Computer organization and architecture

Lesson 2

ARM7TDMI and beyond

RISC and CISC

ARM is a **reduced instruction set computer** (**RISC**) architecture.

The ARM instruction set includes only simple, commonly used instructions.

The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast.

More elaborate operations that are less common are performed using sequences of multiple simple instructions.

Architectures with many complex instructions, such as Intel's x86 architecture, are **complex instruction set computers** (**CISC**).

One CISC operation translates into many, possibly even hundreds, of simple instructions in a RISC machine.

However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

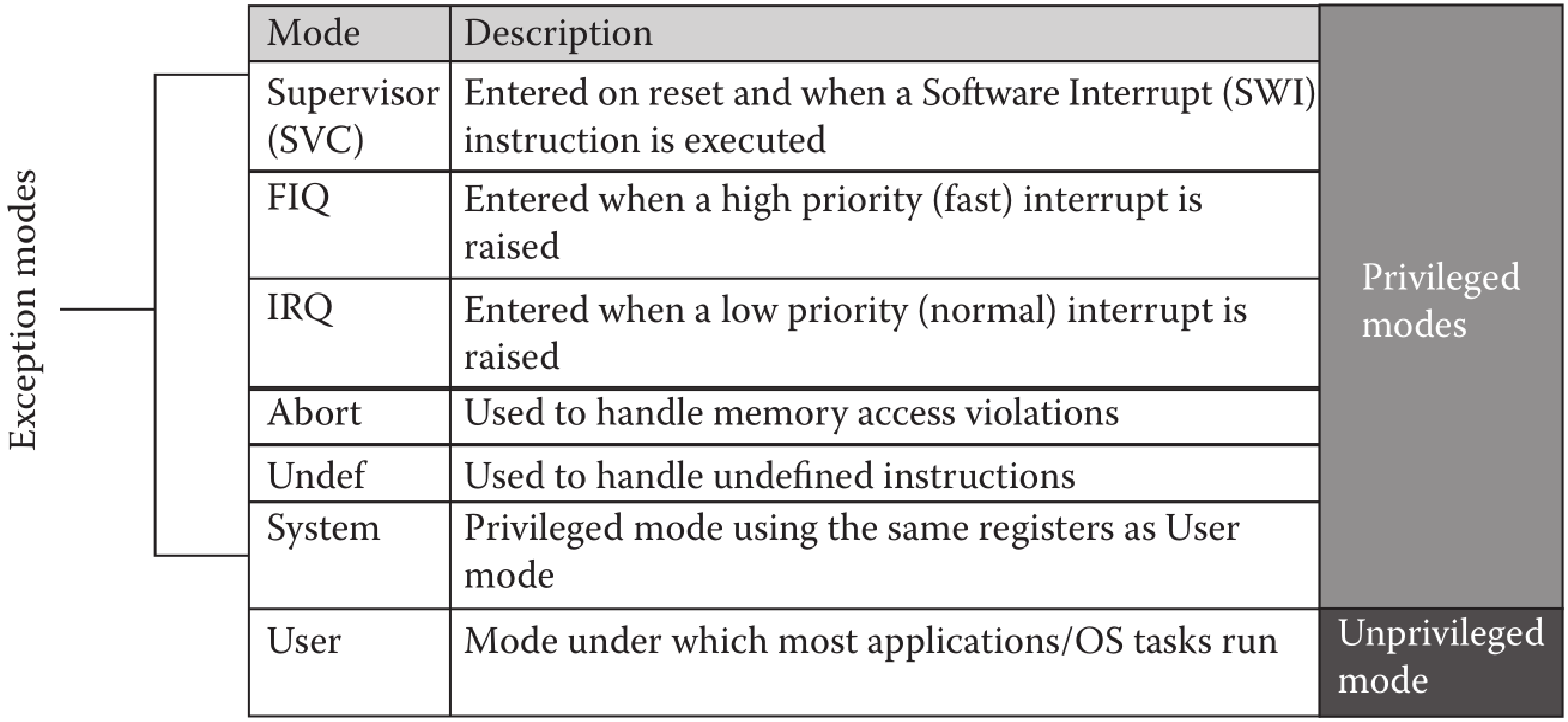
A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small.

An instruction set with 64 simple instructions would need 6 bits to encode the operation.

An instruction set with 256 complex instructions would need 8 bits of encoding per instruction.

