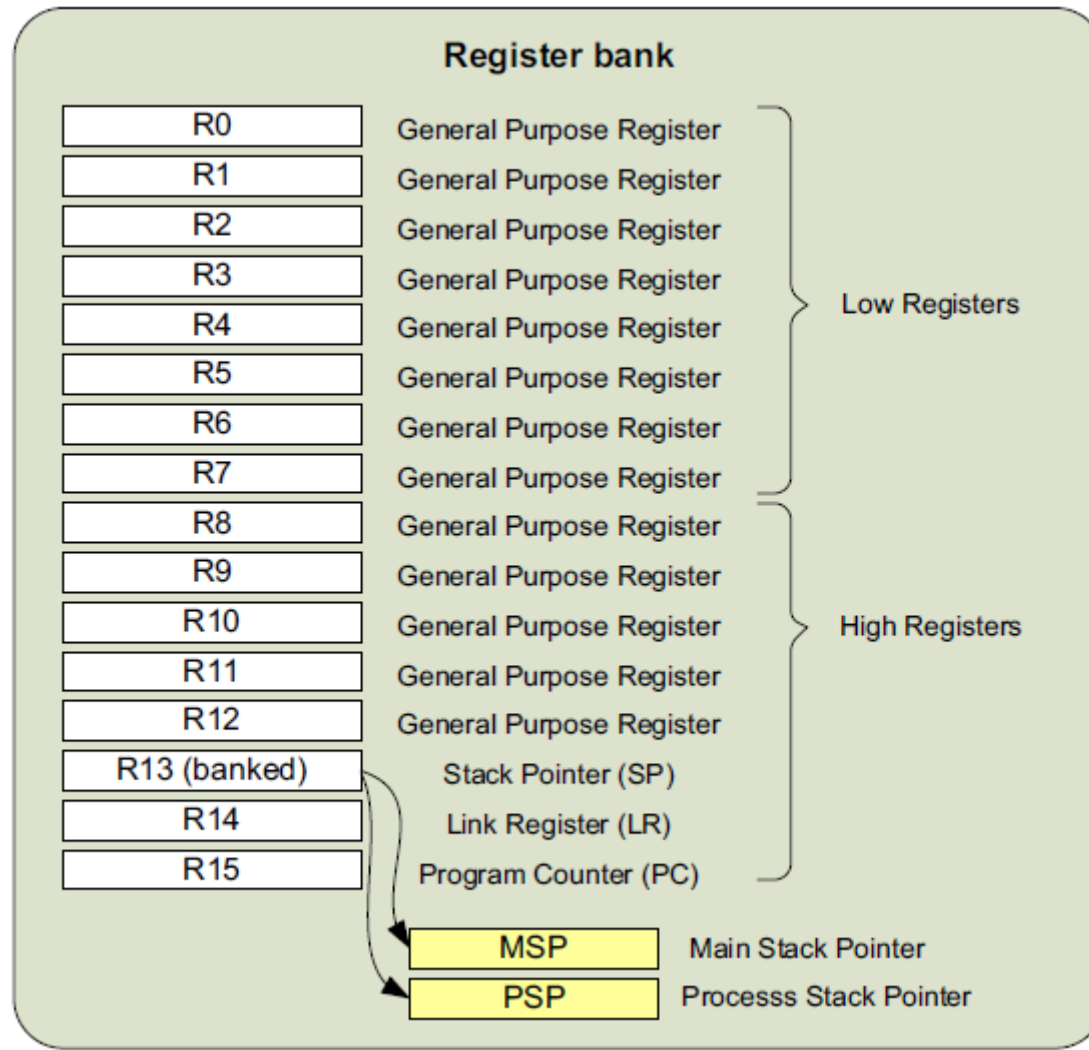


Figure 8: Cortex-M0+ registers

Cortex-M0+ instructions



Figure 9: Cortex-M0+ instructions

ARMv6-M design

- In order to simplify our discussions, we'll focus on ARMv6-M (Cortex-M0+) ISA.
 - KL25Z μ C used in the microcontroller module of this course.
- More specifically, we'll focus only on a **subset** of ARMv6-M ISA in order to exemplify the fundamentals of μ P design that apply to any ISA.
- Instructions are encoded in either Thumb or Thumb-2 format.
 - Thumb: 16 bits.
 - Thumb-2: 32 bits.

ARMv6-M Thumb encoding

- ARMv6-M Thumb encoding

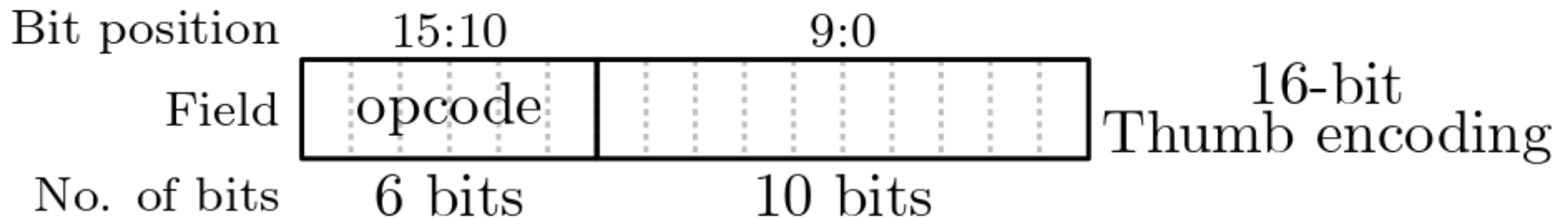


Figure 10: ARMv6-M Thumb encoding

- Bits 15 to 10 specify instruction type.
 - Each instruction type might comprise further instruction subtypes.
- Bits 9 to 0 are used for either operands or instruction subtypes.

ARMv6-M instruction subset

- Arithmetic and logical instructions.
 - **ADDS**: Addition.
 - **SUBS**: Subtraction.
 - **ANDS**: Bitwise logic AND.
 - **ORS**: Bitwise logic OR.
- Data transfer instructions.
 - **LDR**: Load register.
 - **STR**: Store register.
- Conditional branch instruction.
 - **B<cond>**
- Unconditional branch instruction.
 - **B**

ARMv6-M design

- We'll follow an incremental approach.
 - We'll analyze the building blocks required for executing each of the selected ARMv6-M instructions.
 - We'll add those building blocks to previous iterations until we reach to our final design.
- We may begin by analyzing the building blocks required for reading each of the instructions.
 - Can you name some of the building blocks for this purpose?

Instruction fetch datapath

ARMv6-M instruction fetch datapath

- Instruction fetch datapath steps.
 1. Read instruction from IM.
 2. Increment PC on clock edge.
- What are the required building blocks for performing the above steps?

ARMv6-M IF datapath

- IF datapath
 - Program Counter (PC) points to the address in which current instruction is stored.
 - PC is the read address of IM.
 - PC updates its value every clock cycle.

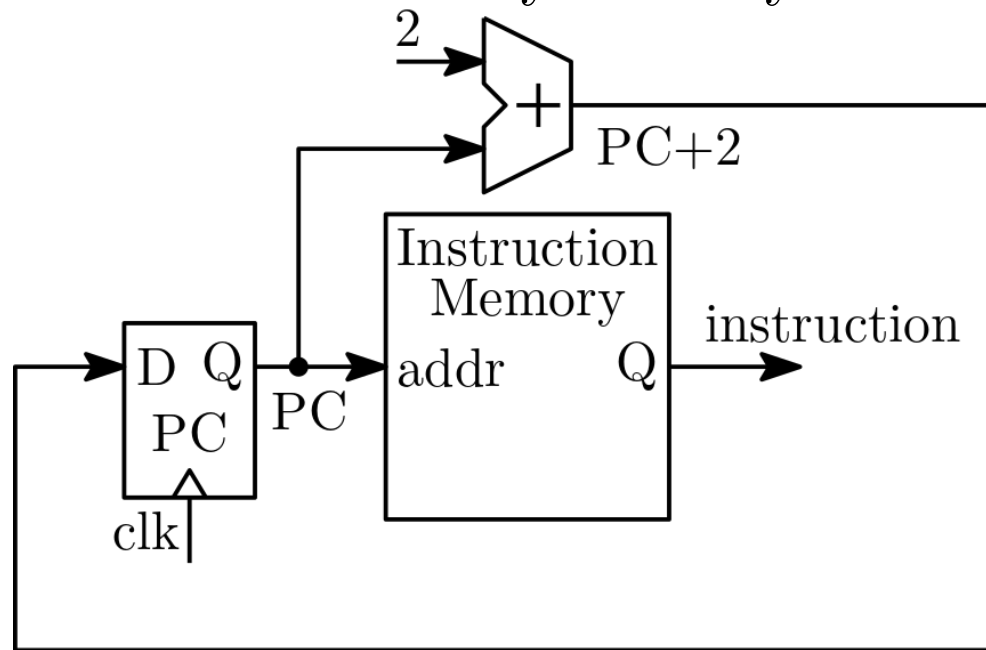


Figure 11: Basic IF datapath

ARMv6-M addressable memory

- Each data byte has a unique address.
- Word address = 32 bits or 4 bytes.
- Addresses increment by 4.

