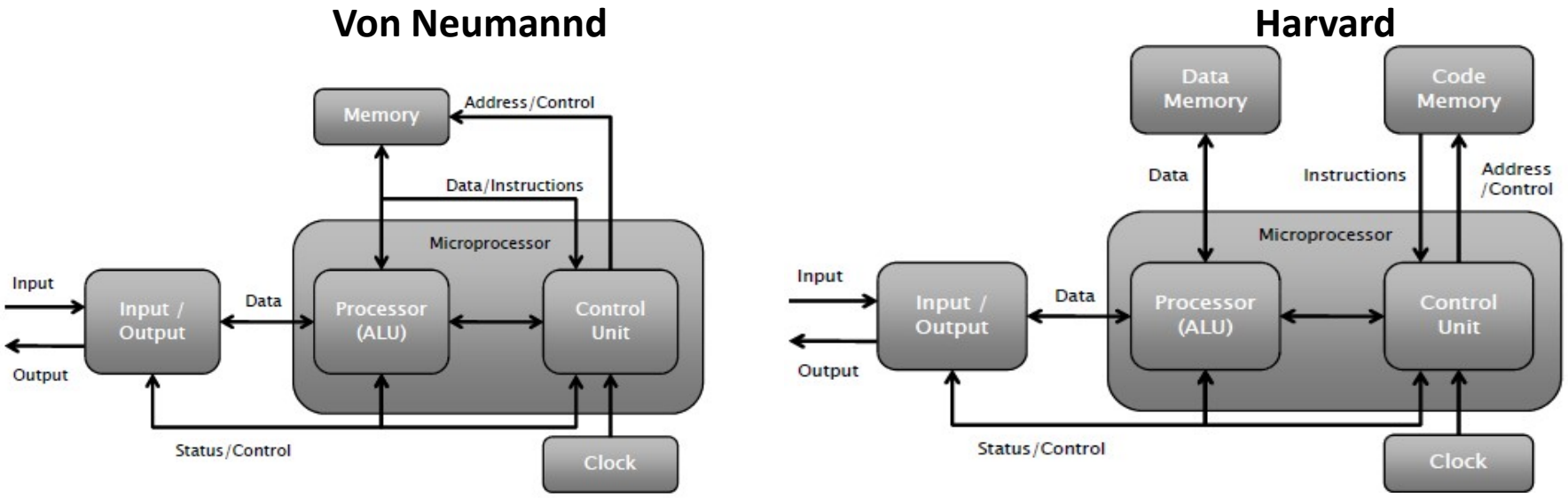
ARM Processors
ITMC301
ARM Microarchitecture - L3
Fall 2023

Dr. Abdussalam Baryun

ARM Computing Architecture

- Von Neumann architecture (old version used for ARM)
  - Single memory contains both the program code and the data.
- Harvard Architecture (most used for ARM)
  - Two separate memories. One contains only data while the other is containing only program code.



- Memory-mapped I/O:
  - No specific instructions for I/O (use Load/Store instr. instead)
  - Peripheral's registers at some memory addresses

