

# ARM Architecture

*Computer Organization and Assembly Languages*

*Yung-Yu Chuang*

*with slides by Peng-Sheng Chen, Ville Pietikainen*

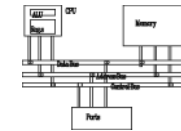
# ARM history

---



- 1983 developed by Acorn computers
  - To replace 6502 in BBC computers
  - 4-man VLSI design team
  - Its simplicity comes from the inexperience team
  - Match the needs for generalized SoC for reasonable power, performance and die size
  - The first commercial RISC implementation
- 1990 ARM (Advanced RISC Machine), owned by Acorn, Apple and VLSI

# ARM Ltd



Design and license ARM core design but not fabricate

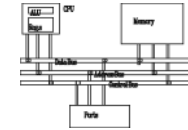
The image displays a central 'ARM in Partnership' logo surrounded by a dense collection of logos for various companies. The logos are organized into four main categories:

- ATAP Partners (Top Left):** Includes logos for WIN-FINITY, BARCO SILEX, SOTA, DNP, Infinite Technology Corporation, SIEMENS, NSW, STEPVIND, Think, MacroTech Research, Inc., COMIT SYSTEMS, YOGITECH, SIDA, SEODU INCHIF, MAZ, ARCADIA, WIPRO, HOYA, INICORE, SCIWORX, FARADAY, MAZ, parthus, nordic, TALITY, FLETRONICS, and synopsys.
- Tools Partners (Top Right):** Includes logos for EPI, ASHLING, CoWare, virtio, Green Hills, INNOVEDA, Computex, YOKOGAWA, ADS, Tektronix, WindRiver, Sophia systems, Axis systems, Verisity, Aptix, Agilent Technologies, Alant Communications, COGENT, ALICATEL, IKOS, IKOS SYSTEMS, LAUTERBACH, and cadence.
- RTOS Partners (Bottom Left):** Includes logos for FIRMWARE SYSTEMS, realogy, BESOL, QNX, LINEO, ACCESS, GEOWORKS, US Software, Tao Systems, KADAK Products Ltd, Microsoft, JAVA, AXE, OSE, Efirmix, WindRiver, MICROWARE, Sun, Etnoteam, Embedded System Products, LYNEXWORKS, and CMX COMPANY.
- Software Partners (Bottom Right):** Includes logos for interniche technologies, inc., Microsoft, EMBLAZE Cellular Technology by CIG, FRONTIER, Packet Video, INTERTRUST THE METATRUST UTILITY, ERICSSON, liquid audio, Bluetooth, symbian, ONX, Dolby, Symmetricom, ASAHI CHEMICAL INDUSTRY CO.,LTD., and CPS.

In the center of the collage is a large globe with the text 'ARM in Partnership' overlaid on it.

# Why ARM?

---



- One of the most licensed and thus widespread processor cores in the world
  - Used in PDA, cell phones, multimedia players, handheld game console, digital TV and cameras
  - ARM7: GBA, iPod
  - ARM9: NDS, PSP, Sony Ericsson, BenQ
  - ARM11: Apple iPhone, Nokia N93, N800
  - 90% of 32-bit embedded RISC processors till 2009
- Used especially in portable devices due to its low power consumption and reasonable performance





# ARM processors

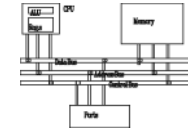
---



- A simple but powerful design
- A whole family of designs sharing similar design principles and a common instruction set

# Naming ARM

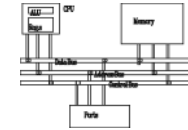
---



- ARMxyzTDMIEJFS
  - x: series
  - y: MMU
  - z: cache
  - T: Thumb
  - D: debugger
  - M: Multiplier
  - I: EmbeddedICE (built-in debugger hardware)
  - E: Enhanced instruction
  - J: Jazelle (JVM)
  - F: Floating-point
  - S: Synthesizable version (source code version for EDA tools)

# Popular ARM architectures

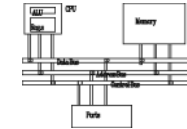
---



- ARM7TDMI
  - 3 pipeline stages (fetch/decode/execute)
  - High code density/low power consumption
  - One of the most used ARM-version (for low-end systems)
  - All ARM cores after ARM7TDMI include TDMI even if they do not include TDMI in their labels
- ARM9TDMI
  - Compatible with ARM7
  - 5 stages (fetch/decode/execute/memory/write)
  - Separate instruction and data cache
- ARM11



# ARM family comparison



ARM family attribute comparison.

year	1995	1997	1999	2003
	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz <sup>a</sup>	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS <sup>b</sup> /MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

<sup>a</sup> Watts/MHz on the same 0.13 micron process.

<sup>b</sup> MIPS are Dhrystone VAX MIPS.