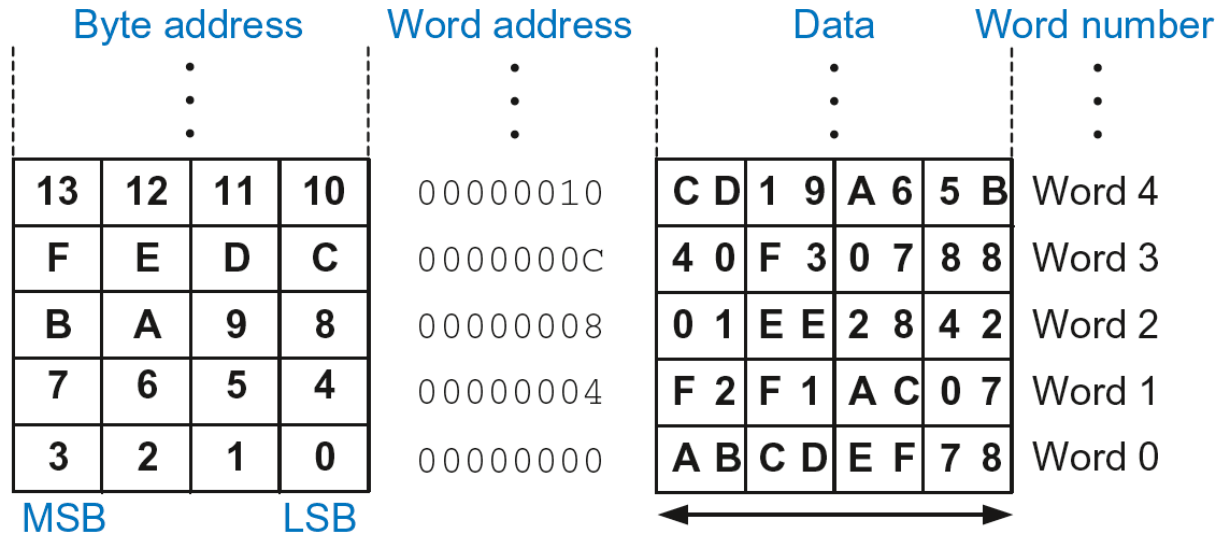


Figure 12: ARMv6-M addressable memory

ARMv6-M memory alignment

- Address alignment affects data accesses and updates to the PC.
- All instruction fetches are halfword-aligned
 - Byte = 8 bits
 - Halfword = 16 bits (2 bytes)
 - Word = 32 bits (4 bytes)
 - Double words = 64 bits (8 bytes)

ARMv6-M memory alignment

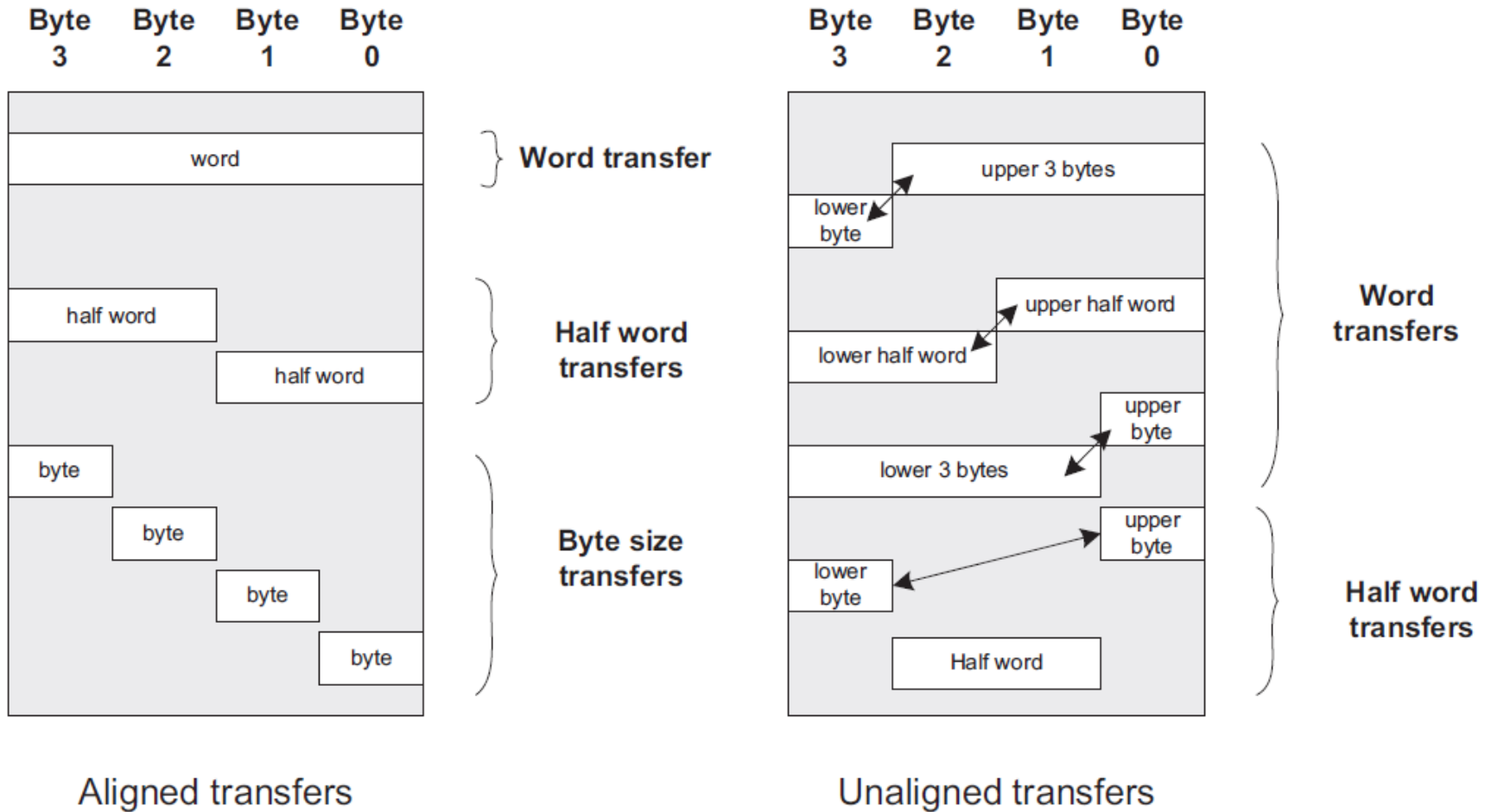


Figure 13: Example of aligned and unaligned transfers

Little-endian & big-endian

- Endianness determines order in which bytes are organized in memory.
- Big-endian
 - Most significant bytes are stored in lower bit positions.
- Little-endian
 - Most significant bytes are stored in higher bit positions.
- Cortex-M0+ uses big endian mode.