SoC

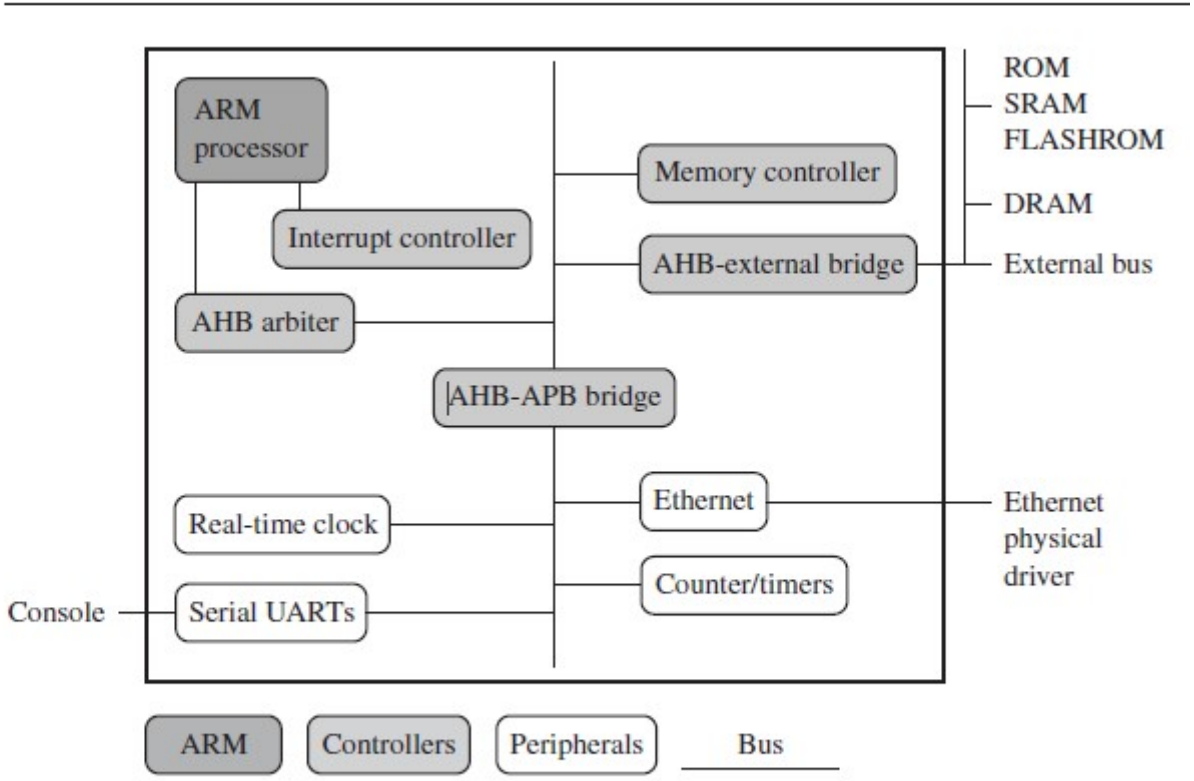


Figure 1.2 An example of an ARM-based embedded device, a microcontroller.

ARM Core/Implementation

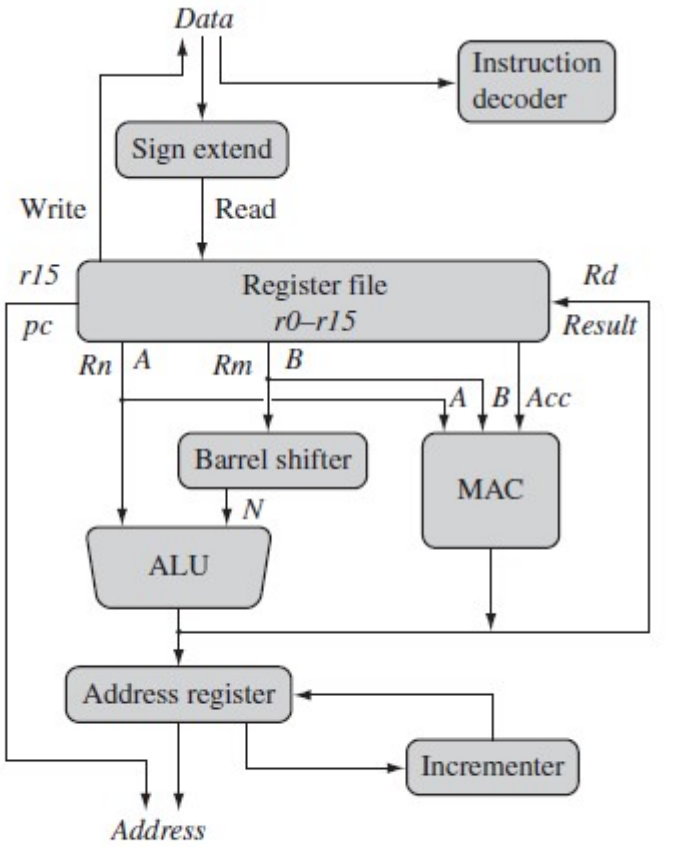


Figure 2.1 ARM core dataflow model.

ARM Specifications

- ARM family or architecture or core?

Family	Architecture	Cores
ARM7TDMI	ARMv4T	ARM7TDMI(S)
ARM9 ARM9E	ARMv5TE(J)	ARM926EJ-S, ARM966E-S
ARM11	ARMv6 (T2)	ARM1136(F), 1156T2(F)-S, 1176JZ(F), ARM11 MPCore™
Profile: Cortex-A	ARMv7-A	Cortex-A5, A7, A8, A9, A15
Cortex-R	ARMv7-R	Cortex-R4(F)
Cortex-M	ARMv7-M	Cortex-M3, M4
	ARMv6-M	Cortex-M1, M0
NEW !	ARMv8-A	64 Bit
NEW !	ARMv9-A	64 Bit

ARM v7 (not ARM7)