# ARM is a RISC

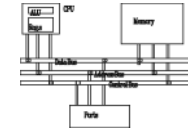
---



- RISC: simple but powerful instructions that execute within a single cycle at high clock speed.
- Four major design rules:
  - Instructions: reduced set/single cycle/fixed length
  - Pipeline: decode in one stage/no need for microcode
  - Registers: a large set of general-purpose registers
  - Load/store architecture: data processing instructions apply to registers only; load/store to transfer data from memory
- Results in simple design and fast clock rate
- The distinction blurs because CISC implements RISC concepts

# ARM design philosophy

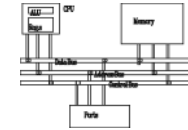
---



- Small processor for lower power consumption (for embedded system)
- High code density for limited memory and physical size restrictions
- The ability to use slow and low-cost memory
- Reduced die size for reducing manufacture cost and accommodating more peripherals

# ARM features

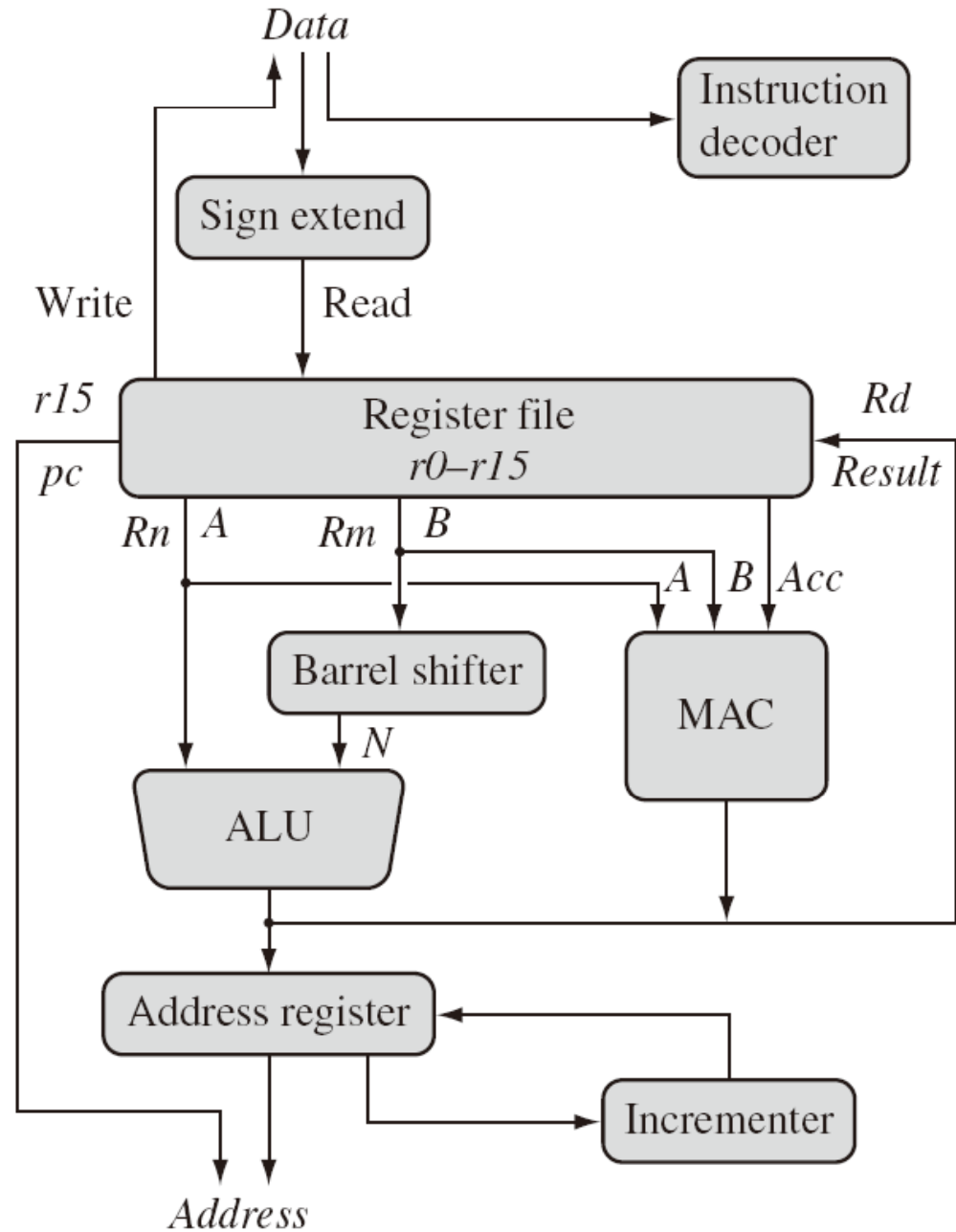
---



- Different from pure RISC in several ways:
  - Variable cycle execution for certain instructions: multiple-register load/store (faster/higher code density)
  - Inline barrel shifter leading to more complex instructions: improves performance and code density
  - Thumb 16-bit instruction set: 30% code density improvement
  - Conditional execution: improve performance and code density by reducing branch
  - Enhanced instructions: DSP instructions