Little endian & big endian

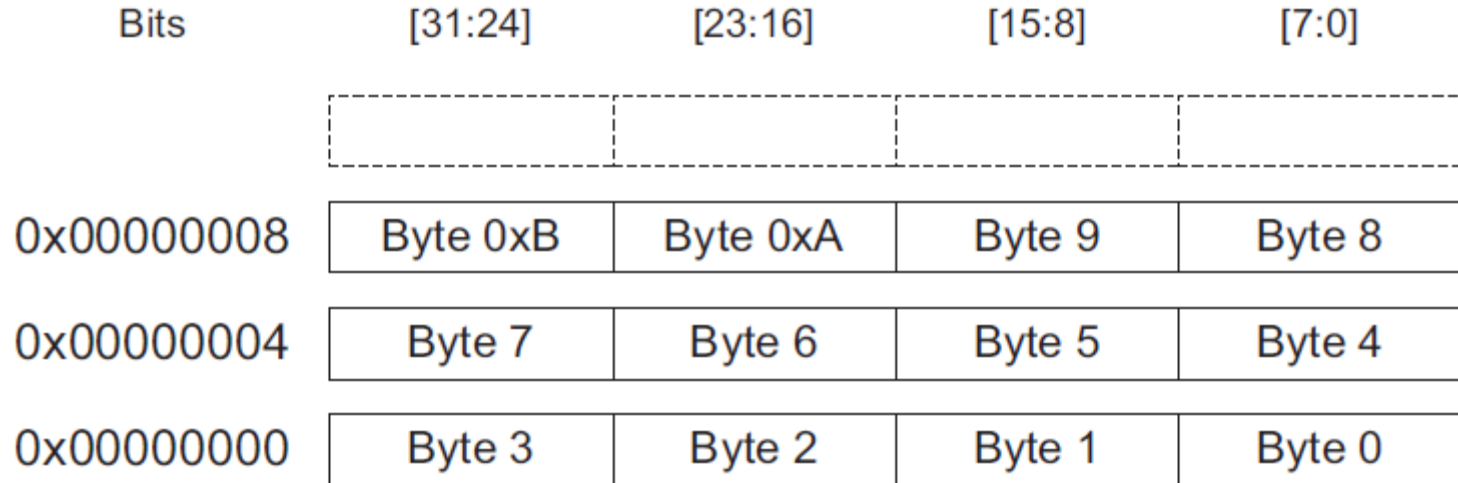


Figure 11: Example of a little-endian memory

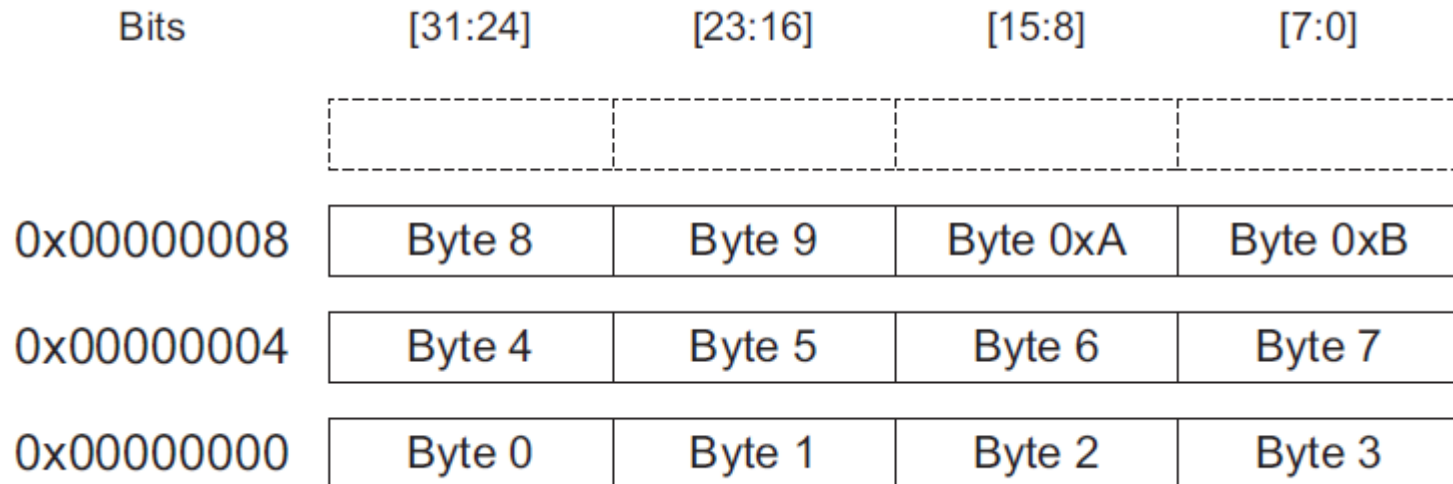


Figure 14: Example of a big-endian memory

Endianness & memory access

Address	Size	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x00000000	Word	Data[31:24]	Data[23:16]	Data[15:8]	Data[7:0]
0x00000000	Half word			Data[15:8]	Data[7:0]
0x00000002	Half word	Data[15:8]	Data[7:0]		
0x00000000	Byte				Data[7:0]
0x00000001	Byte			Data[7:0]	
0x00000002	Byte		Data[7:0]		
0x00000003	Byte	Data[7:0]			

Figure 15: Example of memory access in a little-endian memory

Endianness & memory access

Address	Size	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x00000000	Word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x00000000	Half word	Data[7:0]	Data[15:8]		
0x00000002	Half word			Data[7:0]	Data[15:8]
0x00000000	Byte	Data[7:0]			
0x00000001	Byte		Data[7:0]		
0x00000002	Byte			Data[7:0]	
0x00000003	Byte				Data[7:0]

Figure 16: Example of memory access in a big-endian memory

R-Type datapath

ARMv6-M R-Type design

- R-Type instructions.
 - Operand(s) are retrieved (read) from any of the 8 lower general-purpose registers.
 - Results are stored (written) back to any of the 8 lower general-purpose registers.
- Which building blocks are required?
 - Let's first analyze R-Type instructions.

ARMv6-M R-Type design

- ARMv6-M R-Type instructions.

Name	Syntax	Meaning
Addition	ADDS <i>rd</i> , <i>rn</i> , <i>rm</i>	$rd = rn + rm$
Subtraction	SUBS <i>rd</i> , <i>rn</i> , <i>rm</i>	$rd = rn - rm$
Bitwise AND	ANDS <i>rd</i> , <i>rn</i> , <i>rm</i>	$rd = rn \& rm$
Bitwise OR	ORRS <i>rd</i> , <i>rn</i> , <i>rm</i>	$rd = rn rm$

Table 1: Selected ARMv6-M R-Type instructions

- R-Type datapath steps
 - Read two registers.
 - Perform ALU operation.
 - Write ALU result into a register.
- What are the required building blocks?