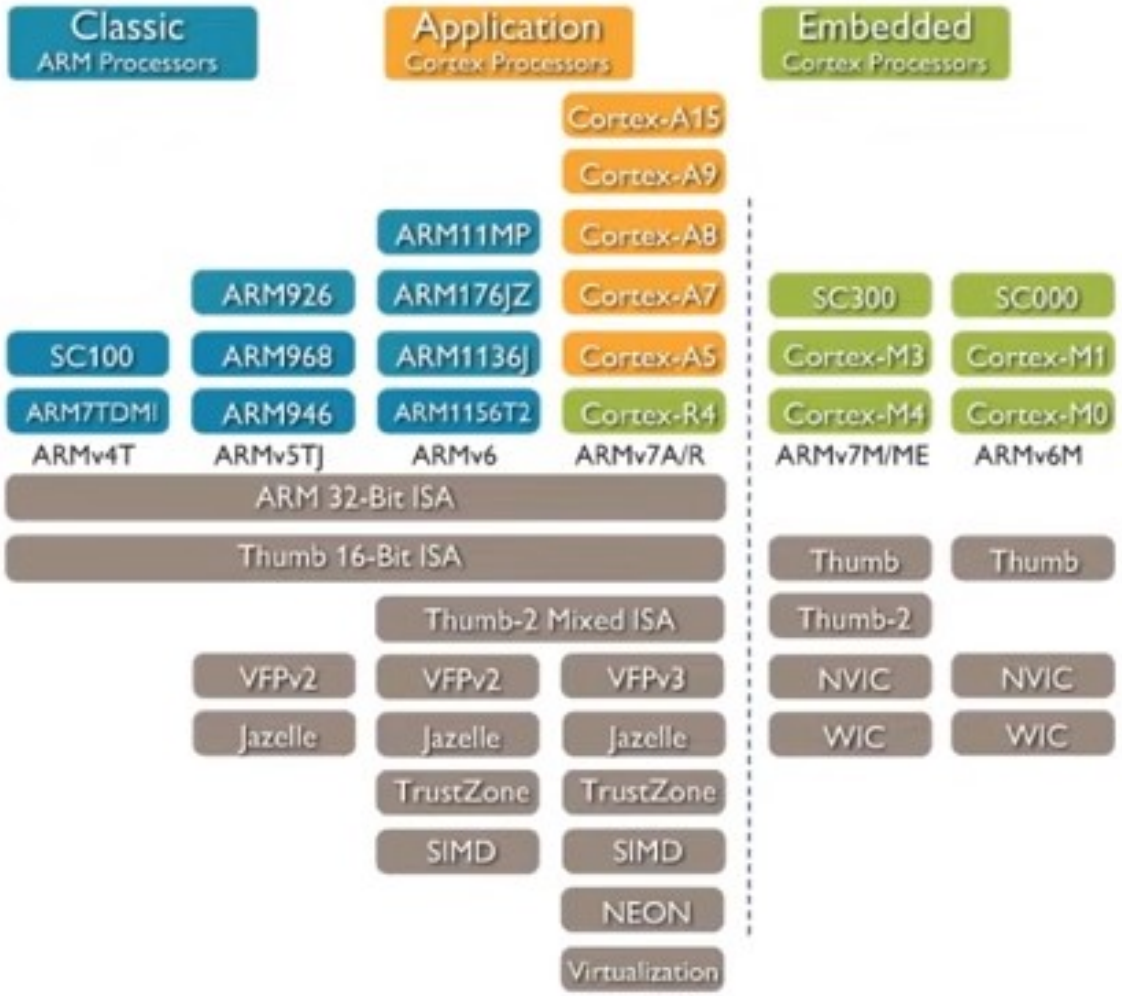
- V7 Architecture

Profiles:

Cortext-A

Cortext-R

Cortext-M



ARM architecture Profile

- Arm Cortex is the brand name used for Arm's processor IP offerings. Our partners offer other processor brands using the Arm architecture.

There are three architecture profiles: A, R and M.

A-profile (Applications)	R-profile (Real-time)	M-profile (Microcontroller)
• High performance	• Targeted at systems with real-time requirements.	• Smallest/lowest power. Small, highly power-efficient devices.
• Designed to run a complex operating system, such as Linux or Windows.	• Commonly found in networking equipment, and embedded control systems.	• Found at the heart of many IoT devices.

ARM Architecture Specifies

Instruction set	• The function of each instruction • How that instruction is represented in memory (its encoding).
Register set	• How many registers there are. • The size of the registers. • The function of the registers. • Their initial state.
Exception model	• The different levels of privilege. • The types of exceptions. • What happens on taking or returning from an exception.
Memory model	• How memory accesses are ordered. • How the caches behave, when and how software must perform explicit maintenance.
Debug, trace, and profiling	• How breakpoints are set and triggered. • What information can be captured by trace tools and in what format.