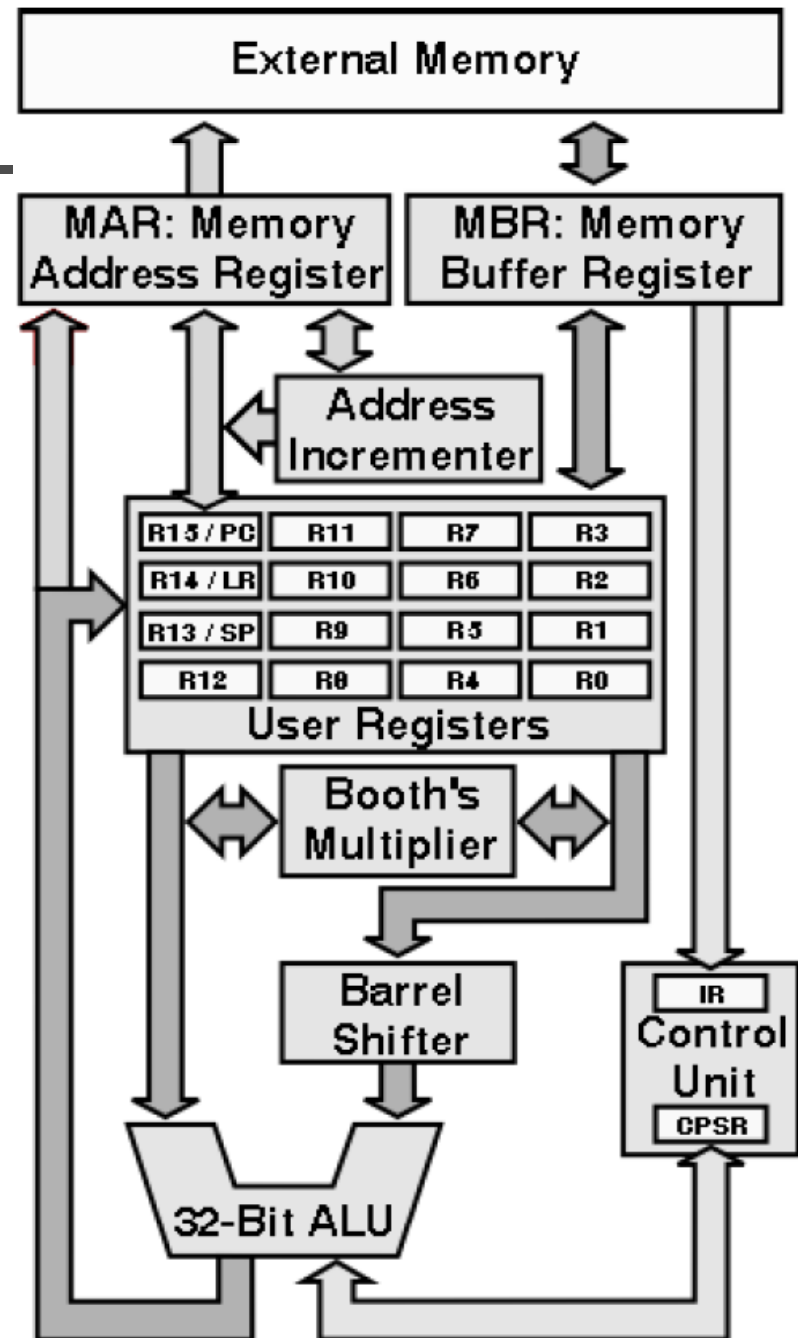
# ARM architecture

---



# ARM architecture

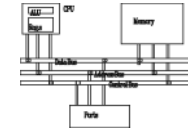
- Load/store architecture
- A large array of uniform registers
- Fixed-length 32-bit instructions
- 3-address instructions