ARMv6-M design

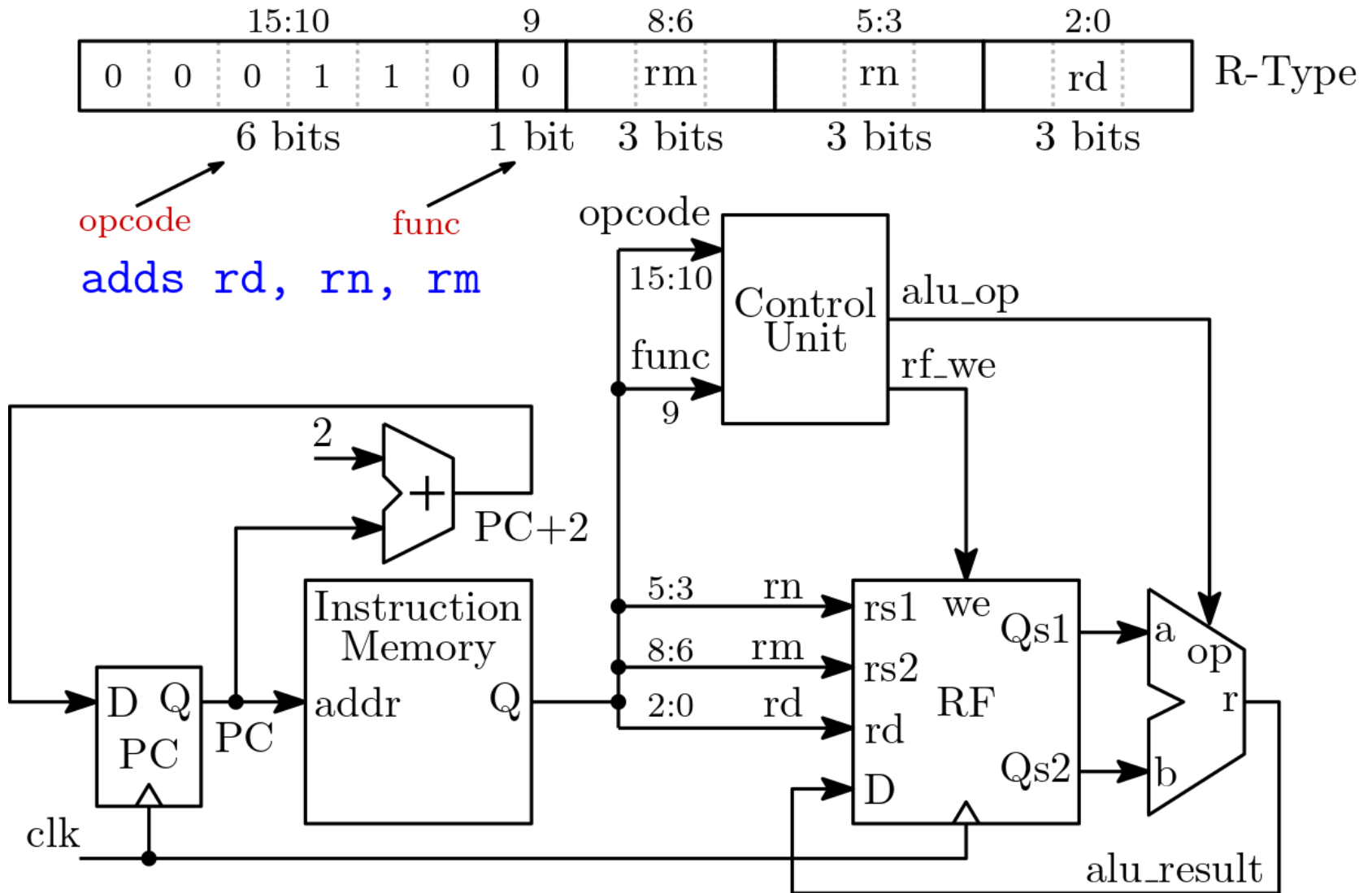


Figure 17: ARMv6-M `adds`/`subs` R-Type datapath

ARMv6-M design

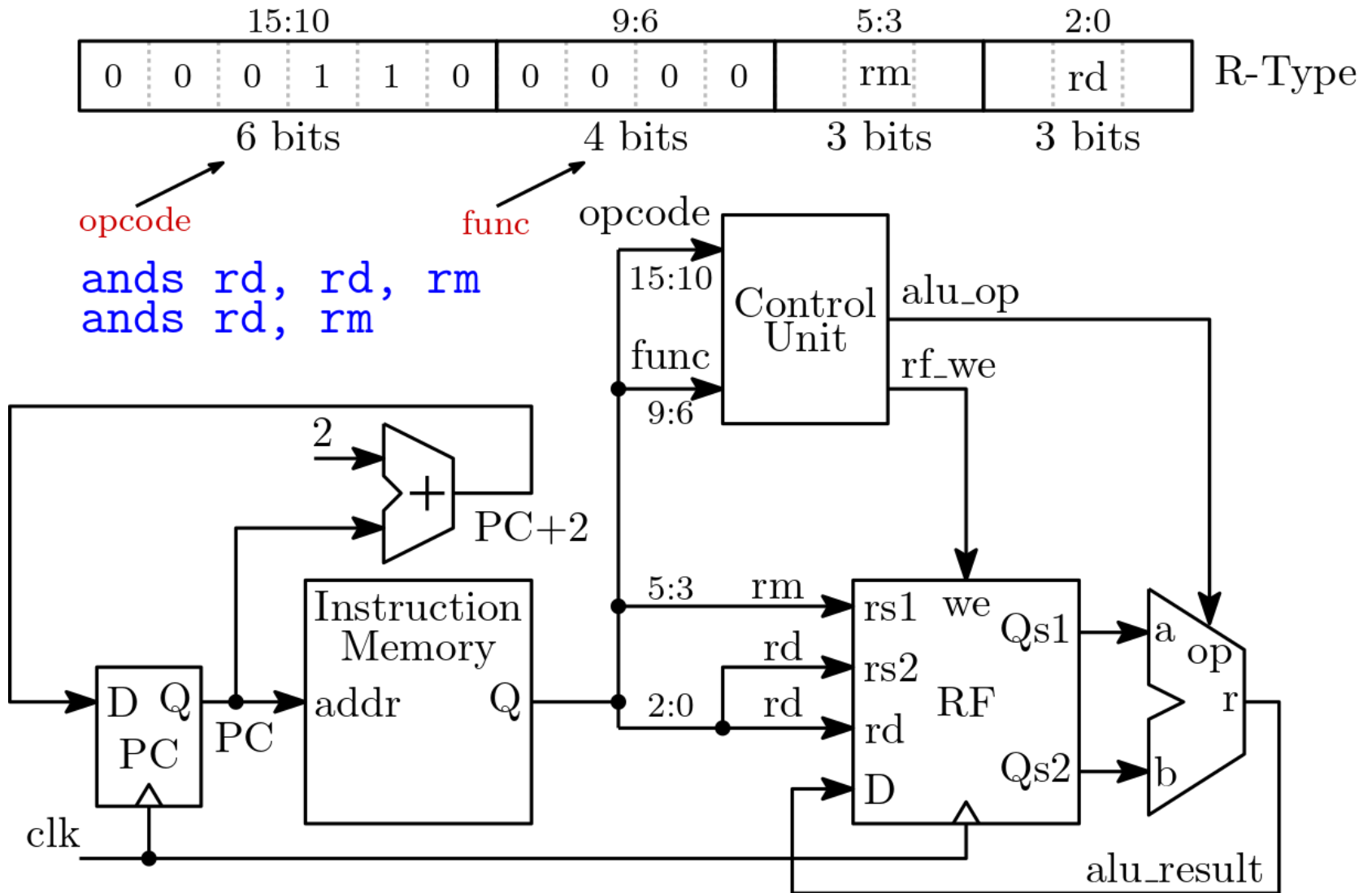


Figure 18: ARMv6-M ands/orrs R-Type datapath

ARMv6-M R-Type schematic

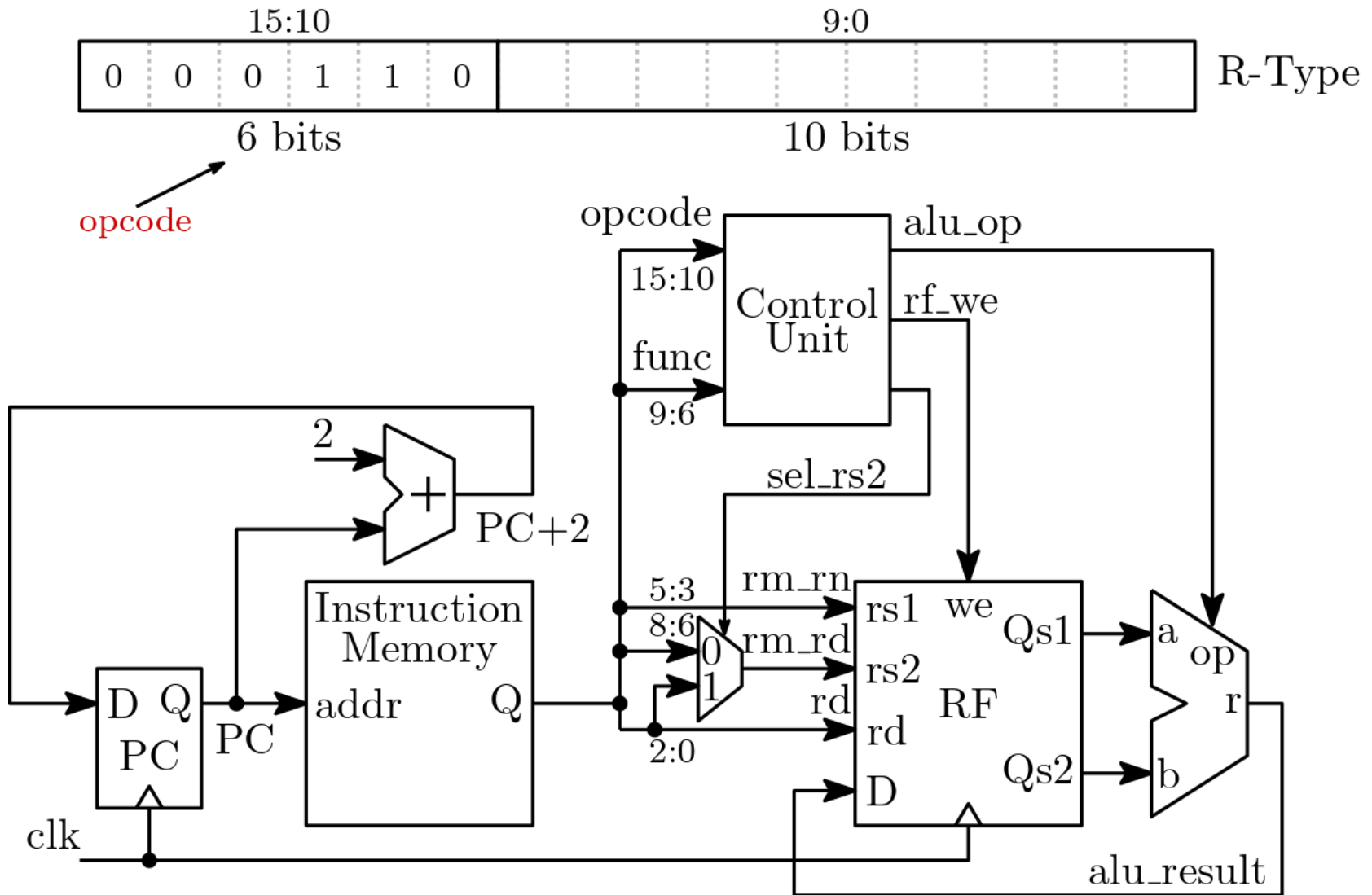


Figure 19: ARMv6-M complete R-Type schematic

Load/store datapath

ARMv6-M load/store instruction

- Load/store instructions read/write from/to Data Memory (DM).
 - Word-size transfer (4 bytes)
- Read/write address is calculated based on a register and an immediate.
 - $\text{address} = \text{reg} + \text{immediate}$

Name	Syntax	Meaning
Load immediate	LDR <i>rt</i> , <i>rn</i> , #imm5	$\text{rt} = \text{mem}[\text{rn} + \# \text{imm5}]$
Store immediate	STR <i>rt</i> , <i>rn</i> , #imm5	$\text{mem}[\text{rn} + \# \text{imm5}] = \text{rt}$

Table 2: Selected ARMv6-M load and store instructions

- Which building blocks are required for these instructions?