Microarchitecture

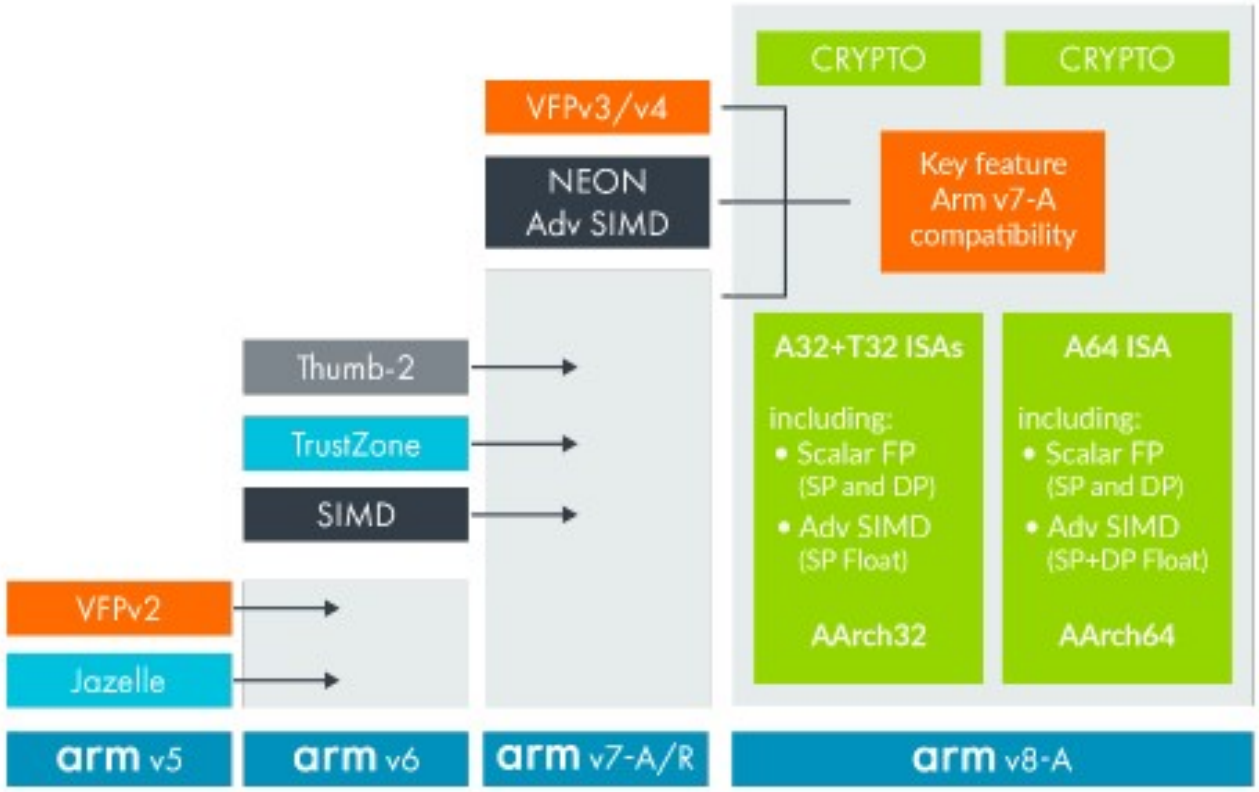
- Architecture does not tell you how a processor is built and works. The build and design of a processor is referred to as micro-architecture. Micro-architecture tells you how a processor works.

- Micro-architecture includes things like:

- Pipeline length and layout.
- Number and sizes of caches.
- Cycle counts for individual instructions.
- Which optional features are implemented.

- For example, Cortex-A53 and Cortex-A72 are both implementations of the Armv8-A architecture. This means that they have the same architecture, but they have very different micro-architectures.
- Software that is architecturally-compliant can run on either the Cortex-A53 or Cortex-A72 without modification, because they both implement the same architecture.

- the development of the Arm architecture from version 5 to version 8, with the new features that were added each time.



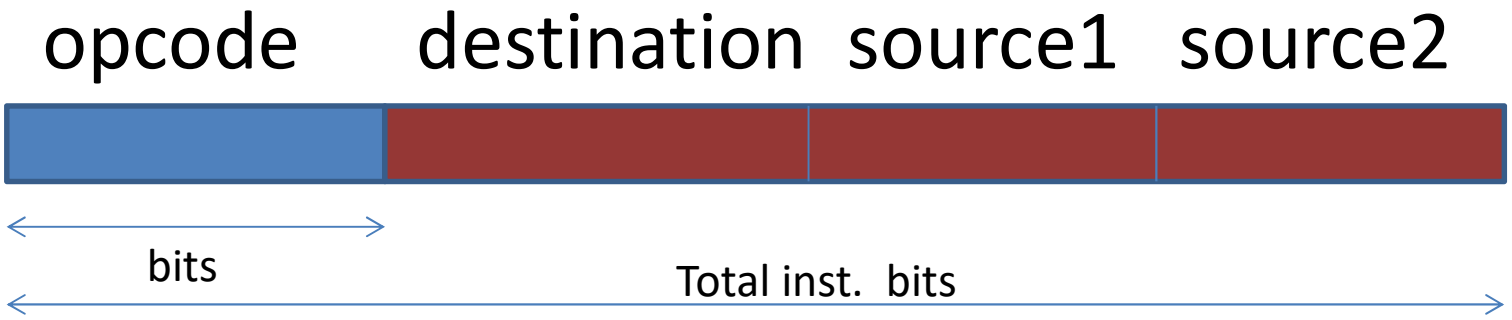
Simple Processor (RISC)

MU0 - a simple processor

A simple form of processor can be built from a few basic components:

- **a program counter** (PC) register that is used to hold the address of the current instruction;
- a single register called an **accumulator** (ACC) that holds a data value while it is worked upon;
- **an arithmetic-logic unit** (ALU) that can perform a number of operations on binary operands, such as add, subtract, increment, and so on;
- **an instruction register** (IR) that holds the current instruction while it is executed;
- instruction decode and control logic that employs the above components to achieve the desired results from each instruction.