In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

ARM7TDMI processor modes


Exception modes
Privileged modes
Unprivileged mode

Mode	Description
Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed
FIQ	Entered when a high priority (fast) interrupt is raised
IRQ	Entered when a low priority (normal) interrupt is raised
Abort	Used to handle memory access violations
Undef	Used to handle undefined instructions
System	Privileged mode using the same registers as User mode
User	Mode under which most applications/OS tasks run

It is possible to make mode changes under software control, but most are normally caused by external conditions or exceptions.

Most application programs will execute in **User mode**.

The other modes are known as **privileged modes**

They provide a way to service exceptions or to access protected resources, such as bits that disable sections of the core.

ARM7TDMI has two types of **interrupts**:

- fast interrupts
- lower priority interrupts

Think of the **fast interrupt** as one that might be used to indicate that the machine is about to lose power in a few milliseconds!

Lower priority interrupts might be used for indicating that a peripheral needs to be serviced, a user has touched a screen, or a mouse has been moved.

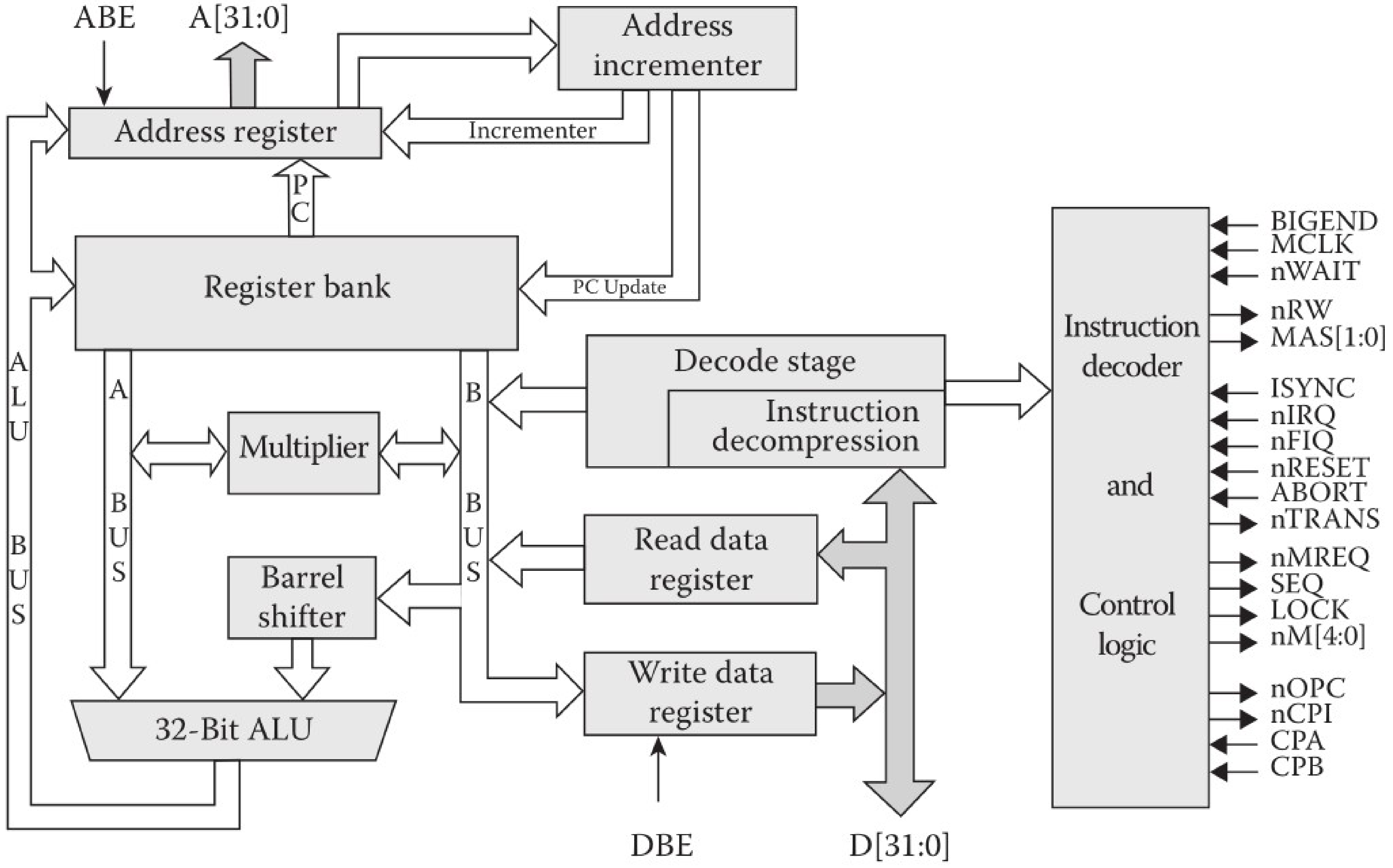
Abort mode allows the processor to recover from exceptional conditions such as a memory access to an address that doesn't physically exist, for either an instruction or data.