ARMv6-M load datapath

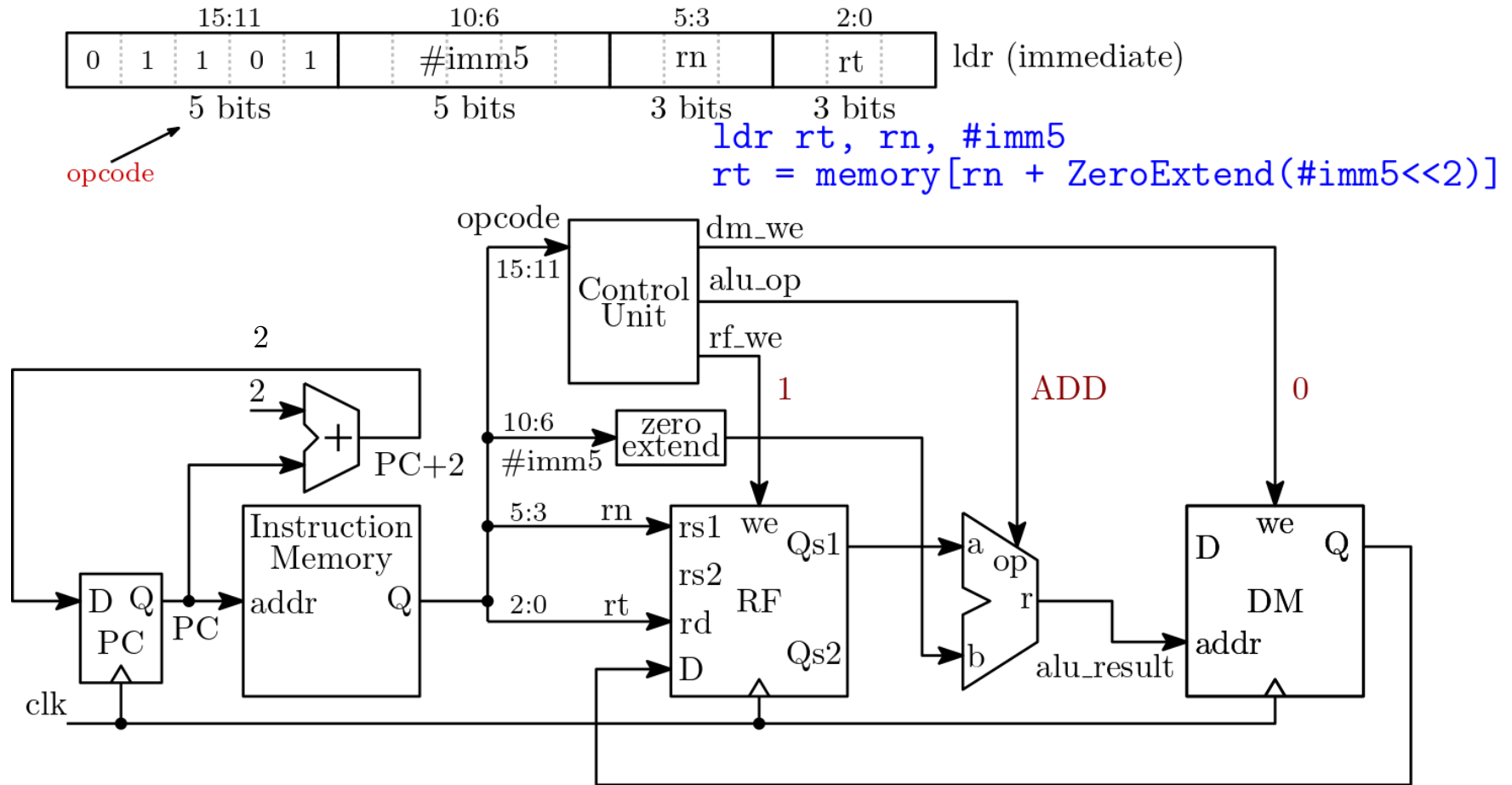


Figure 20: ARMv6-M load datapath

ARMv6-M store datapath

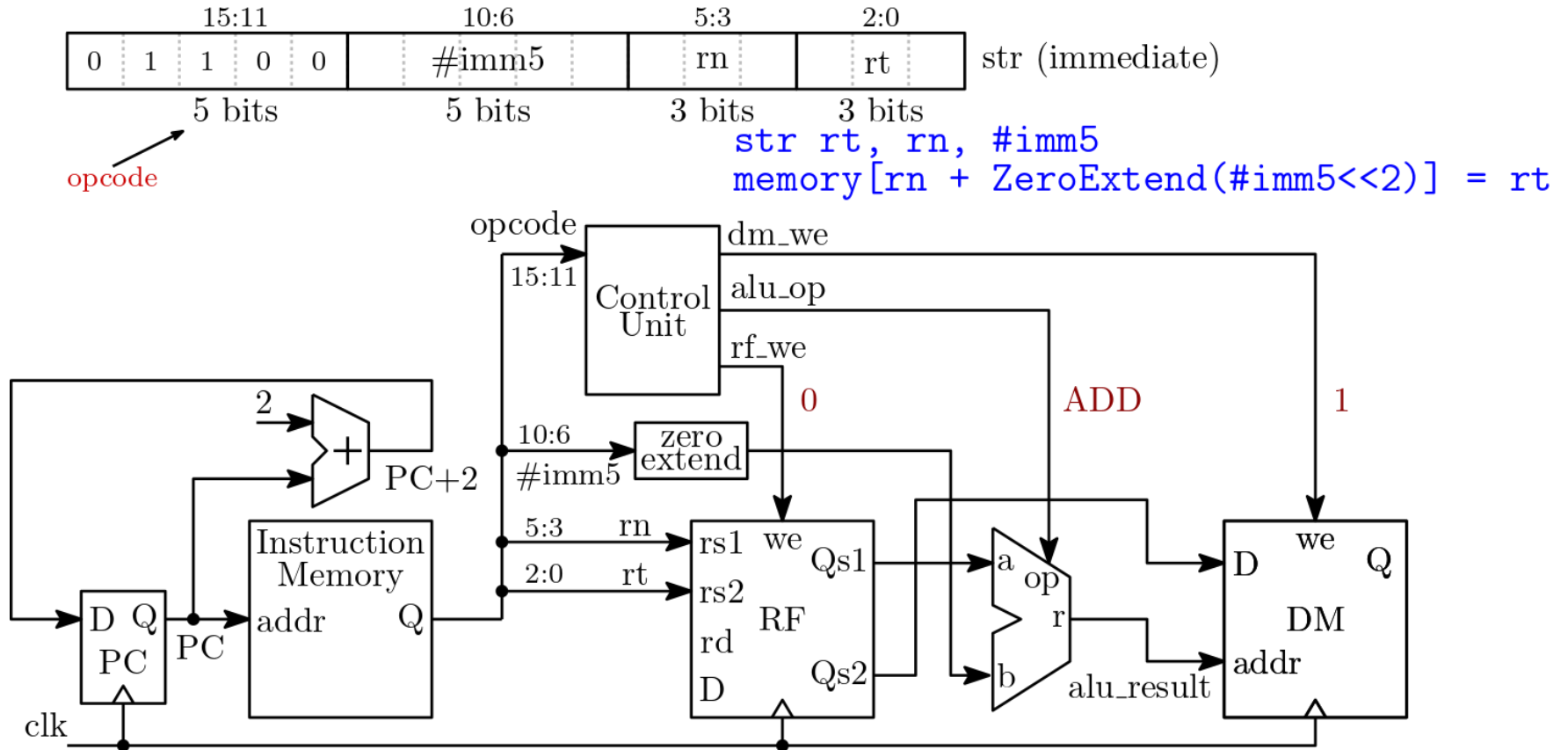


Figure 21: ARMv6-M store datapath

ARMv6-M load/store schematic

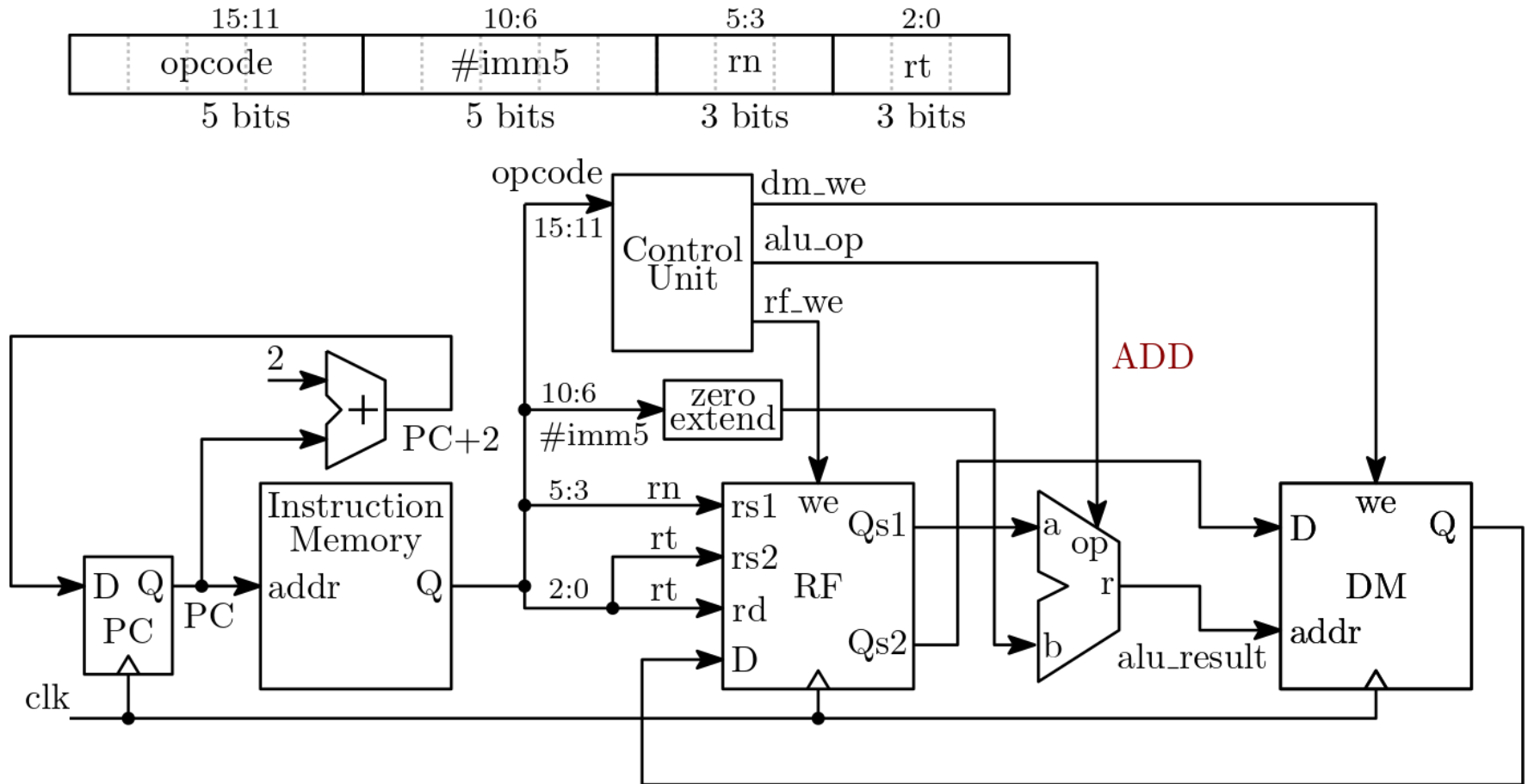


Figure 22: ARMv6-M load/store datapath

ARMv6 partial schematic

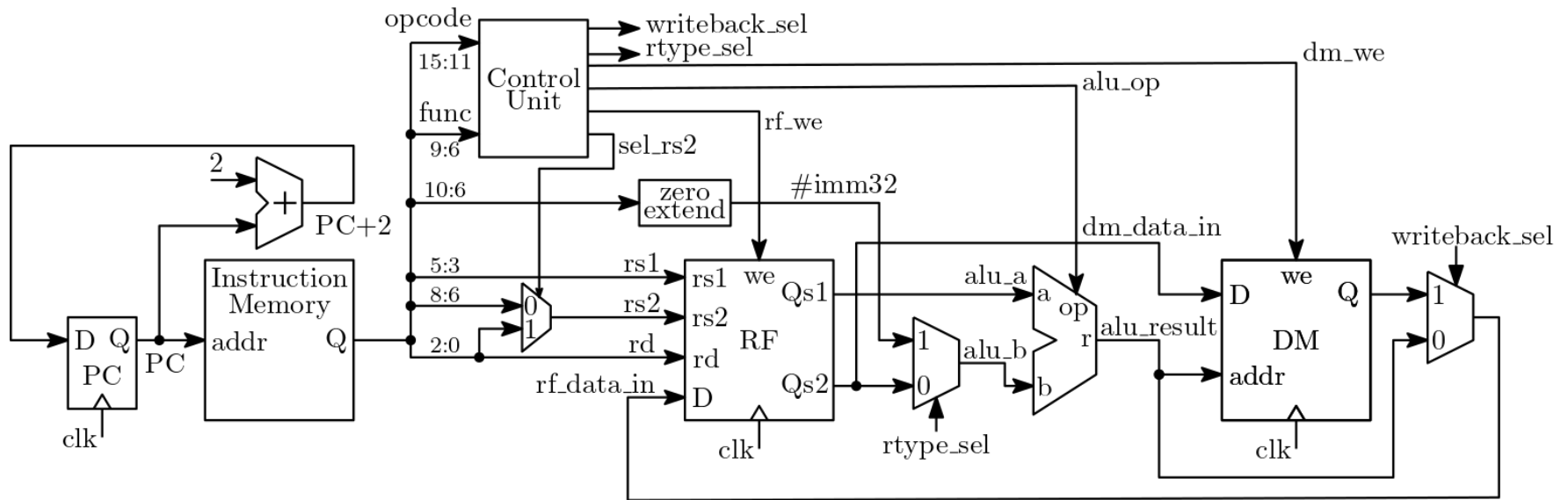


Figure 23: ARMv6-M partial schematic.
This includes R-Type and load/store instructions

Conditional branch datapath

ARMv6-M conditional branch datapath

- Branches are used to change the flow of the program.
- Conditional branches must verify whether a condition has been met.