Operation Code (opcode)



Three address instruction format

ADD X3, X1, X2 —Effect→ X3 = X1+X2

MU0 of Simple opcode of 4 bits

The MU0 instruction set

MU0 is a 16-bit machine with a 12-bit address space, so it can address up to 8 Kbytes of memory arranged as 4,096 individually addressable 16-bit locations. Instructions are 16 bits long, with a 4-bit operation code (or **opcode)** and a 12-bit address field (S) as shown in Figure 1.4. The simplest instruction set uses only eight of the 16 available opcodes and is summarized in Table 1.1.

An instruction such as 'ACC := ACC + mem_{16}[S]' means 'add the contents of the (16-bit wide) memory location whose address is S to the accumulator'. Instructions are fetched from consecutive memory addresses, starting from address zero, until an instruction which modifies the PC is executed, whereupon fetching starts from the new address given in the 'jump' instruction.

MU0 instruction format

MU0 logic design

To understand how this instruction set might be implemented we will go through the design process in a logical order. The approach taken here will be to separate the design into two components:

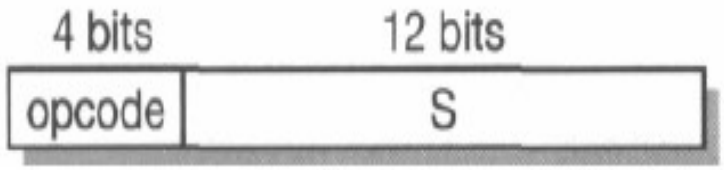


Figure 1.4 The MU0 instruction format.

Simple Instruction Set

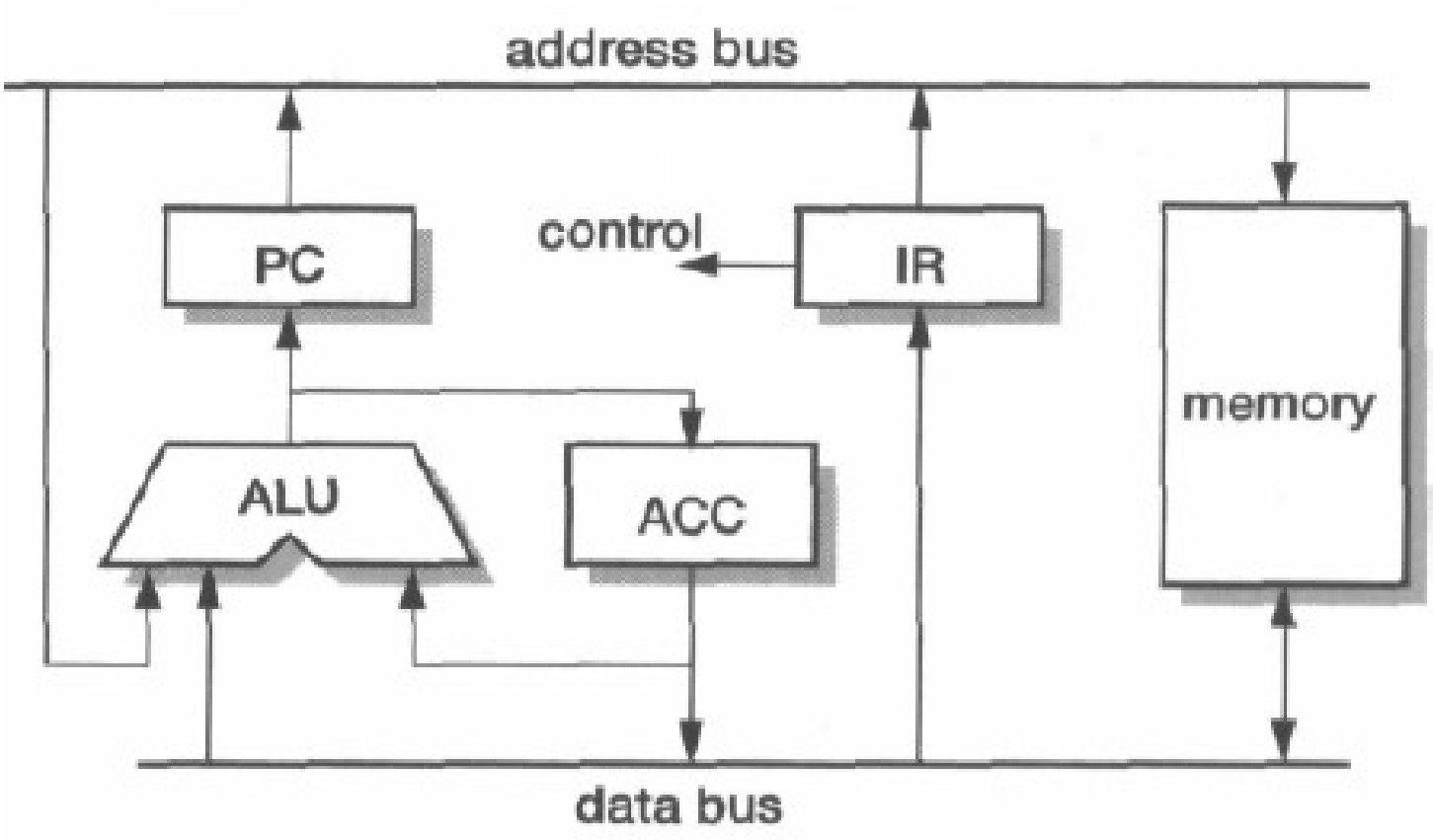
Table 1.1 The MU0 instruction set.