This mode can also be used to support virtual memory systems, often a requirement of operating systems such as Linux.

The processor will switch to **Undefined mode** when it sees an instruction in the pipeline that it does not recognize.

It is now the programmer's (or the operating system's) responsibility to determine how the machine should recover from such as error.

The ARM7TDMI



Registers

The ARM7TDMI processor has a total of 37 registers:

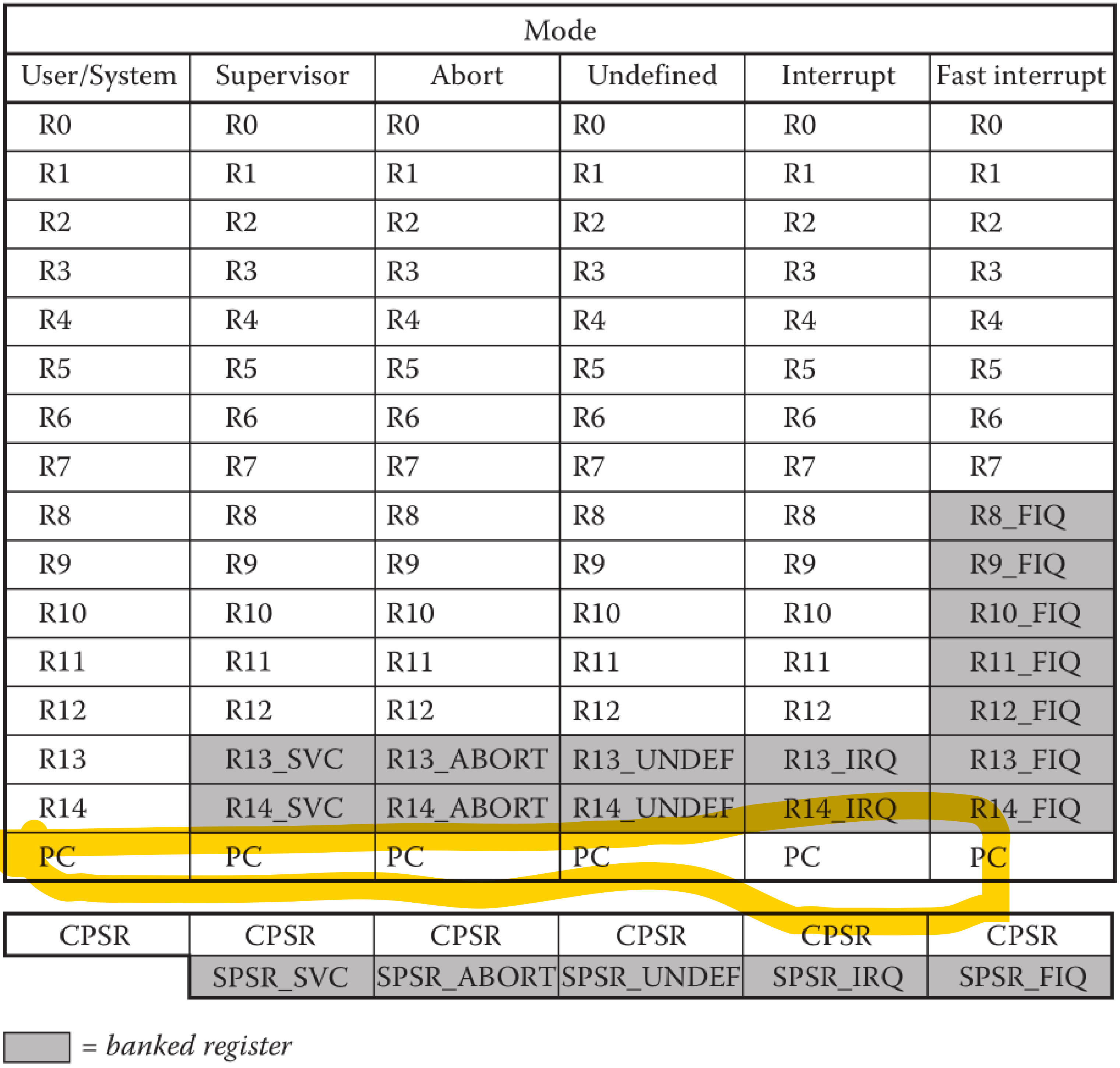
- 30 general-purpose registers
- 6 status registers
- A Program Counter register

The general-purpose registers are 32 bits wide

They are named R0, R1, etc.