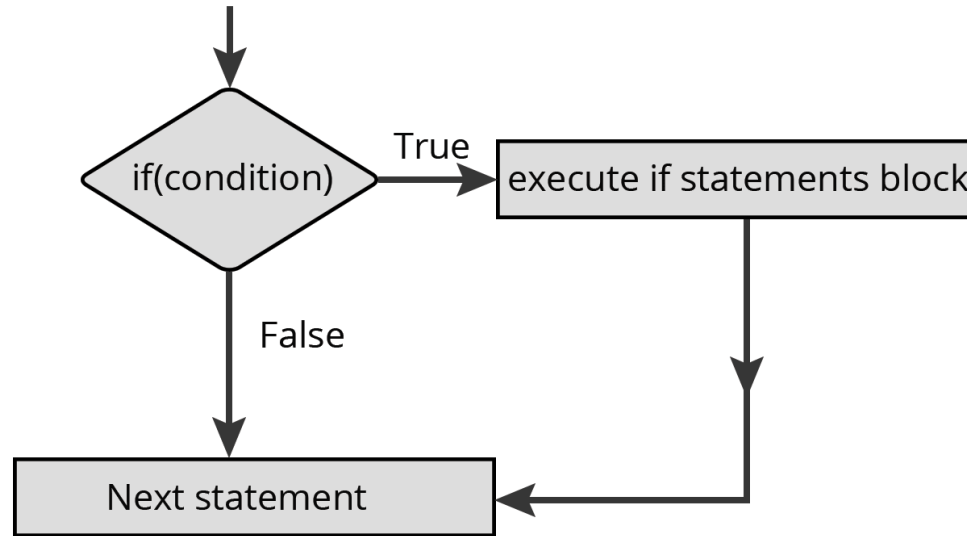


Figure 24: Flowchart for a simple if statement.

- Most common high-level language example is an **if** statement.

ARMv6-M conditional branch datapath

- Effectively, the processor must change the order in which instructions are performed.
 - How can this be achieved?
- PC points to the address in the IM of the instruction to be executed.
- If we modify the value of the PC, we may effectively change the flow of our program.

ARMv6-M conditional branch datapath

- We must update the value of PC.

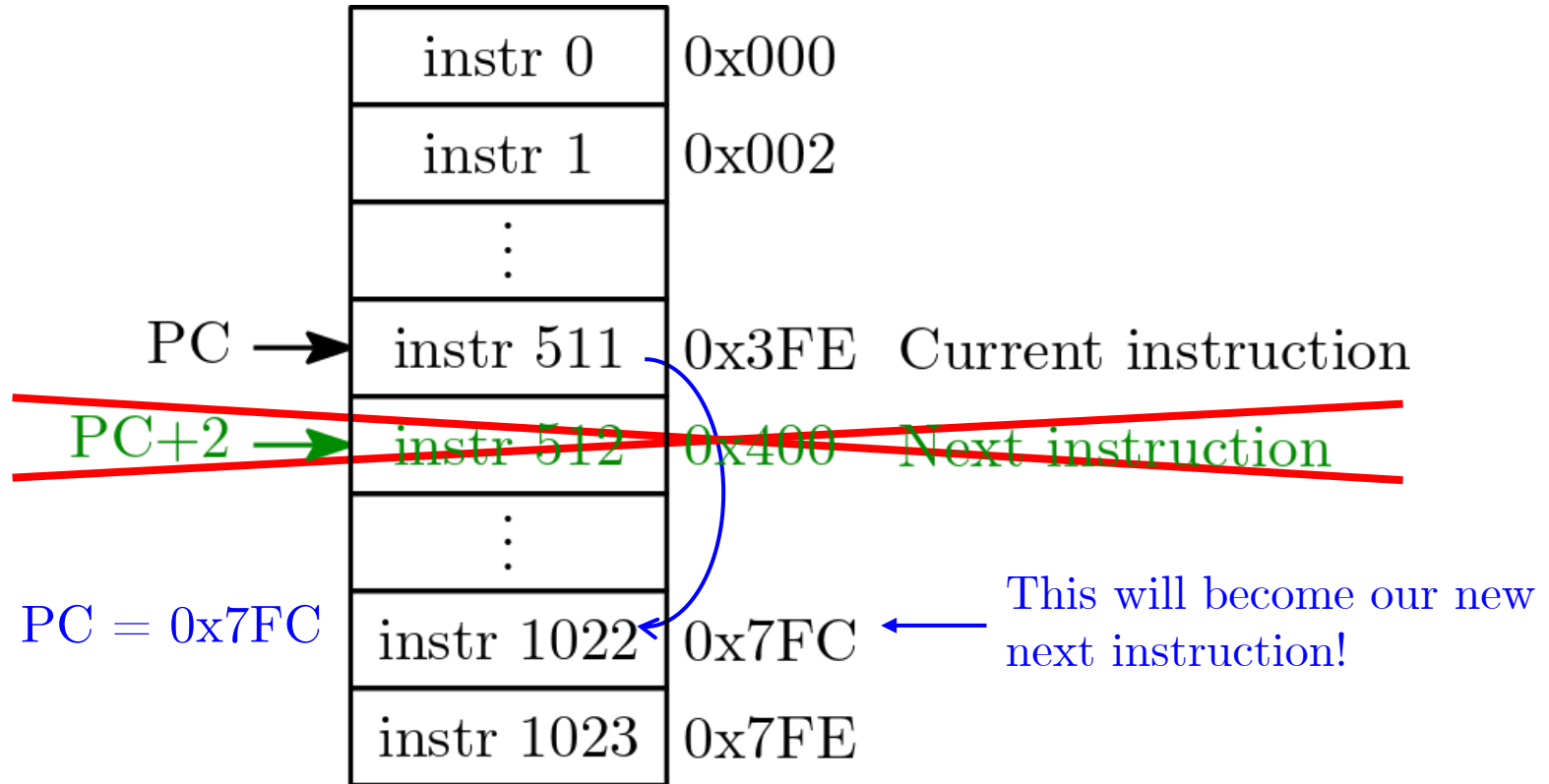
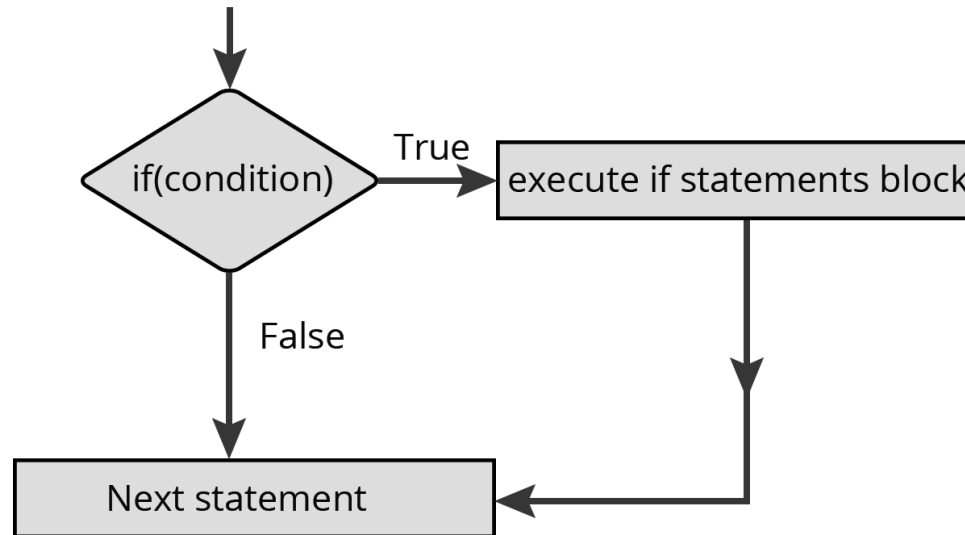