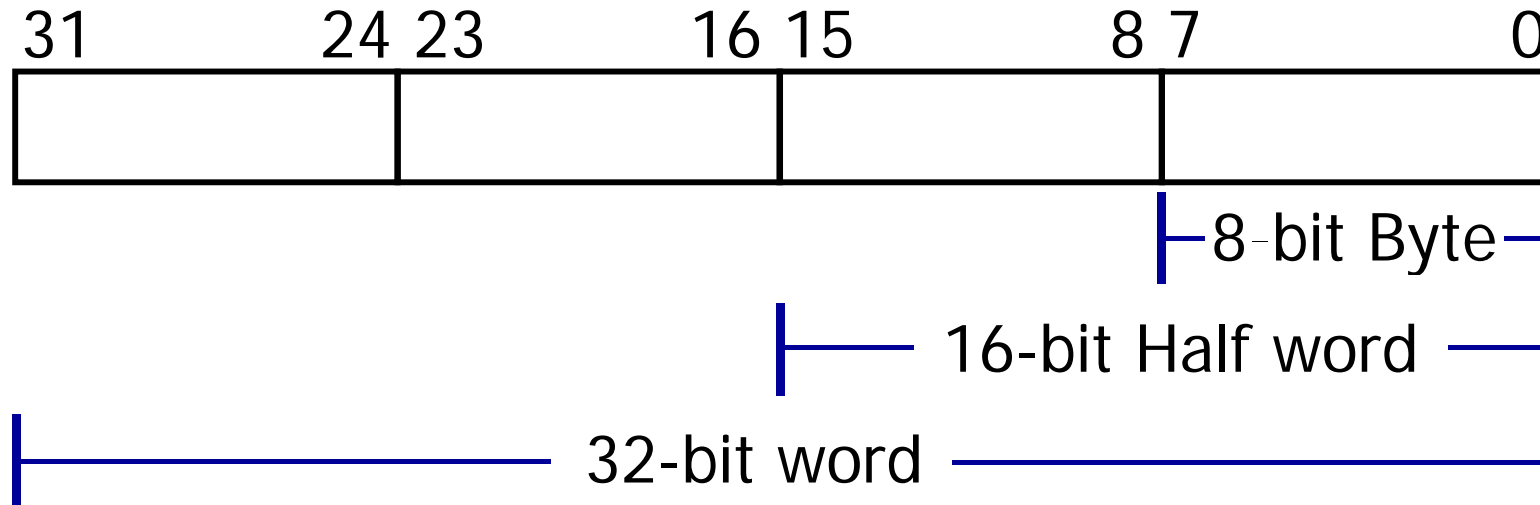
# Registers

---



- Only 16 registers are visible to a specific mode.  
A mode could access
  - A particular set of r0-r12
  - r13 (sp, stack pointer)
  - r14 (lr, link register)
  - r15 (pc, program counter)
  - Current program status register (cpsr)
  - The uses of r0-r13 are orthogonal

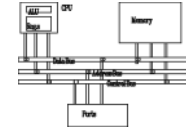
# General-purpose registers



- 6 data types (signed/unsigned)
- All ARM operations are 32-bit. Shorter data types are only supported by data transfer operations.

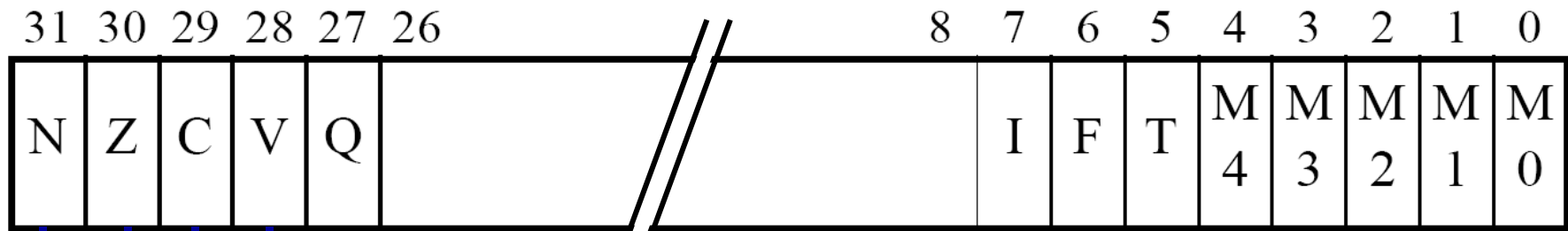
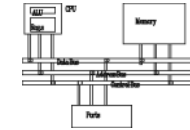
# Program counter

---



- Store the address of the instruction to be executed
- All instructions are 32-bit wide and word-aligned
- Thus, the last two bits of pc are undefined.

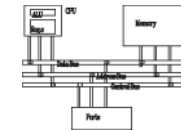
# Program status register (CPSR)