The registers are arranged in partially overlapping **banks**, meaning that you as a programmer see a different **register bank** for each processor mode.

At any one time, 15 general-purpose registers (R0 - R14), one or two status registers, and the Program Counter (PC or R15) are visible.



Mode					
User/System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

▨ = *banked register*

The registers have the same names,

but depending on the mode, they are different registers.

In User/System mode, you have registers R0 to R14, a Program Counter, and a Current Program Status Register (CPSR) available.

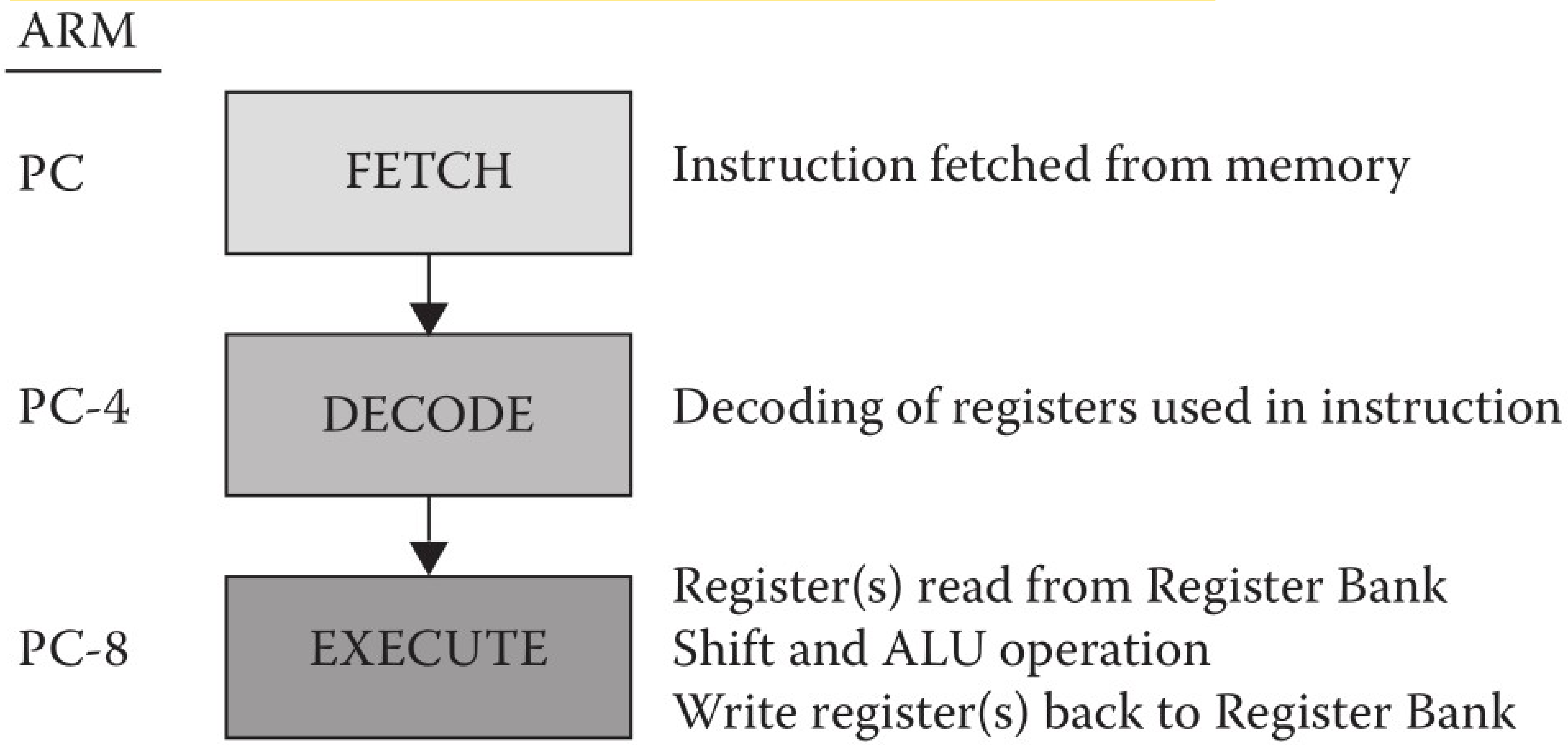
If the processor were to suddenly change to Abort mode, it would swap, or bank out, registers R13 and R14 with different R13 and R14 registers.