Figure 25: Changing the flow of the program by updating PC.

ARMv6-M conditional branch datapath

- Going back to the high-level language example, `if` statements verify whether a condition is satisfied.
- This means there are comparisons involved.



- Which tasks should the processor perform for conditional branches?

ARMv6-M conditional branch datapath

- In a conditional branch, the processor might
 - Verify if a condition is satisfied.
 - Compare two numbers ($==$, $!=$, \leq , \geq , $<$, $>$)
 - Check status of flags
 - Negative, carry, zero, overflow
 - Update PC with new address.
 - Usually known as target address.
 - Immediate value.

Name	Syntax	Meaning
Conditional branch	B<cond> label	If (cond == true) then PC = label

Table 3: ARMv6-M conditional branch instructions

- Which building block are required for this purpose?

ARMv6-M conditional branch datapath

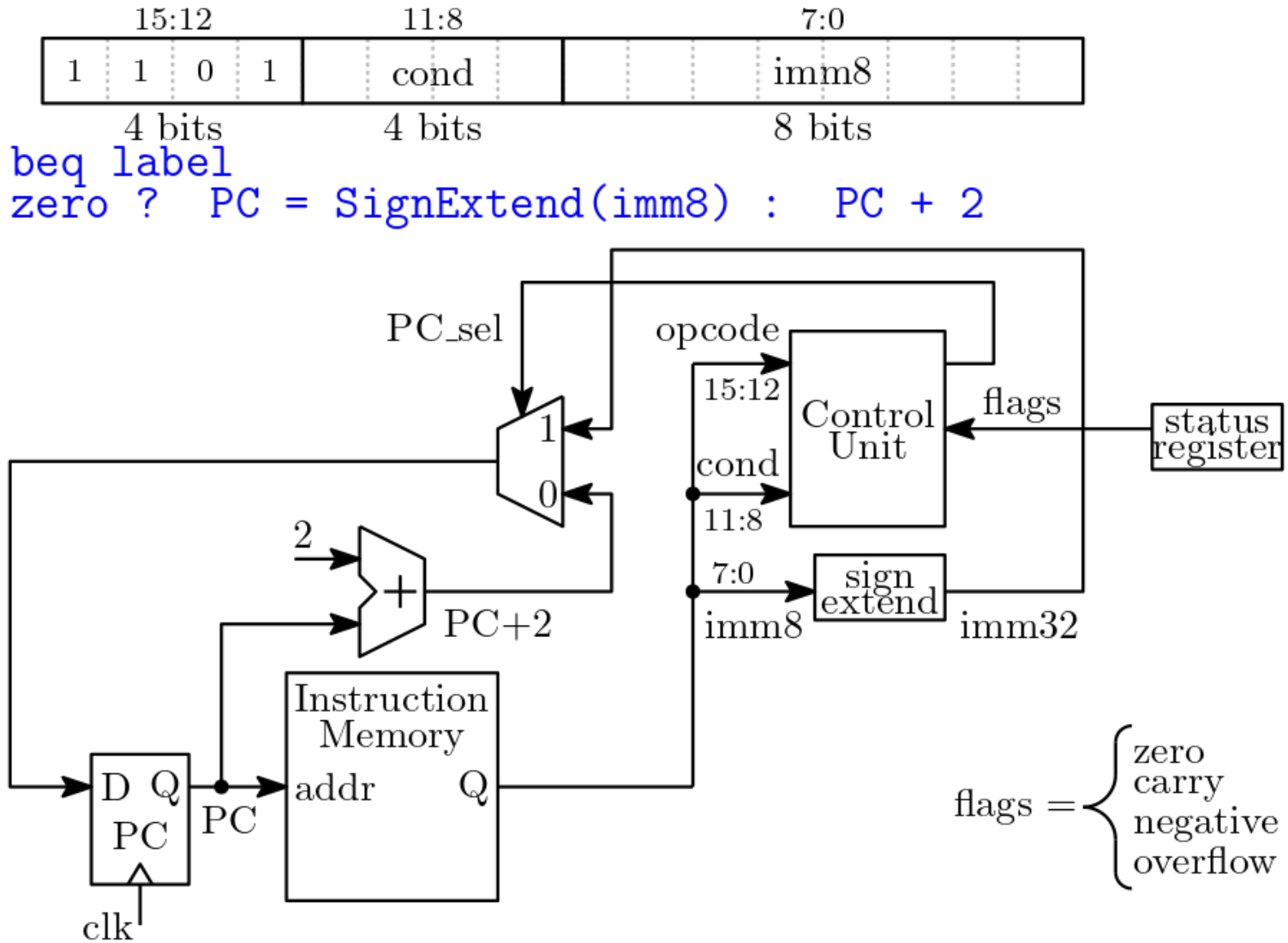


Figure 26: Conditional branch datapath.

ARMv6-M conditional branch

- ARMv6-M assembler instruction is B<cond>

cond	Mnemonic extension	Meaning	Condition flags
0000	EQ	Equal	$Z == 1$
0001	NE	Not equal	$Z == 0$
0010	CS ^a	Carry set	$C == 1$
0011	CC ^b	Carry clear	$C == 0$
0100	MI	Minus, negative	$N == 1$
0101	PL	Plus, positive or zero	$N == 0$
0110	VS	Overflow	$V == 1$
0111	VC	No overflow	$V == 0$
1000	HI	Unsigned higher	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	$N == V$
1011	LT	Signed less than	$N != V$
1100	GT	Signed greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	$Z == 1$ or $N != V$
1110 ^c	None (AL) ^d	Always (unconditional)	Any