negative  
zero  
carry/borrow  
overflow

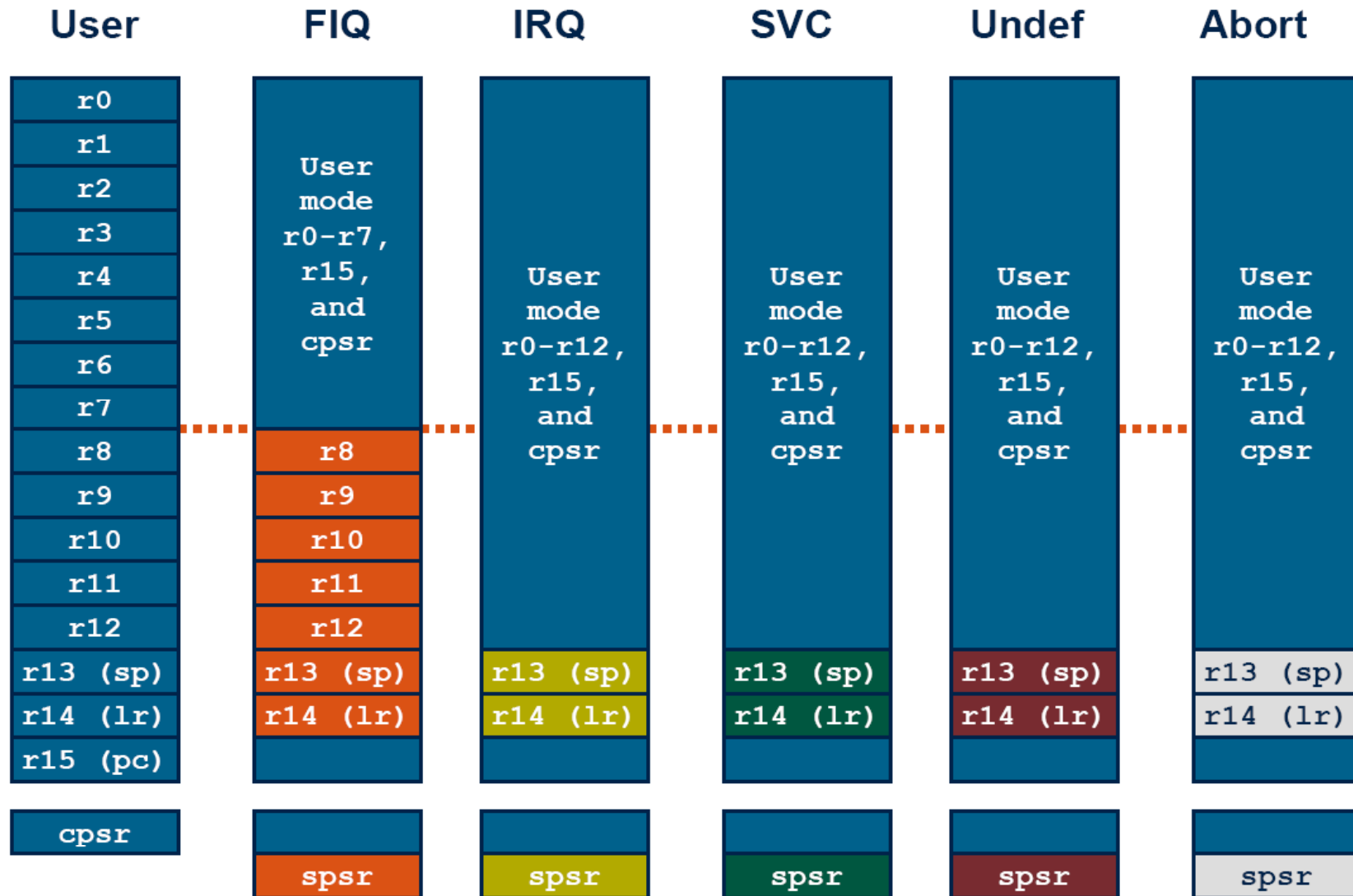
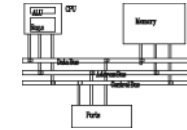
mode bits  
Thumb state  
FIQ disable  
IRQ disable

# Processor modes



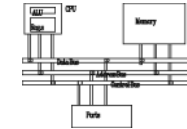
Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