The largest number of registers swapped occurs when the processor changes to FIQ mode.

During an interrupt, it is normally necessary to drop everything you’re doing and begin to work on one task: namely, saving the state of the machine and transition to handling the interrupt code quickly.

Rather than moving data from all the registers on the processor to external memory, the machine swaps certain registers with new ones to allow the programmer access to fresh registers.

The ARM7TDMI is a pipelined architecture meaning that while one instruction is being fetched, another is being decoded, and yet another is being executed.


ARM
PC
FETCH
Instruction fetched from memory
PC-4
DECODE
Decoding of registers used in instruction
PC-8
EXECUTE
Register(s) read from Register Bank
Shift and ALU operation
Write register(s) back to Register Bank

The address of the instruction that is being fetched (not the one being executed) is contained in the Program Counter.

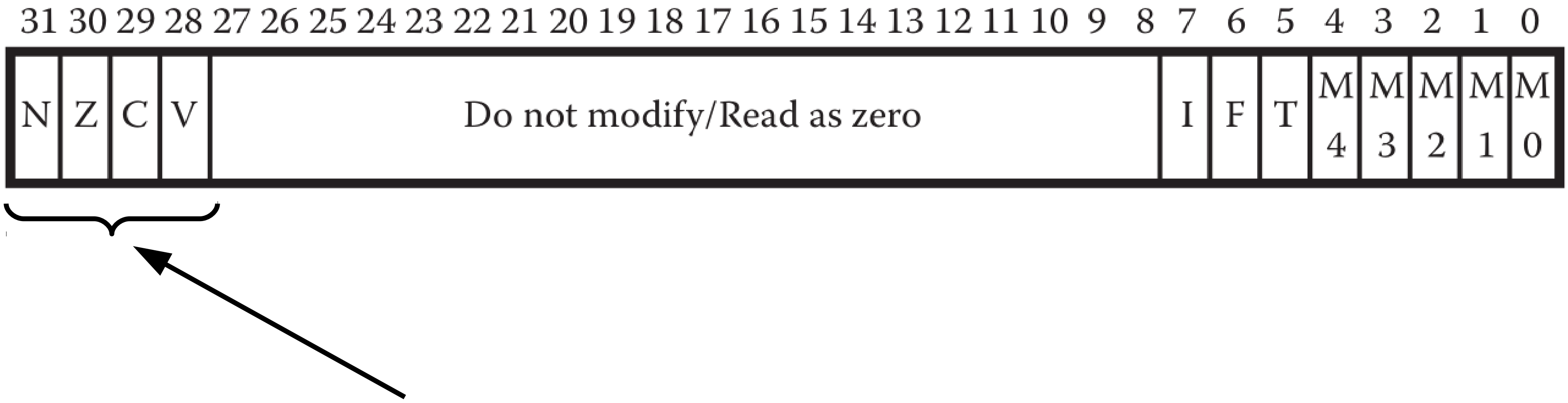
The **Current Program Status Register** (**CPSR**) contains condition code flags, interrupt enable flags, the current mode, and the current state.

Each privileged mode (except System mode) has a **Saved Program Status Register** (SPSR) that is used to preserve the value of the CPSR when an exception occurs.

Since User mode and System mode are not entered on any exception, they do not have an SPSR, and a register to preserve the CPSR is not required.

In later ARM versions, the CPSR is called the **Application Program Status Register** (**APSR**).

The format of the Current Program Status Register and the Saved Program Status Register:



The **condition code flags** in the CPSR can be altered by arithmetic and logical instructions, such as subtractions, logical shifts, and rotations.

By allowing these bits to be used with all the instructions on the ARM7TDMI, the processor can conditionally execute an instruction.

The bottom eight bits of a status register (the mode bits M[4:0], I, F, and T) are known as the control bits.

The I and F bits are the **interrupt disable bits**, which disable interrupts in the processor if they are set.

The I bit controls the IRQ interrupts.

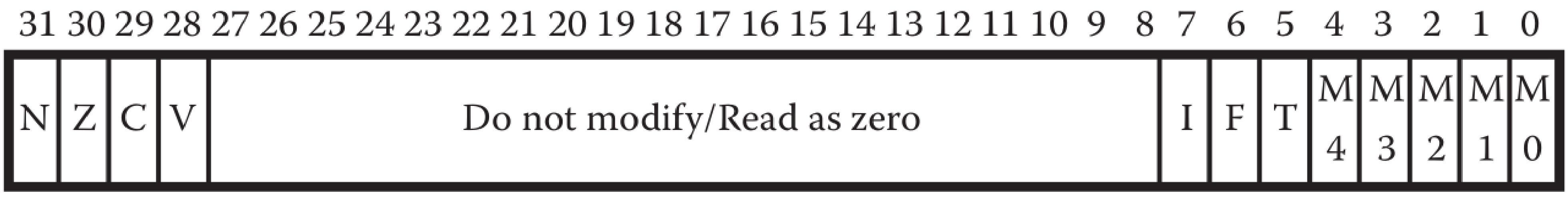
The F bit controls the FIQ interrupts.

The T bit is a status bit, meant only to indicate the state of the machine, so as a programmer you would only read this bit, not write to it.

If the bit is set to 1, the core is executing Thumb code, which consists of 16-bit instructions.

The processor changes between ARM and Thumb state via a special instruction.

These control bits can be altered by software only when the processor is in a privileged mode.



The Mode Bits

xPSR[4:0]	Mode
10000	User mode
10001	FIQ mode
10010	IRQ mode
10011	Supervisor mode
10111	Abort mode
11011	Undefined mode
11111	System mode

?

ARM, Thumb, and Thumb-2 instructions

ARM instructions are 32 bits wide.

Thumb instructions are 16 bits long to achieve higher code density.

They are identical to regular ARM instructions but generally have limitations.

ARM instruction	`ADD R0, R0, R2`
Thumb instruction	`ADD R0, R2`

Almost all ARM instructions have Thumb equivalents.

Because Thumb instructions are less powerful, more are required to write an equivalent program.

However, the instructions are half as long, giving overall Thumb code size of about 65% of the ARM equivalent.

The Thumb instruction set is valuable not only to reduce the size and cost of code storage memory, but also to allow for an inexpensive 16-bit bus to instruction memory and to reduce the power consumed by fetching instructions from the memory.

Thumb instruction encoding is more complex and irregular than ARM instructions to pack as much useful information as possible into 16-bit halfwords.

ARM subsequently refined the Thumb instruction set and added a number of 32-bit **Thumb-2** instructions to boost performance of common operations and to allow any program to be written in Thumb mode.

Thumb-2 is a superset of Thumb instructions, including new 32-bit instructions for more complex operations.

In other words, Thumb-2 is a combination of both 16-bit and 32-bit instructions.

Some cores, such as the Cortex-M3 and M4, only execute Thumb-2 instructions—there are no ARM instructions at all.

The good news is that Thumb-2 code looks very similar to ARM code.

DSP instructions

Digital signal processors (DSPs) are designed to efficiently handle signal processing algorithms such as the Fast Fourier Transform (FFT) and Finite/Infinite Impulse Response filters (FIR/IIR).

Applications include audio and video encoding and decoding, motor control, and speech recognition.

ARM provides a number of DSP instructions for these purposes.

DSP instructions include multiply, add, and multiply-accumulate (MAC)—multiply and add the result to a running sum:

$$\text{sum} = \text{sum} + \text{src1} \times \text{src2}$$

MAC is a distinguishing feature separating DSP instruction sets from regular instruction sets.

It is very commonly used in DSP algorithms and doubles the performance relative to separate multiply and add instructions.

However, MAC requires specifying an extra register to hold the running sum.

Floating-point instructions

Floating-point is widely used in graphics, scientific applications, and control algorithms.

The ARMv5 instruction set includes optional floating-point instructions.

These instructions access at least 16 64-bit double-precision registers separate from the ordinary registers.

These registers can also be treated as pairs of 32-bit single-precision registers.

The registers are named D0–D15 as double-precision or S0–S31 as single-precision.

Power-saving instructions

Battery-powered devices save power by spending most of their time in sleep mode.

ARMv6K introduced instructions to support such power savings.

The wait for interrupt (WFI) instruction allows the processor to enter a low-power state until an interrupt occurs.