In assembler language we would write this instruction as
BEQ
BNE
BCS
BCC
etc

Table 3: ARMv6-M condition codes

ARMv6-M conditional branch

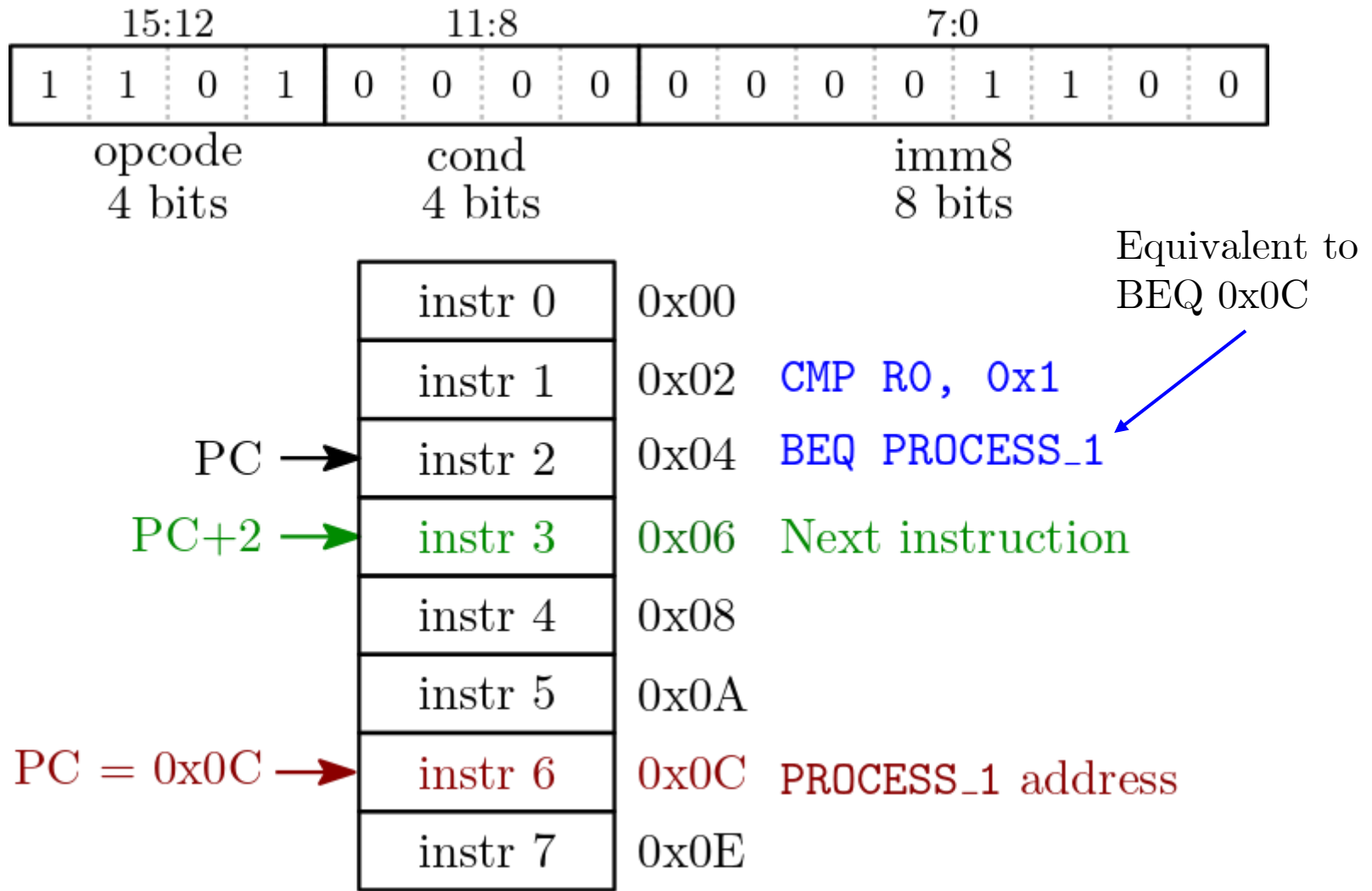


Figure 27: Conditional branch datapath.

ARMv6-M conditional branch

- B<cond> instruction evaluates a condition defined by the previous instruction.
- Its execution is based entirely on the Status Register (SR), which stores the flags zero, carry, negative, overflow.
- Example:
 - C pseudocode

```
if (a == 1) {  
    do_process1()  
}
```
 - BEQ only checks the status of the flags, but it does not compare values nor set flags.

ARMv6-M conditional branch

- C pseudocode

```
if (a == 1) {  
    do_process1()  
}
```

- This would be translated to ARMv6-M assembler

```
cmp r0, 0x1    ; r0 represents variable "a" in C code  
beq process_1  ; branch to process_1 if r0==0x1
```

- `cmp` instruction not only compares the immediate value 0x1 with the contents of r0, but also updates the condition flags based on the results and discards the result.

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');  
    APSR.N = result<31>;  
    APSR.Z = IsZeroBit(result);  
    APSR.C = carry;  
    APSR.V = overflow;
```

- For simplicity, we are not going to focus on the datapath for `cmp`.

ARMv6-M conditional branch

- SR in ARMv6-M is called Application Program Status Register (APSR).

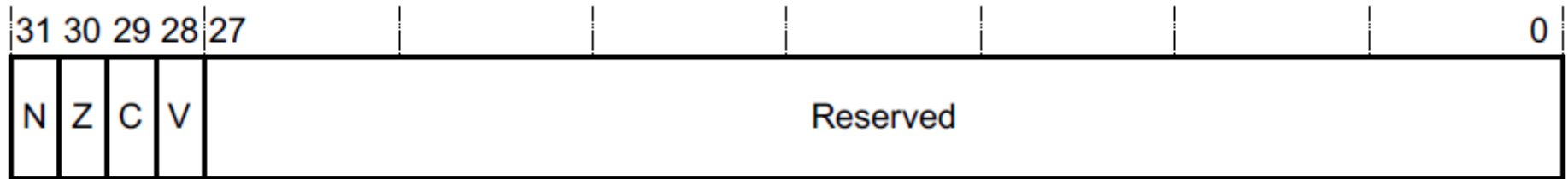


Figure 28: ARMv6-M APSR.

N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result is negative and set to 0 if it is positive or zero.

Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

Unconditional branch datapath

ARMv6-M unconditional branch datapath

- Unconditional branches simply update the contents of PC in order to modify the flow of the program.
- No need to test conditions.

Name	Syntax	Meaning
Unconditional branch	B label	PC = label

Table 4: ARMv6-M unconditional branch instructions

- Which building block are required for this purpose?

ARMv6-M conditional branch datapath

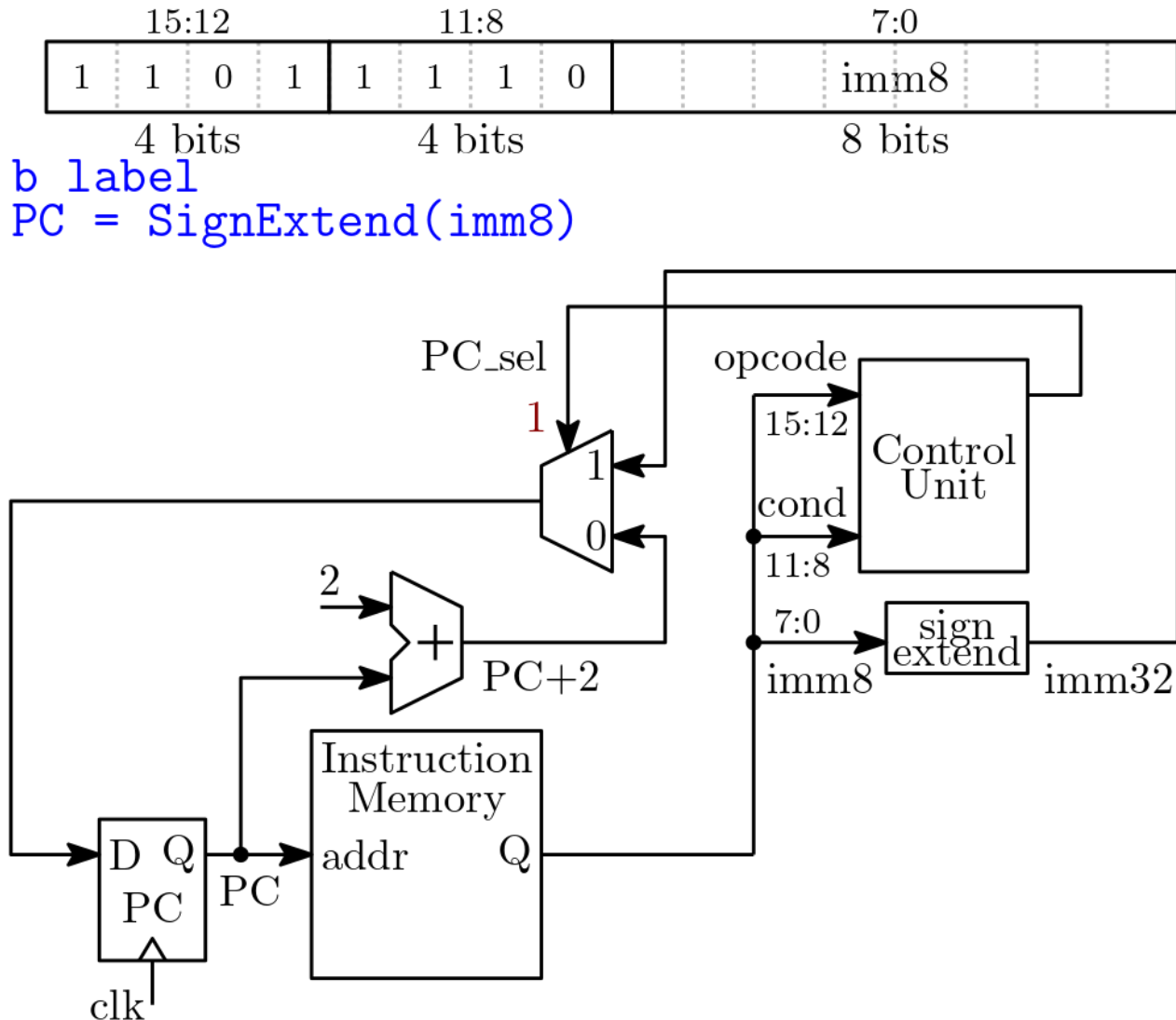


Figure 29: ARMv6-M unconditional branch datapath.

ARMv6-M complete datapath

- How can we create a schematic that includes all necessary building blocks, connections and control signals required for performing all selected 8 ARMv6-M instructions?
- Consider the incremental approach.
 - Take schematic in Figure 23 as the base design.
 - Include elements from Figure 26 and Figure 29.

Summary

- We demonstrated the basic methodology of a processor design.
 - This methodology might be applied to different areas, including design of embedded systems.
 - Divide and conquer approach.
 - Incremental design.
- We analysed some of the ARMv-6M ISA characteristics.
- Difference between ARMv-6M, Cortex-M0+, KL25Z microcontroller.