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC – mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop

Datapath design

- Each instruction takes exactly the number of clock cycles defined by the number of memory accesses it must make.
- Referring back to Table 1.1 we can see that the first *four instructions* each require two memory accesses (one to fetch the instruction itself and one to fetch or store the operand) whereas the last four instructions can execute in one cycle since they do not require an operand. (In practice we would probably not worry about the efficiency of the STP instruction since it halts the processor for ever.)

MU0 data path



Data path operation

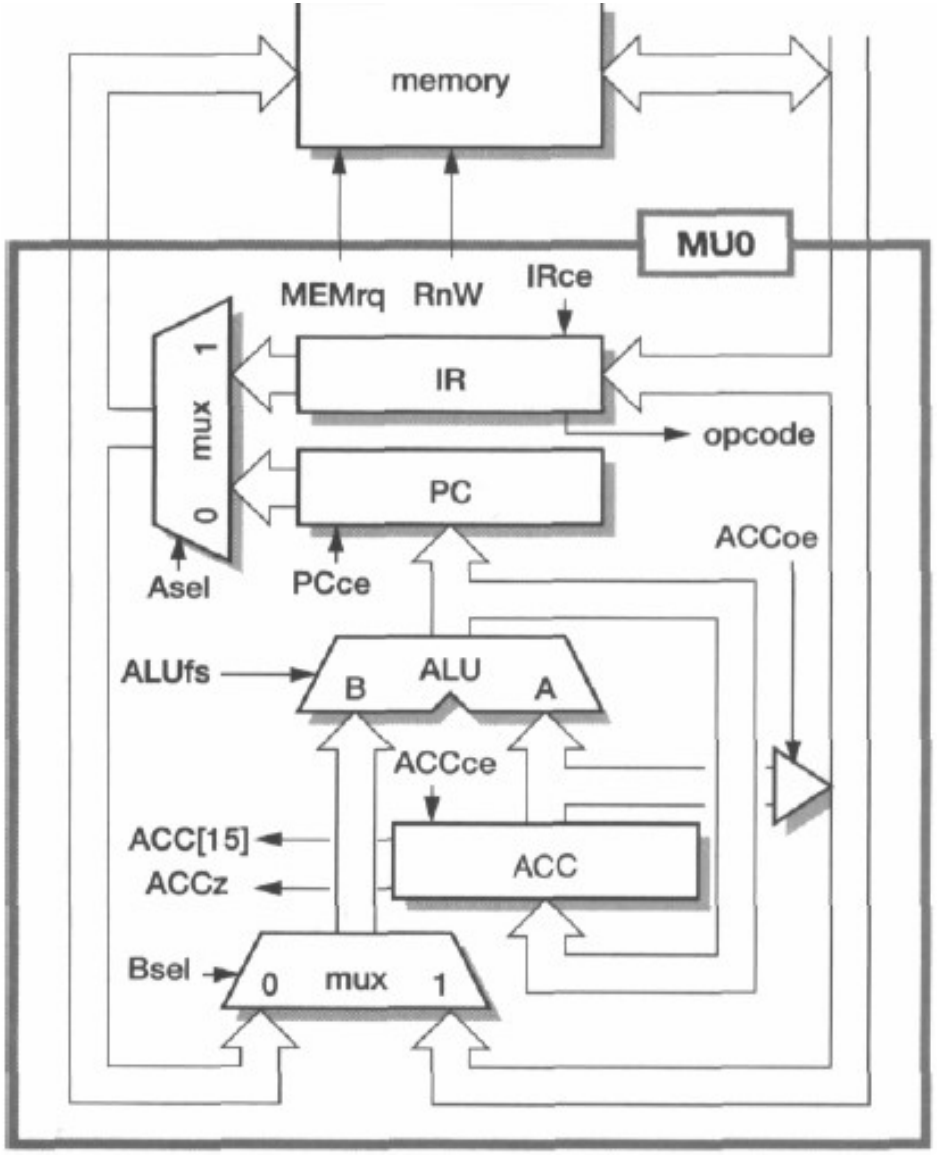
- The design we will develop assumes that each instruction starts when it has arrived in the instruction register.
- Until it is in the instruction register, MU0 cannot know which instruction it is dealing with.
- The processor must start in a known state. Usually this requires a *reset input to cause it* to start executing instructions from a known address.

an instruction executes in two stages, possibly omitting the first of these:

1. Access the memory operand and perform the desired operation. The address in the instruction register is issued and either an operand is read from memory, combined with the accumulator in the ALU and written back into the accumulator, or the accumulator is stored out to memory.
2. Fetch the next instruction to be executed. Either the PC or the address in the instruction register is issued to fetch the next instruction, and in either case the address is incremented in the ALU and the incremented value saved into the PC.