The system may generate interrupts based on user events (such as touching a screen) or on a periodic timer.

Virtualization and security instructions

ARMv7 supports virtualization and security.

In virtualization, multiple operating systems can run concurrently on the same processor, unaware of each other's existence.

A hypervisor switches between the operating systems.

With security extensions, the processor defines a secure state with limited means of entry and restricted access to secure portions of memory.

For example, the secure kernel may be used to disable a stolen phone or to enforce digital rights management such that a user can't duplicate copyrighted content.

SIMD instructions

SIMD (pronounced “sim-dee”) – single instruction multiple data

A single instruction acts on multiple pieces of data in parallel.

A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing.

This is also called **packed arithmetic**.

Example: a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components.

Using an entire 32-bit word to process one of these components wastes the upper 24 bits.

When the components from 16 adjacent pixels are packed into a 128-bit **quadword**, the processing can be performed 16 times faster.

Coordinates in a 3-dimensional graphics space are generally represented with 32-bit (single-precision) floating-point numbers.

Four of these coordinates can be packed into a 128-bit quadword.

The ARMv7 Advanced SIMD instructions share the registers from the floating-point unit.

Moreover, these registers can also be paired to act as eight 128-bit quad words Q0–Q7.

The registers pack together several 8-, 16-, 32-, or 64-bit integer or floating-point values.

ARMv6 also defined a more limited set of SIMD instructions operating on the regular 32-bit registers.

These include 8- and 16-bit addition and subtraction, and instructions to efficiently pack and unpack bytes and halfwords into a word.

These instructions are useful to manipulate 16-bit data in DSP code.

ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

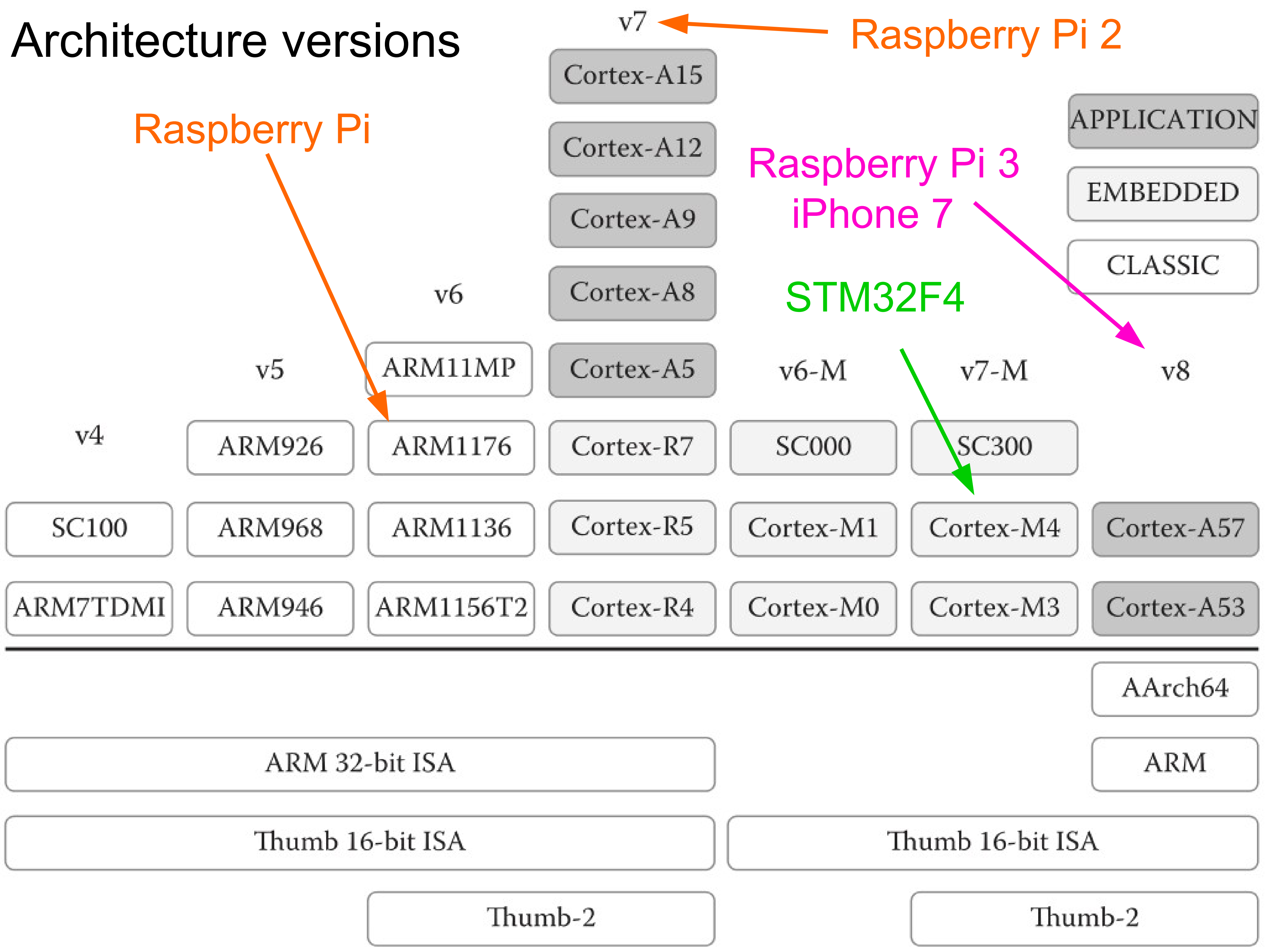
Version 8 of the ARM architecture has been extended to 64 bits.