# Register organization





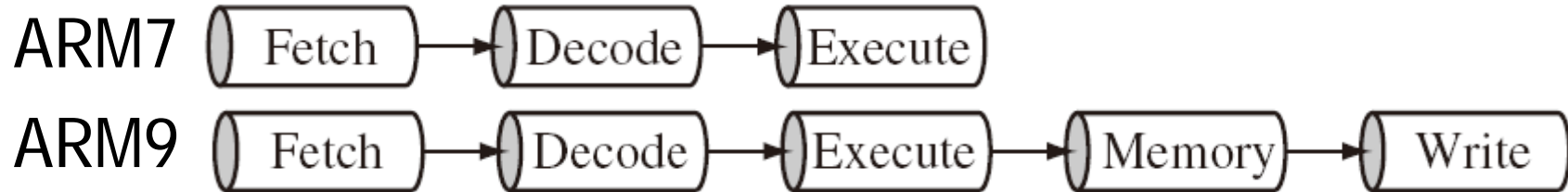
# Instruction sets



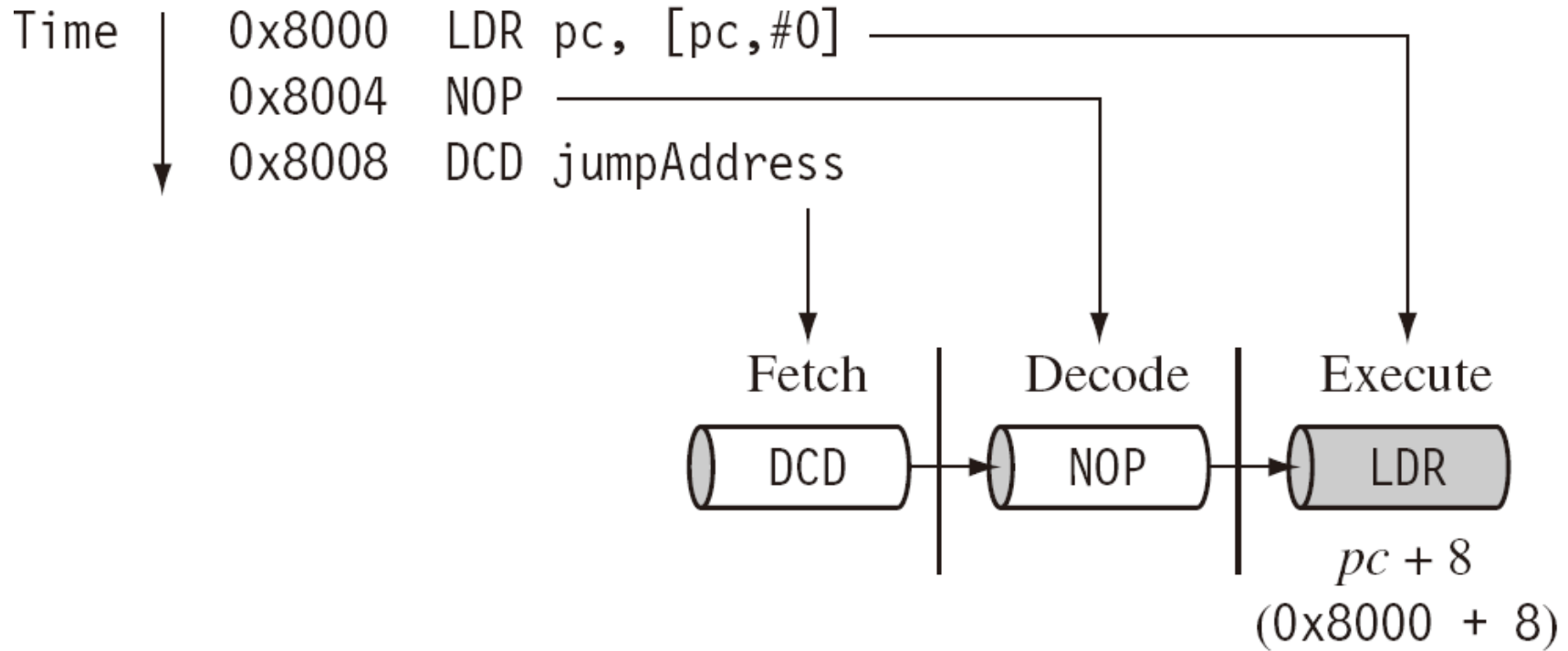
- ARM/Thumb/Jazelle

	ARM ( <i>cpsr</i> $T = 0$ )	Thumb ( <i>cpsr</i> $T = 1$ )
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution <sup>a</sup>	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 + <i>pc</i>	8 general-purpose registers + 7 high registers + <i>pc</i>
Jazelle ( <i>cpsr</i> $T = 0, J = 1$ )		
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.	