Control Logic design

- The control logic simply has to decode the current instruction and generate the appropriate levels on the datapath control signals, using the control inputs from the datapath where necessary.
- Although the control logic is a finite state machine, and therefore in principle the design should start from a state transition diagram, in this case the FSM is trivial and the diagram not worth drawing.
- The implementation requires only two states, 'fetch' and 'execute', and one bit of state *(Ex/ft) is therefore sufficient.*



MU0 register transfer level organization.

Register transfer level design

The next step is to determine exactly the control signals that are required to cause the datapath to carry out the full set of operations. We assume that all the registers change state on the falling edge of the input clock, and where necessary have control signals that may be used to prevent them from changing on a particular clock edge. The PC, for example, will change at the end of a clock cycle where *PCce* is ' 1' but will not change when *PCce* is '0'.

A suitable register organization is shown in Figure 1.6 on page 11. This shows enables on all of the registers, function select lines to the ALU (the precise number and interpretation to be determined later), the select control lines for two multiplexers, the control for a tri-state driver to send the ACC value to memory and memory request *(MEMrq)* and read/write *(RnW)* control lines. The other signals shown are outputs from the datapath to the control logic, including the opcode bits and signals indicating whether ACC is zero or negative which control the respective conditional jump instructions.

- The control logic can be presented in tabular form as shown in Table 1.2.
- *Once the ALU function select codes* have been assigned the table may be implemented directly as a PLA (programmable logic array) or translated into combinatorial logic and implemented using standard gates.

Table 1.2 MU0 control logic.

Inputs						Outputs									
	Opcode		Ex/ft	ACC15			Bsel		PCce	ACCoe		MEMrq		Ex/ft	
Instruction	↓	Reset ↓		ACCz ↓		Asel ↓	ACCce ↓		IRce ↓		ALUfs ↓	RnW ↓			
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

Instruction set design

If the MU0 instruction set is not a good choice for a high-performance processor, what other choices are there?

Starting from first principles, let us look at a basic machine operation such as an instruction to add two numbers to produce a result.

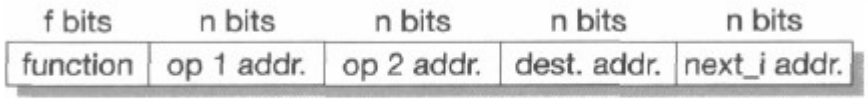


Figure 1.8 A 4-address instruction format.

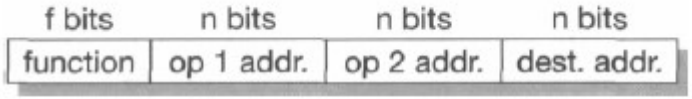


Figure 1.9 A 3-address instruction format.

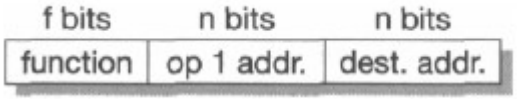


Figure 1.10 A 2-address instruction format.

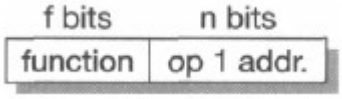


Figure 1.11 A 1-address (accumulator) instruction format.

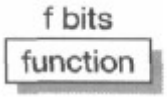


Figure 1.12 A 0-address instruction format.

Addresses

An address in the MU0 architecture is the straightforward 'absolute' address of the memory location which contains the desired operand. However, the three addresses in the ARM 3-address instruction format are register specifiers, not memory addresses. In general, the term '3-address architecture' refers to an instruction set where the two source operands and the destination can be specified independently of each other, but often only within a restricted set of possible values.

Instruction Sets for efficient implementations

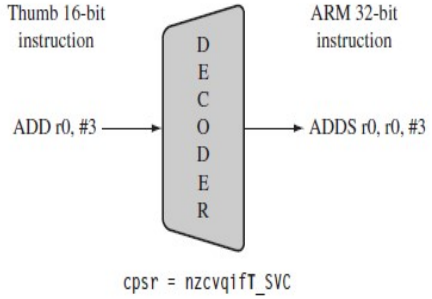
- The art of processor design is to define an instruction set that supports the functions that are useful to the programmer whilst allowing an implementation that is as efficient as possible.
- The ***semantic gap*** between a high-level language construct and a **machine instruction** is bridged by a **compiler,** which is a (usually complex) computer program that translates a high-level language program into a sequence of machine instructions.
- ***Instruction sets*** continue to evolve to give better support for *efficient implementations* and for new applications such as multimedia.

Three Instruction Sets

	ARM	Thumb	Jazelle
Instruction Size	32 bits	16 bits	8 bits
Core instructions	58	30	> 60% of Java byte codes in hardware; rest in software
Conditional Execution	most	Only branch instructions or in an IT block	N/A
Data processing instructions	Access to barrel shifter and ALU	Separate barrel shifter and ALU instructions	N/A
Program status register	Read/write in privileged mode	No direct access	N/A
Register usage	15 general purpose registers + pc	8 general purpose registers + 7 high registers + pc	N/A

Microprocessor Instruction States

- ARM Instruction within active ARM state:
  - Load-Store architecture
  - 32-bit wide and 3-address instruction format
  - Original RISC processor (lots of parallelism)
- Thumb Instruction within active Thumb state:
  - Subset of ARM instruction with some restrictions
  - 16-bit wide
  - Less parallelism
- Jazelle Instruction within active Jazelle state:
  - To speed up Java byte code
  - 8-bit wide



Thumb instruction decoding.

Current Program Status Register (CPSR)

- The ARM core uses the *cpsr register* to monitor and control internal operations and instruction set state activation.
- The *cpsr* is a dedicated 32-bit register and resides in the register file.
- Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future
- expansion.
- The *cpsr* is divided into four fields, each 8 bits wide: flags, status, extension, and control.
- In current designs the extension and status fields are reserved for future use.
- The control field contains the processor mode, state, and interrupt mask bits.
- The flags field contains the condition flags.

Generic Program Status Register (GPSR)

- The current processor mode is stored in the *cpsr*. It holds the current status of the processor core as well interrupt masks, condition flags, and state. The state determines which instruction set is being executed.

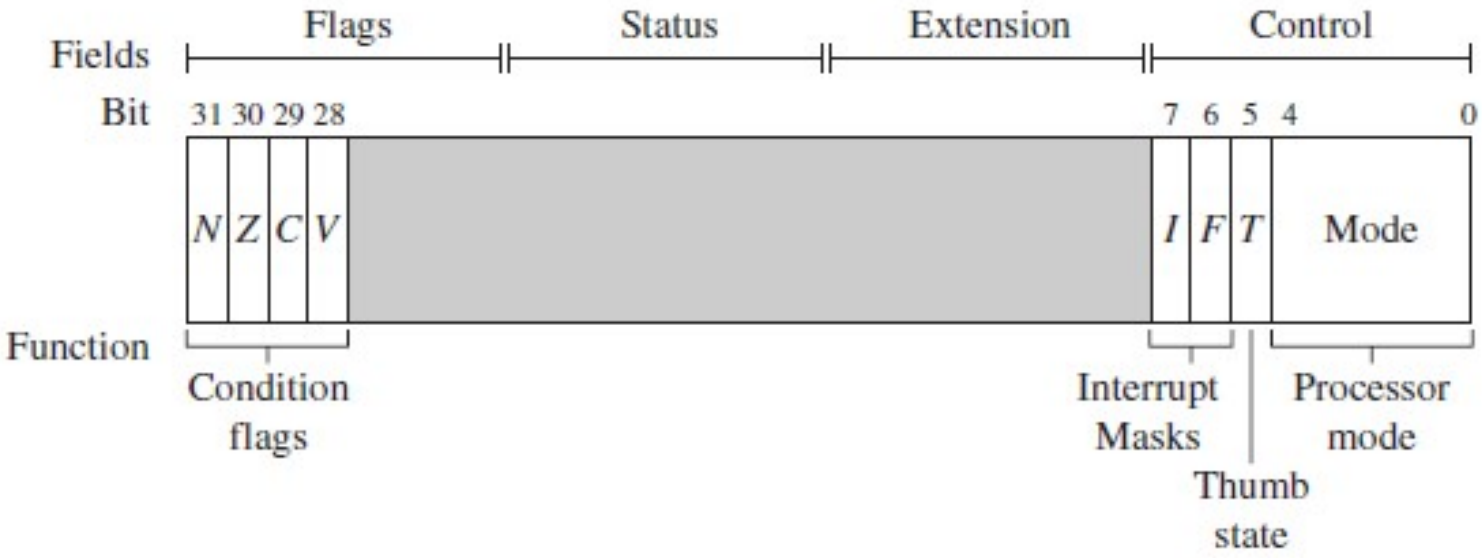


Figure 2.3 A generic program status register

States by CPSR register's T bit

Table 2.2 ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

[a] See Section 2.2.6.

Table 2.3 Jazelle instruction set features.

	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

Some notes

- Note when the processor is in Thumb state the *pc* is the instruction address plus 4.
- You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.
- To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.
- It is important to note that the hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software.

Some notes

- One of the most significant changes to the ISA was the introduction of the Thumb instruction set inARMv4T (the ARM7TDMI processor).
- Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems.
- There are no Thumb instructions to access the coprocessors, and cpsr register.
- Thumb has higher/better *code density*—the space taken up in memory by an executable program—than ARM (see fig 4.1).

Code Density

ARM code

```
ARMDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        SUBS    r0,r0,r1
        ADDGE   r3,r3,#1
        BGE     loop
        ADD     r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

```
ThumbDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        ADD     r3,#1
        SUB     r0,r1
        BGE     loop
        SUB     r3,#1
        ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Figure 4.1 Code density.

References

- A., Sloss, et al., ARM System Developer's Guide.
- Introducing the ARM architecture, ARM document, 2019, sited at www.arm.com.