ARMv8 introduced a new instruction set

The ARMv8 instructions are still 32 bits long and the instruction set looks very much like ARMv7, but with some problems cleaned up.

In ARMv8, the register file is expanded to 31 64-bit registers (called X0–X30) and the PC and SP are no longer part of the general-purpose registers.

Architecture versions



```
    AREA Prog1, CODE, READONLY
    ENTRY
    MOV  R0, #0x11              ; load initial value
    LSL  R1, R0, #1             ; shift 1 bit left
    LSL  R2, R1, #1             ; shift 1 bit left
stop  B  stop                   ; stop program
    END
```

The `AREA` directive instructs the assembler to assemble a new code or data section.

We are creating instructions, not just data (hence the `CODE` option), and we specify the block to be read-only.

```
    AREA SomeData, DATA, READWRITE
```

– contains data, not instructions

You can choose any name for your sections.

Names starting with a non-alphabetic character must be enclosed in bars

```
    AREA |1_DataArea|, CODE, READONLY
```

`READONLY`

Indicates that this section must not be written to.

The section can be placed in read-only memory (default for sections of CODE)

`READWRITE`

Indicates that this section can be read from and written to.

The section can be placed in read-write memory (default for sections of DATA)

Exercise 2.1

Run the code in Keil

Once you've started the debugger, single-step through the code, executing one instruction at a time until you come to the last instruction (the branch).

View the assembly listing as it appears in memory.

View → Memory Windows

In the Disassembly Window, observe the mnemonics in the sample program alongside their equivalent binary representations.

A stored program computer holds instructions in memory.

In this exercise for the ARM7TDMI, memory begins at address 0x00000000 and the last instruction of our program can be found at address 0x0000000C.

Notice that the branch instruction at this address has been changed, and that our label called stop has been replaced with its numerical equivalent

```
0x0000000C EAFFFFFE B 0x0000000C
```

The label `stop` in this case is the address of the B instruction, which is 0x0000000C

In general, a **label** is a name that you choose to represent an address somewhere in memory.

Labels must start at the beginning of the line.

The instructions, directives, and pseudo-instructions must be preceded by a white space, either a tab or any number of spaces, even if you don’t have a label at the beginning.

Directives are directions for the assembler to do something other than simply translate an assembly language instruction into its corresponding machine code.

A breakpoint is an instruction that has been tagged in such a way that the processor stops just before its execution.

To set a breakpoint on an instruction, simply double-click the instruction in the gray bar area.

You can use either the source window or the disassembly window.

You should notice a red circle beside the breakpointed instruction.

When you run your code, the processor will stop automatically upon hitting the breakpoint.

Exercise 2.2 $n! = 1 \times 2 \times \ldots \times (n-1) \times n$

```
   AREA Prog2, CODE, READONLY
   ENTRY
   MOV  r6,#5  ; load n into r6
   MOV  r7,#1  ; if n = 0, at least n! = 1
loop  CMP  r6, #0
   MULGT  r7, r6, r7
   SUBGT  r6, r6, #1 ; decrement n
   BGT  loop  ; do another mul if counter!= 0
stop  B  stop  ; stop program
   END
```

Experiment with RST, Run (F5), and breakpoints at different lines.

Find the command values in the memory.