# Pipeline

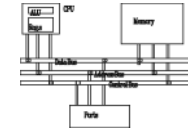


In execution, pc always 8 bytes ahead



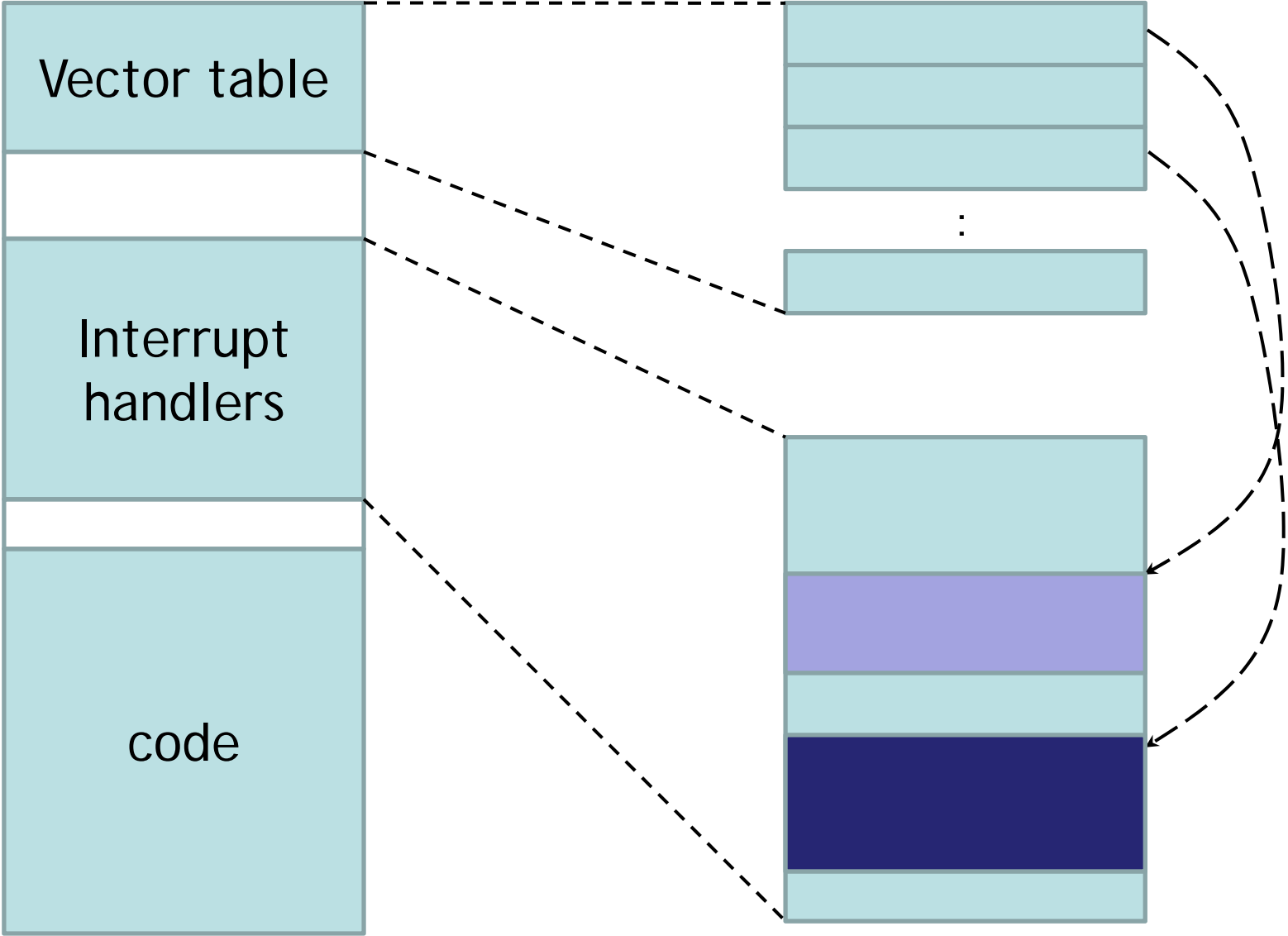
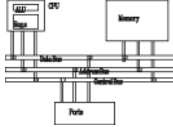
# Pipeline

---



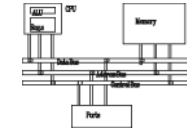
- Execution of a branch or direct modification of pc causes ARM core to flush its pipeline
- ARM10 starts to use branch prediction
- An instruction in the execution stage will complete even though an interrupt has been raised. Other instructions in the pipeline are abandoned.

# Interrupts



# Interrupts

---

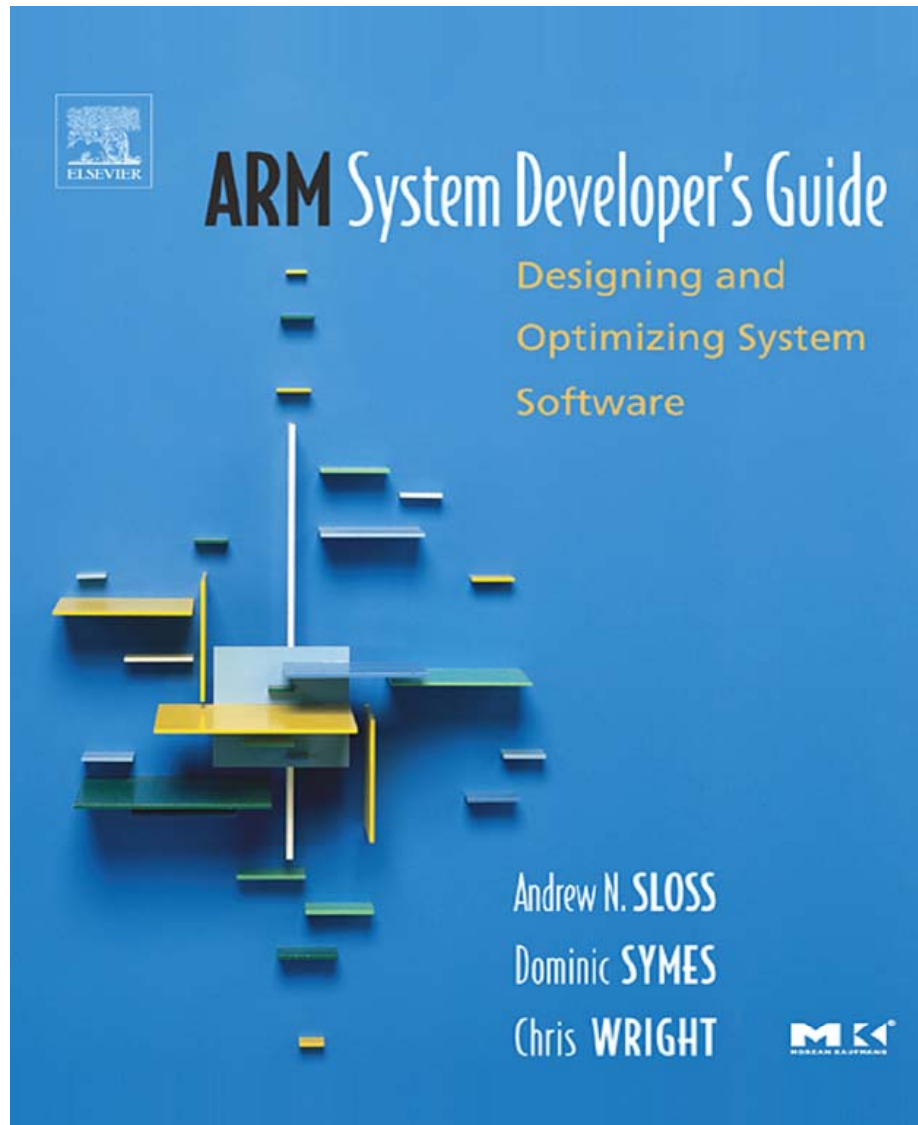
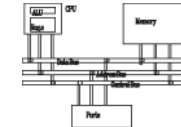


---

Exception/interrupt	Shorthand	Address
Reset	RESET	0x00000000
Undefined instruction	UNDEF	0x00000004
Software interrupt	SWI	0x00000008
Prefetch abort	PABT	0x0000000c
Data abort	DABT	0x00000010
Reserved	—	0x00000014
Interrupt request	IRQ	0x00000018
Fast interrupt request	FIQ	0x0000001c

---

# References



23. Whirlwind Tour of ARM Assembly

- Introduction
- General assembly
- ARM assembly
- THUMB assembly
- LAAZ, the CPU assembler
- A real world example: the 2002 file server

23.1. Introduction

Very broadly speaking, you can divide programming languages into 4 classes. At the lowest level is machine code: raw numbers that the CPU decodes into instructions to execute. One step up is assembly. This is essentially machine code as words: each assembly instruction corresponds to one machine code instruction. Above that are compiled languages like C, which use structured language elements to read more like English, but need to be compiled to machine code to be able to run. Finally, there are scripted languages like PHP (and formerly VB) and Java, which are run through interpreters configured to run the right kinds of machine code for the desired effects.

Every step up the ladder increases the human readability factor and portability, at the cost of runtime speed and program size. In the old days, programmers wrote Real Programs and did these work in machine code or assembly because of clock speed and/or memory constraints. For PCs, these days are long gone and most work is done in the higher level languages. This, admittedly, is a good thing: code can be written faster and maintained more easily. However, there are still a few corners where the higher languages are insufficient. The GBA, with its 16.7MHz CPU and less than 1MB of work RAM is one of them. Here the efficiency of the highest languages will cost you dearly, if it's not at all. This is why most GBA work is done in C/C++, sometimes affectionately nicknamed 'portable assembly', because it still has the capability of working with memory directly. But sometimes even that isn't enough. Sometimes you really have to make every cycle count. And for this, you need assembly.

Now, in case you're the word 'assembly' can be used to frighten small programmers. Because it is so closely tied to the CPU, you can make it do everything, but that also means you have to do everything. Being close to hardware also means you're bypassing all the safety features that higher languages may have, so that it's much easier to break things. In short, it is harder and more dangerous. Although some may prefer the term "adventurous".

The program in assembly, you need to know how a processor actually works and write in a way it can understand, rather than rely on a compiler or assembler to do it for you. There are no structured for- or while- loops or even if/else branches, just goto, no structs or classes with inheritance, and even datatypes are mostly absent. It's assembly, but the lack of bureaucracy is exactly what makes that code possible. Speed is more acute, there are other reasons why learning assembly might be a good idea. Like I said, it forces you to actually understand how the CPU functions, and you can use that knowledge in your C code as well. A good example of this is the 'best' datatype for variables. Because the ARM processor is 32bit, it will prefer ints for most things, and other types will be slower, sometimes much slower. And while this is obvious from the description of the processor itself, knowledge of assembly will show you why they are slower.

A third reason, and not an unreasonable one, is just for general coolness vs. The very fact that it is harder than higher languages should appeal to your inner geek, who relishes such challenges. The simplicity of the statements themselves have an aesthetic quality as well: no nesting about with classes, different loop styles, operator precedence, etc - it's one line, one opcode and never more than a handful of parameters.

Anyway, about the chapter. A complete document on assembly is nothing less than a full user's manual for a CPU. This would require an entire book in itself, which is not something I'm aiming at. My intention here is to give you an introduction (but a thorough one) to ARM assembly. It explains the most important instructions of the ARM and THUMB instruction sets, what you can and cannot do with them (and a little bit about why). It also covers how to use GCC's assembler to actually assemble the code and how to make your assembly and C files work together. Lastly, it gives an example of a fast memory copier as an illustration of both ARM and Thumb code.

With that information, you should be able to do a lot of stuff, or at least know how to make use of the various reference documents out there. This chapter is not an ebook, I am assuming you have some or all of the following documents:

