21CSS201T
COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-5

Contents

- Parallelism: Need, types , applications and challenges
- Architecture of Parallel Systems-Flynn’s classification
- ARM Processor: The thumb instruction set
- Processor and CPU cores, Instruction Encoding format
- Memory load and Store instruction
- Basics of I/O operations.
- Case study: ARM 5 and ARM 7 Architecture

Parallelism

- Executing two or more operations at the same time is known as parallelism.
- Parallel processing is a method to improve computer system performance by executing two or more instructions simultaneously
- A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem.
- Two or more ALUs in CPU can work concurrently to increase throughput
- The system may have two or more processors operating concurrently

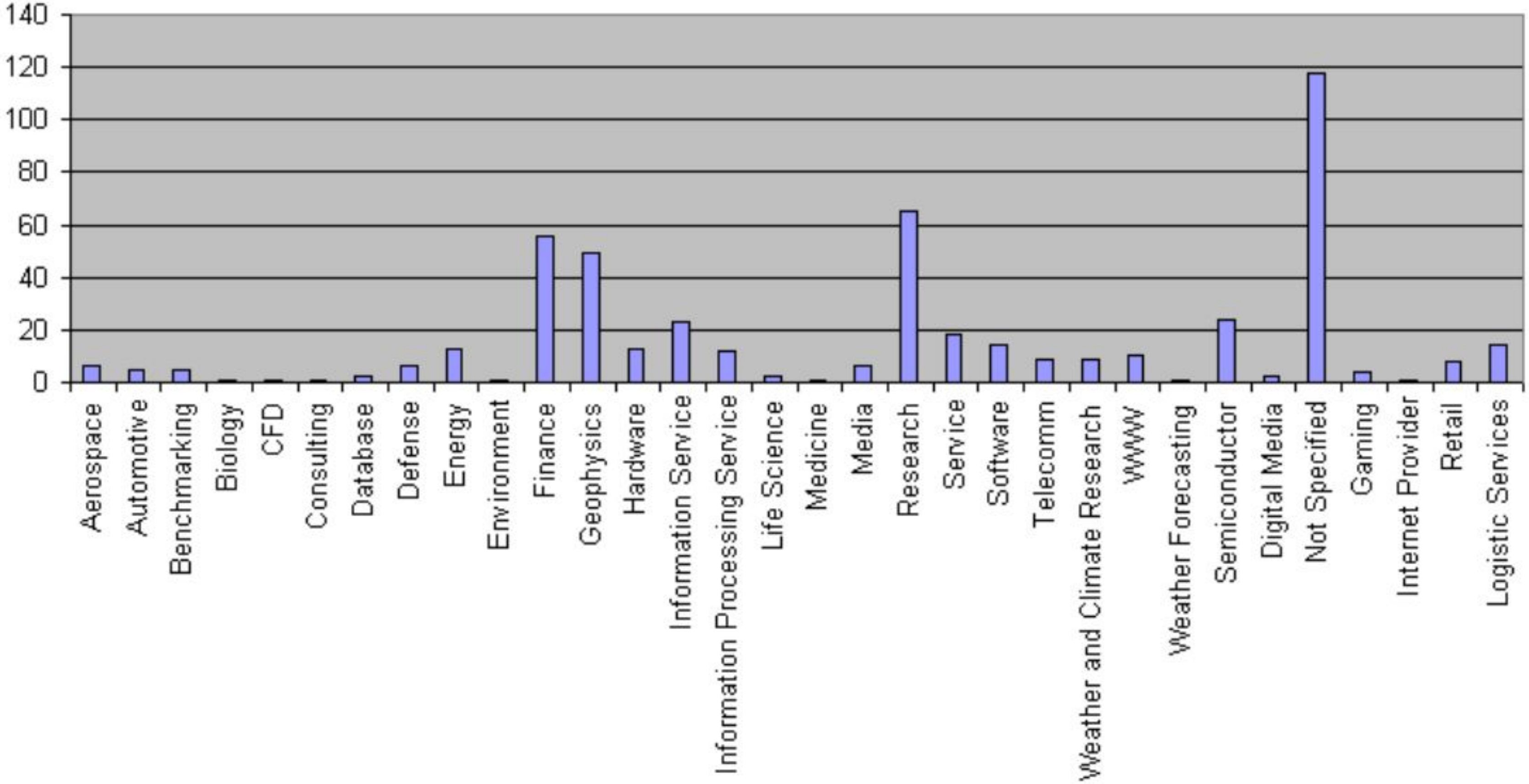
Goals of parallelism

- To increase the computational speed (ie) to reduce the amount of time that you need to wait for a problem to be solved
- To increase throughput (ie) the amount of processing that can be accomplished during a given interval of time
- To improve the performance of the computer for a given clock speed
- To solve bigger problems that might not fit in the limited memory of a single CPU

Applications of Parallelism

- Numeric weather prediction
- Socio economics
- Finite element analysis
- Artificial intelligence and automation
- Genetic engineering
- Weapon research and defence
- Medical Applications
- Remote sensing applications

Applications of Parallelism



Types of parallelism

1. Hardware Parallelism
2. Software Parallelism

- Hardware Parallelism :

  The main objective of hardware parallelism is to increase the processing speed. Based on the hardware architecture, we can divide hardware parallelism into two types: Processor parallelism and memory parallelism.
- **Processor parallelism**

  Processor parallelism means that the computer architecture has multiple nodes, multiple CPUs or multiple sockets, multiple cores, and multiple threads.
- **Memory parallelism** means shared memory, distributed memory, hybrid distributed shared memory, multilevel pipelines, etc. Sometimes, it is also called a parallel random access machine (PRAM). “It is an abstract model for parallel computation which assumes that all the processors operate synchronously under a single clock and are able to randomly access a large shared memory. In particular, a processor can execute an arithmetic, logic, or memory access operation within a single clock cycle”. This is what we call using overlapping or pipelining instructions to achieve parallelism.

Hardware Parallelism

- One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle.
- If a processor issues k instructions per machine cycle, then it is called a **k-issue processor.**
- In a modern processor, two or more instructions can be issued per machine cycle.
- A conventional processor takes one or more machine cycles to issue a single instruction. These types of processors are called **one-issue machines, with a single instruction pipeline in the processor.**
- A multiprocessor system which built n k-issue processors should be able to handle a maximum of nk threads of instructions simultaneously

Software Parallelism

- It is defined by the control and data dependence of programs.
- The degree of parallelism is revealed in the program flow graph.
- Software parallelism is a function of algorithm, programming style, and compiler optimization.
- The program flow graph displays the patterns of simultaneously executable operations.
- Parallelism in a program varies during the execution period .
- It limits the sustained performance of the processor.

Example Detection of parallelism in a program

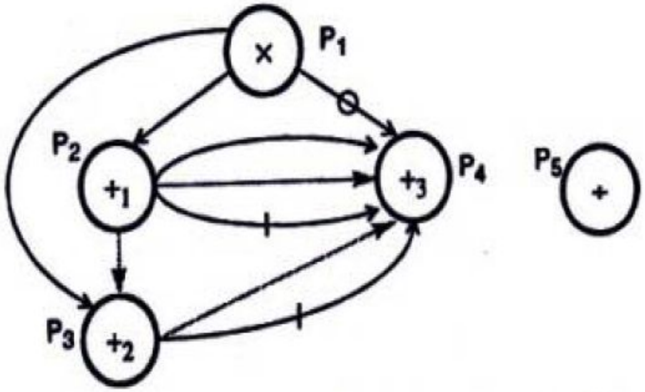
Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following instructions P_1, P_2, P_3, P_4, and P_5.

$$\begin{aligned}
P_1 &: C = D \times E \\
P_2 &: M = G + C \\
P_3 &: A = B + C \\
P_4 &: C = L + M \\
P_5 &: F = G \div E
\end{aligned}$$

Assume that each statement requires one step to execute.

No pipelining is considered here.

The dependence graph is →

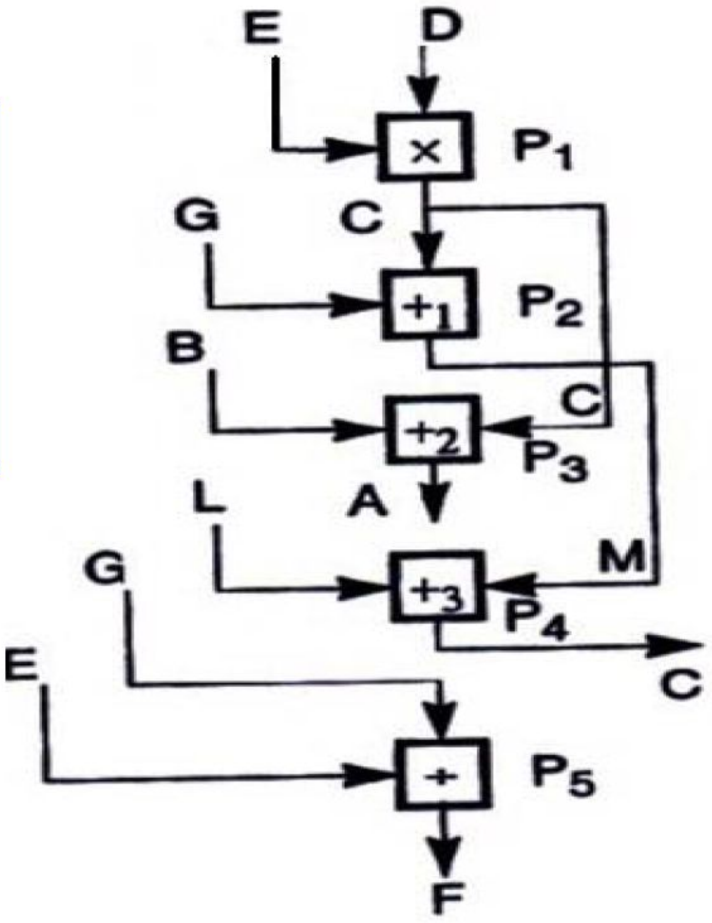


A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)

Sequential Execution:

$$\begin{array}{llcl}
P_1 : & C & = & D \times E \\
P_2 : & M & = & G + C \\
P_3 : & A & = & B + C \\
P_4 : & C & = & L + M \\
P_5 : & F & = & G \div E
\end{array}$$

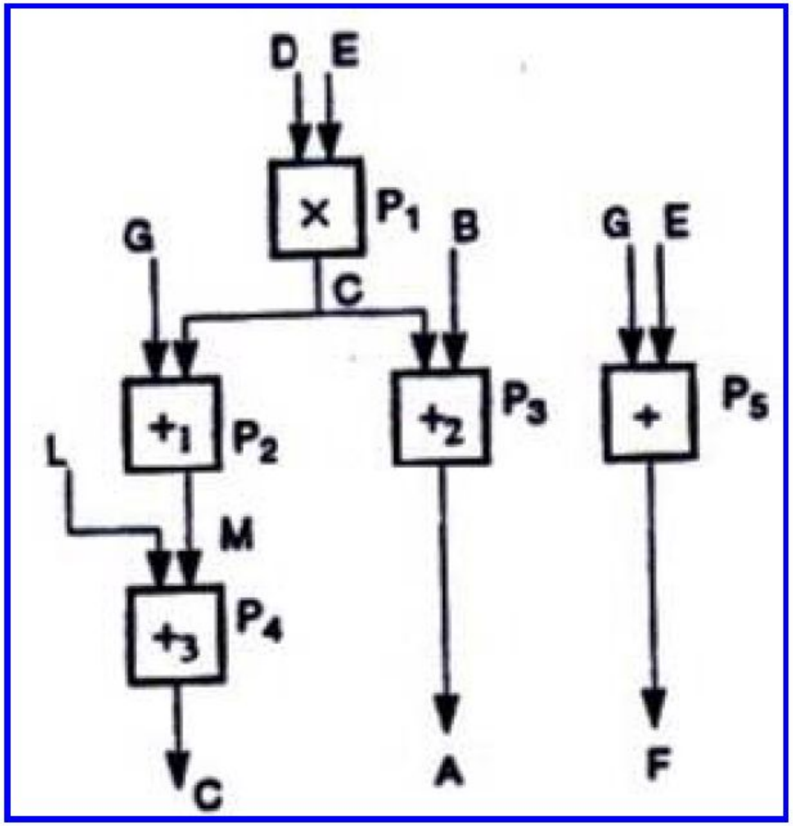
Sequential execution in five steps, assuming one step per statement (no pipelining)



Parallel Execution:

$$
\begin{array}{llcl}
P_1: & C & = & D \times E \\
P_2: & M & = & G + C \\
P_3: & A & = & B + C \\
P_4: & C & = & L + M \\
P_5: & F & = & G \div E
\end{array}
$$

- Parallel execution in THREE steps, assuming TWO adders are available per step.
- If TWO adders are available, the parallel execution requires only THREE steps.
- Only 5 pairs can execute in parallel, if there is no resource conflicts.
- In this example, as shown in the fig, only P2 ‖ P3 ‖ P5 is possible.



Mismatch between S/W & H/W Parallelism:

Example:

$A = L_1 * L_2 + L_3 * L_4$

$B = L_1 * L_2 - L_3 * L_4$

Software Parallelism:

There are 8 instructions;

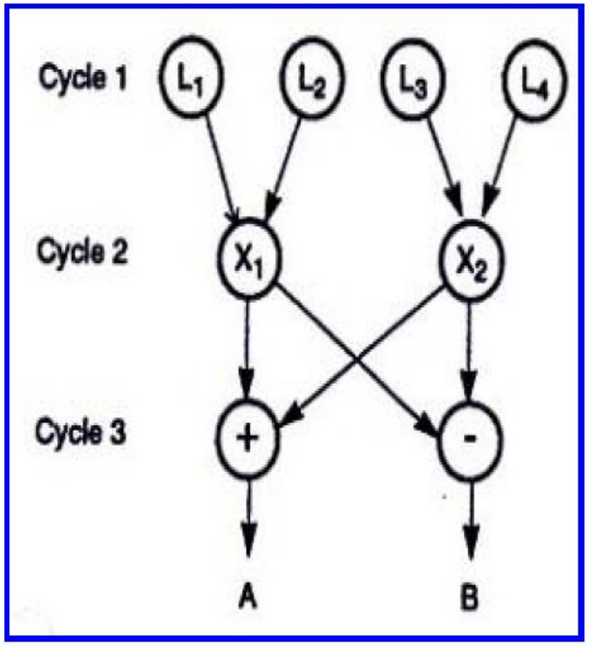
FOUR Load instructions (L1, L2, L3 & L4).

TWO Multiply instructions (X1 & X2).

ONE Add instruction (+)

ONE Subtract instruction (-)

- The parallelism varies from 4 to 2 in three cycles.

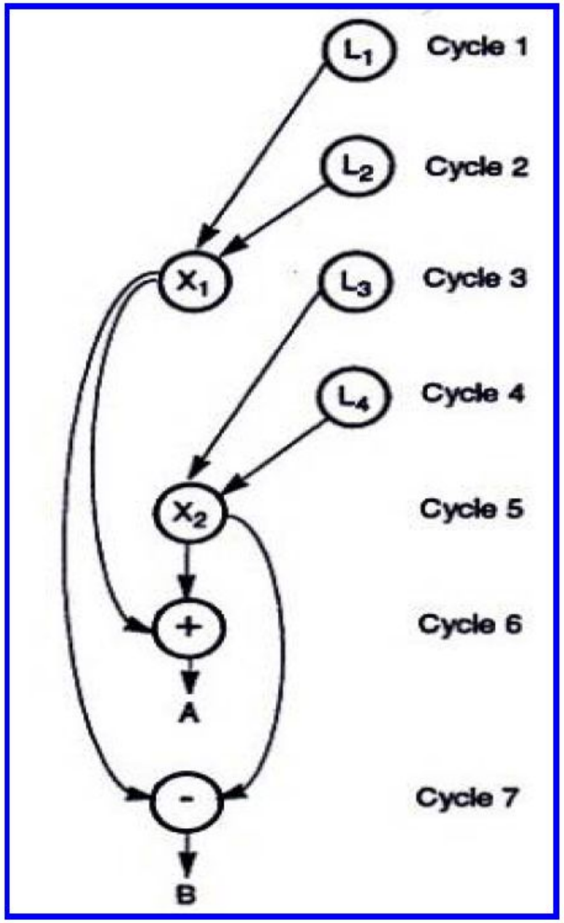


$$\text{Average S/W Parallelism} = \frac{8 \text{ cycles}}{3 \text{ cycles}} = \frac{8}{3} = 2.67$$

Hardware Parallelism:

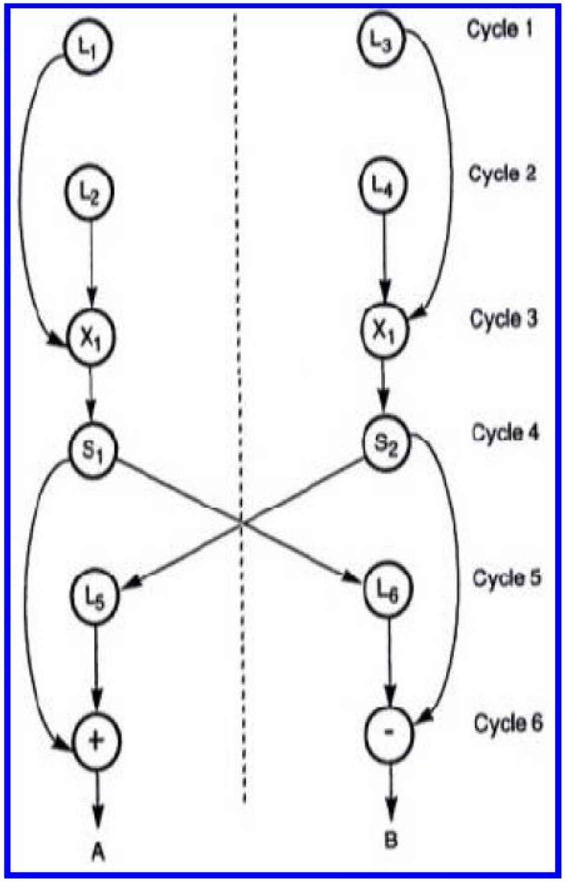
Parallel Execution:

- Using TWO-issue processor:
- The processor can execute one memory access (Load or Store) and one arithmetic operation (multiply, add, subtract) simultaneously.
- The program must execute in 7 cycles.
- The h/w parallelism average is 8/7=1.14.
- It is clear from this example the mismatch between the s/w and h/w parallelism.



Example:

- A h/w platform of a Dual-Processor system, single issue processors are used to execute the same program.
- Six processor cycles are needed to execute the 12 instructions by two processors.
- S1 & S2 are two inserted store operations.
- L5 & L6 are two inserted load operations.
- The added instructions are needed for inter-processor communication through the shared memory.



Software Parallelism - types

Parallelism in Software

- Instruction level parallelism
- Task-level parallelism
- Data parallelism
- Transaction level parallelism

Instruction level parallelism

- Instruction level Parallelism (ILP) is a measure of how many operations can be performed in parallel at the same time in a computer.

- Parallel instructions are set of instructions that do not depend on each other to be executed.
- ILP allows the compiler and processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

Eg. Instruction level parallelism

Consider the following example

1. x= a+b
2. y=c-d
3. z=x * y

Operation 3 depends on the results of 1 & 2

So ‘Z ‘ cannot be calculated until X & Y are calculated

But 1 & 2 do not depend on any other. So they can be computed simultaneously.

- If we assume that each operation can be completed in one unit of time then these 3 operations can be completed in 2 units of time .
- ILP factor is 3/2=1.5 which is greater than without ILP.
- A superscalar CPU architecture implements ILP inside a single processor which allows faster CPU throughput at the same clock rate.

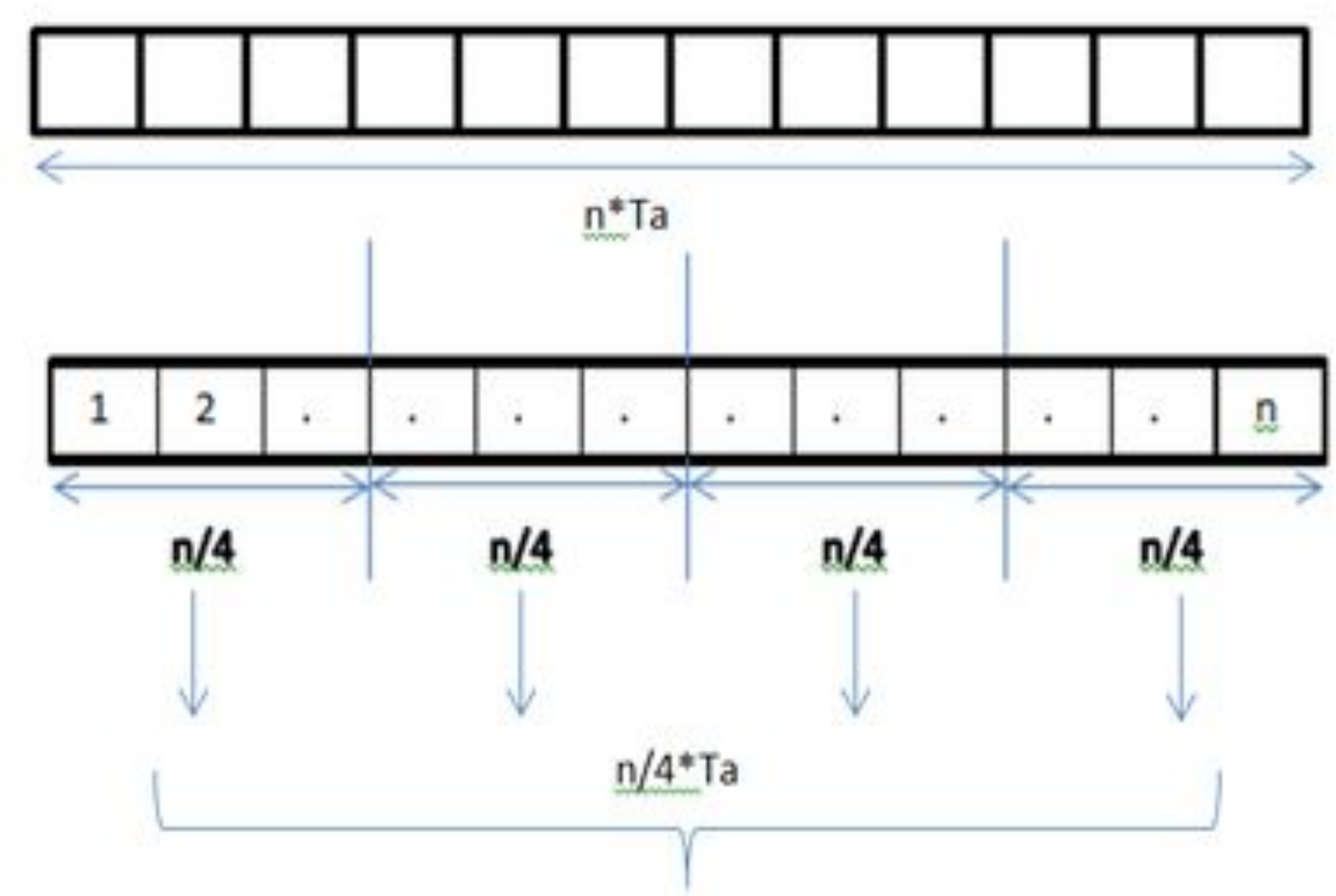
Data-level parallelism (DLP)

- **Data parallelism** is parallelization across multiple processors in **parallel computing** environments.
- It focuses on distributing the **data** across different nodes, which operate on the **data** in **parallel**.
- Instructions from a single stream operate concurrently on several data

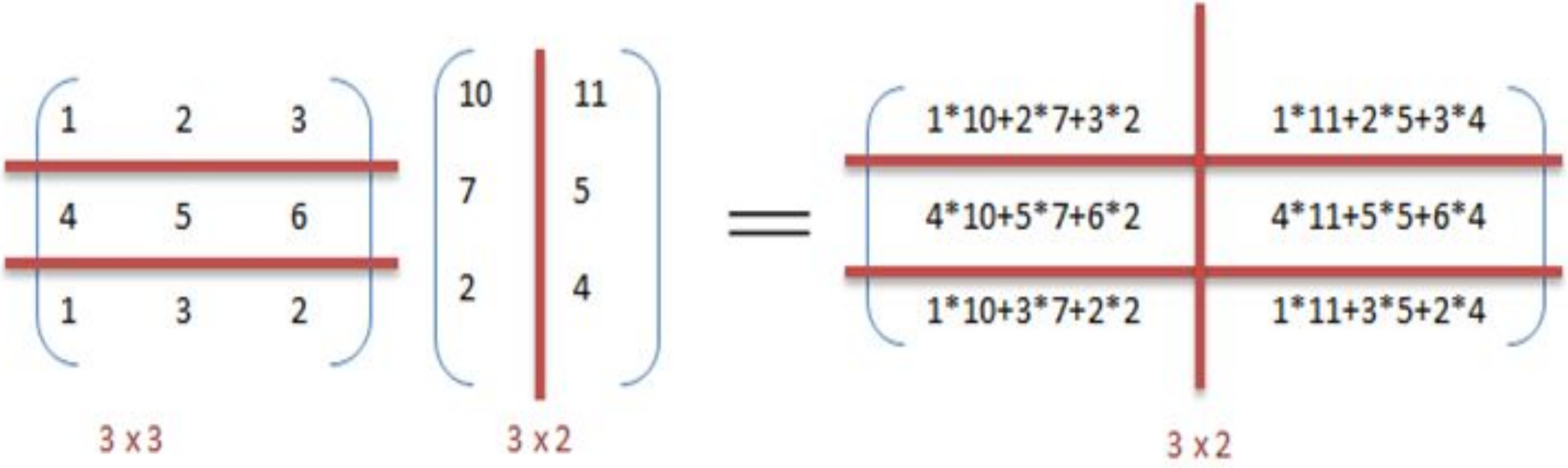
DLP - example

- Let us assume we want to sum all the elements of the given array of size n and the time for a single addition operation is Ta time units.
- In the case of sequential execution, the time taken by the process will be n*Ta time unit
- if we execute this job as a data parallel job on 4 processors the time taken would reduce to (n/4)*Ta + merging overhead time units.

DLP in Adding elements of array



DLP in matrix multiplication



$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} 10 & 11 \\ 7 & 5 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 1*10+2*7+3*2 & 1*11+2*5+3*4 \\ 4*10+5*7+6*2 & 4*11+5*5+6*4 \\ 1*10+3*7+2*2 & 1*11+3*5+2*4 \end{pmatrix}$$

- *A[m x n] dot B [n x k]* can be finished in O(n) instead of O(m∗n∗k) when executed in parallel using *m*k* processors.

- The locality of data references plays an important part in evaluating the performance of a data parallel programming model.
- Locality of data depends on the memory accesses performed by the program as well as the size of the cache.

Flynn’s Classification

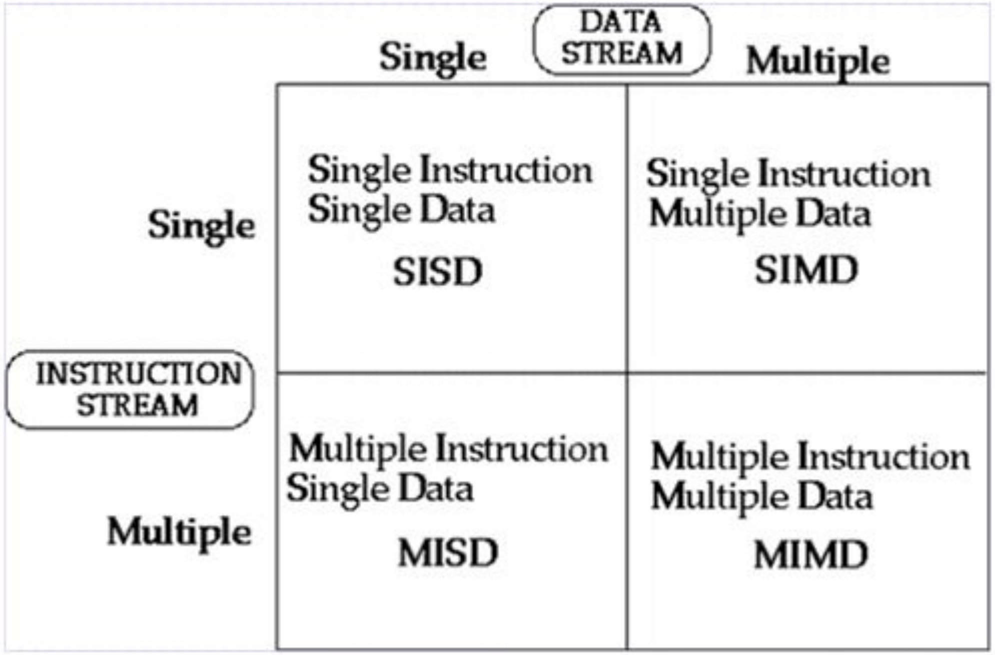
- Was proposed by researcher Michael J. Flynn in 1966.
- It is the most commonly accepted taxonomy of computer organization.
- In this classification, computers are classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets.

Flynn’s Classification

- This taxonomy distinguishes multi-processor computer architectures according to the two independent dimensions of Instruction stream and Data stream.
- An instruction stream is sequence of instructions executed by machine.
- A data stream is a sequence of data including input, partial or temporary results used by instruction stream.
- Each of these dimensions can have only one of two possible states: Single or Multiple.
- Flynn’s classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections.

Flynn's Classification

- Four category of Flynn classification

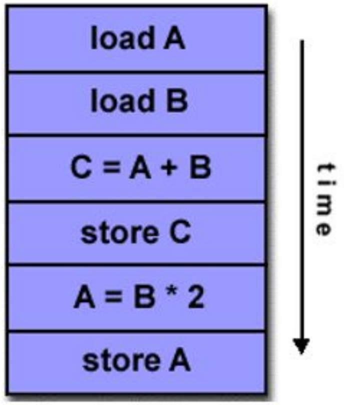


		DATA STREAM	
		Single	Multiple
INSTRUCTION STREAM	Single	Single Instruction Single Data SISD	Single Instruction Multiple Data SIMD
	Multiple	Multiple Instruction Single Data MISD	Multiple Instruction Multiple Data MIMD

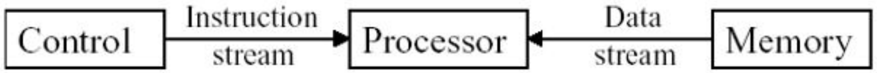
SISD

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.
- Deterministic execution.
- Instructions are executed sequentially.

- SISD computer having one control unit, one processor unit and single memory unit.
-



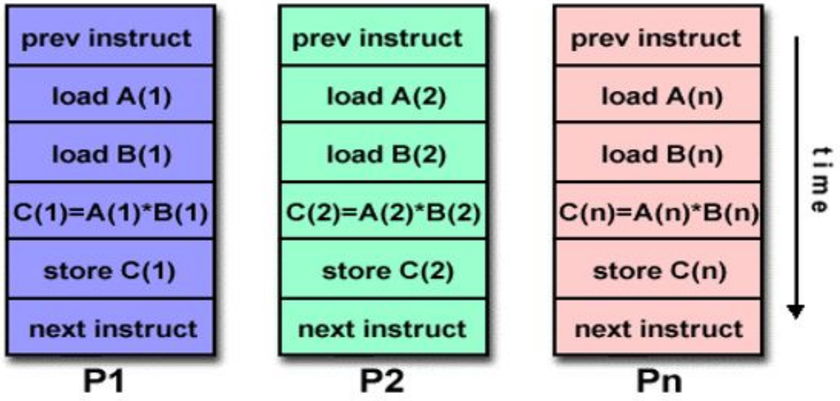
load A
load B
C = A + B
store C
A = B * 2
store A

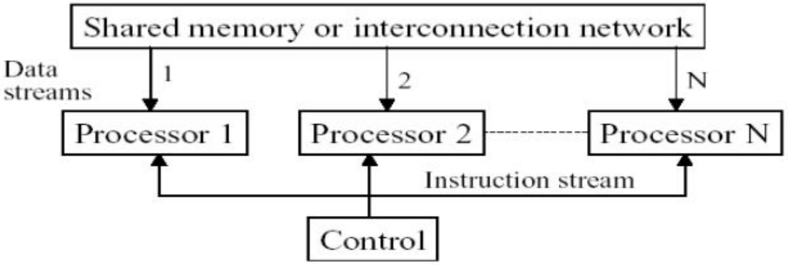


Control
Instruction stream
Processor
Data stream
Memory

SIMD

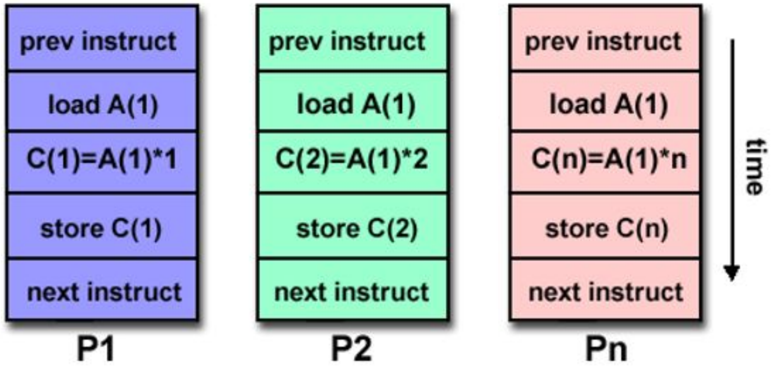
- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle .
- Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit
- single instruction is executed by different processing unit on different set of data

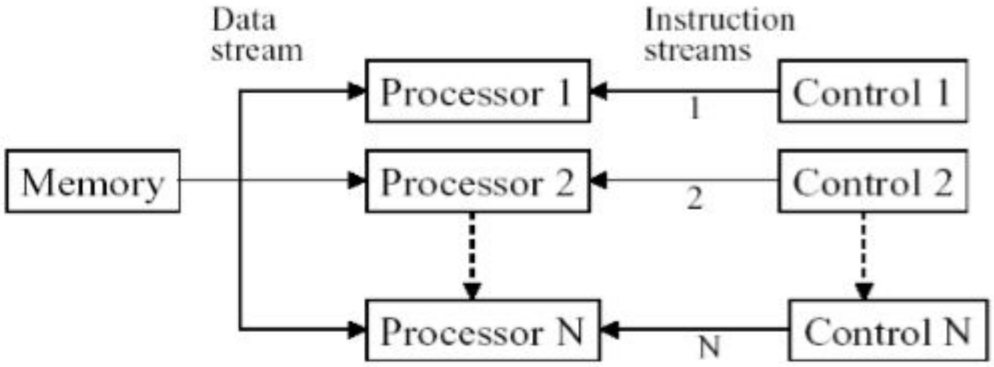




MISD

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.
- same data flow through a linear array of processors executing different instruction streams

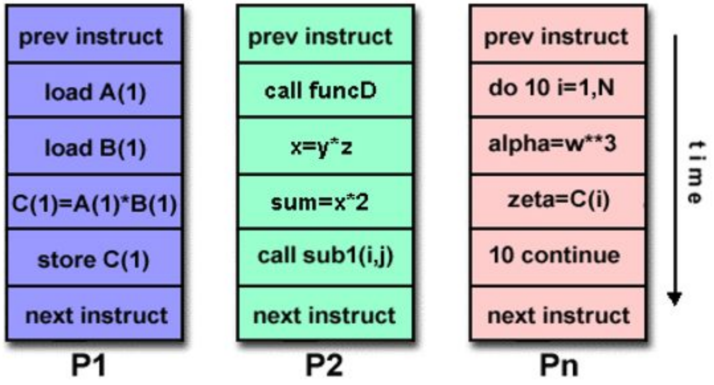




MIMD

- Multiple Instruction: every processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream.
- Execution can be synchronous asynchronous, deterministic nondeterministic

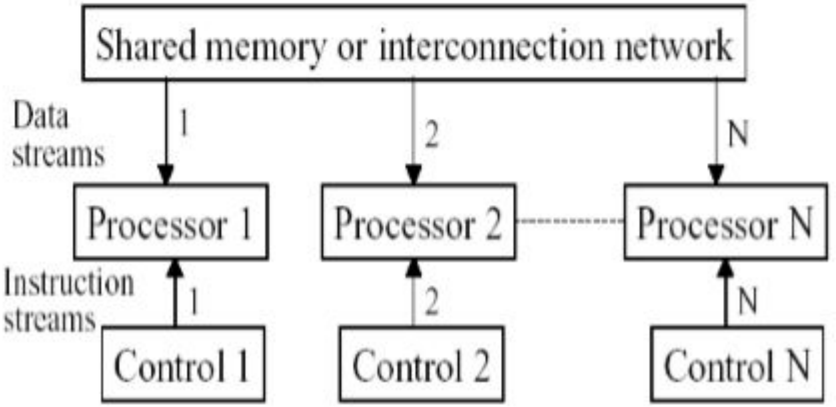
- Different processor each processing different task.

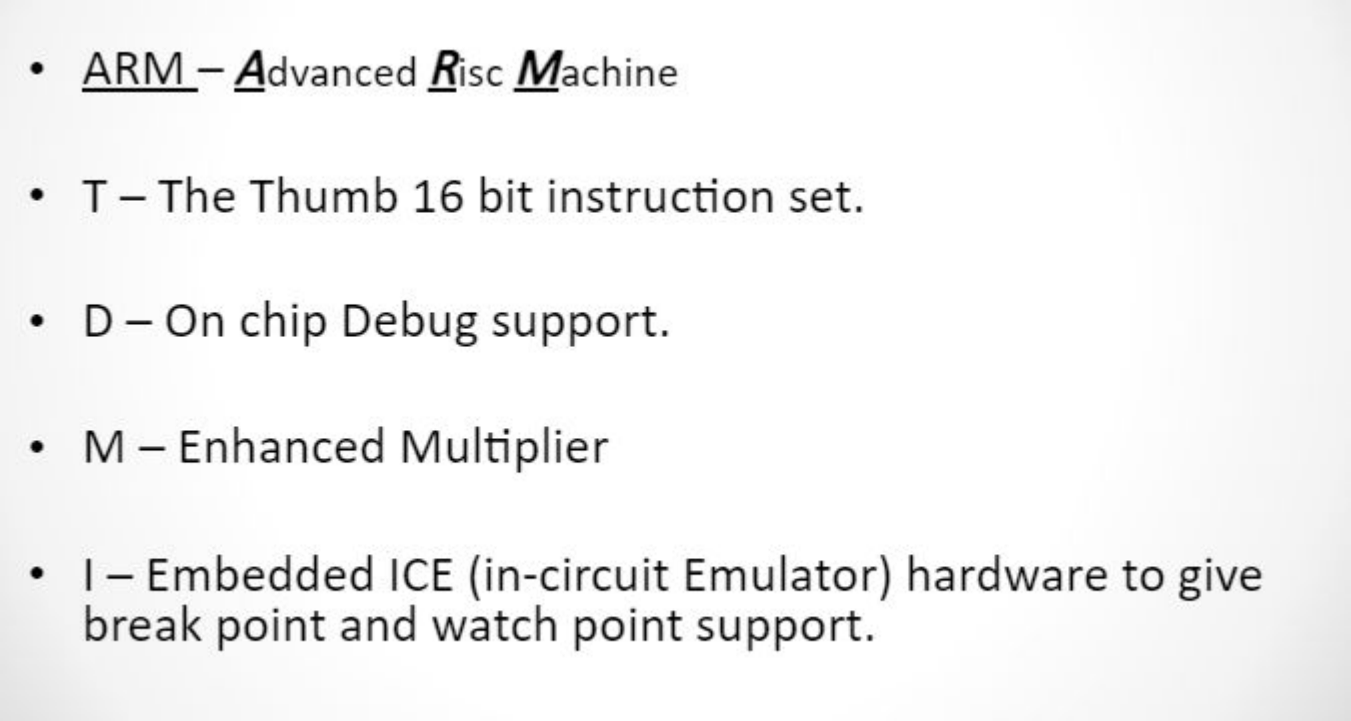


P1: prev instruct, load A(1), load B(1), C(1)=A(1)*B(1), store C(1), next instruct

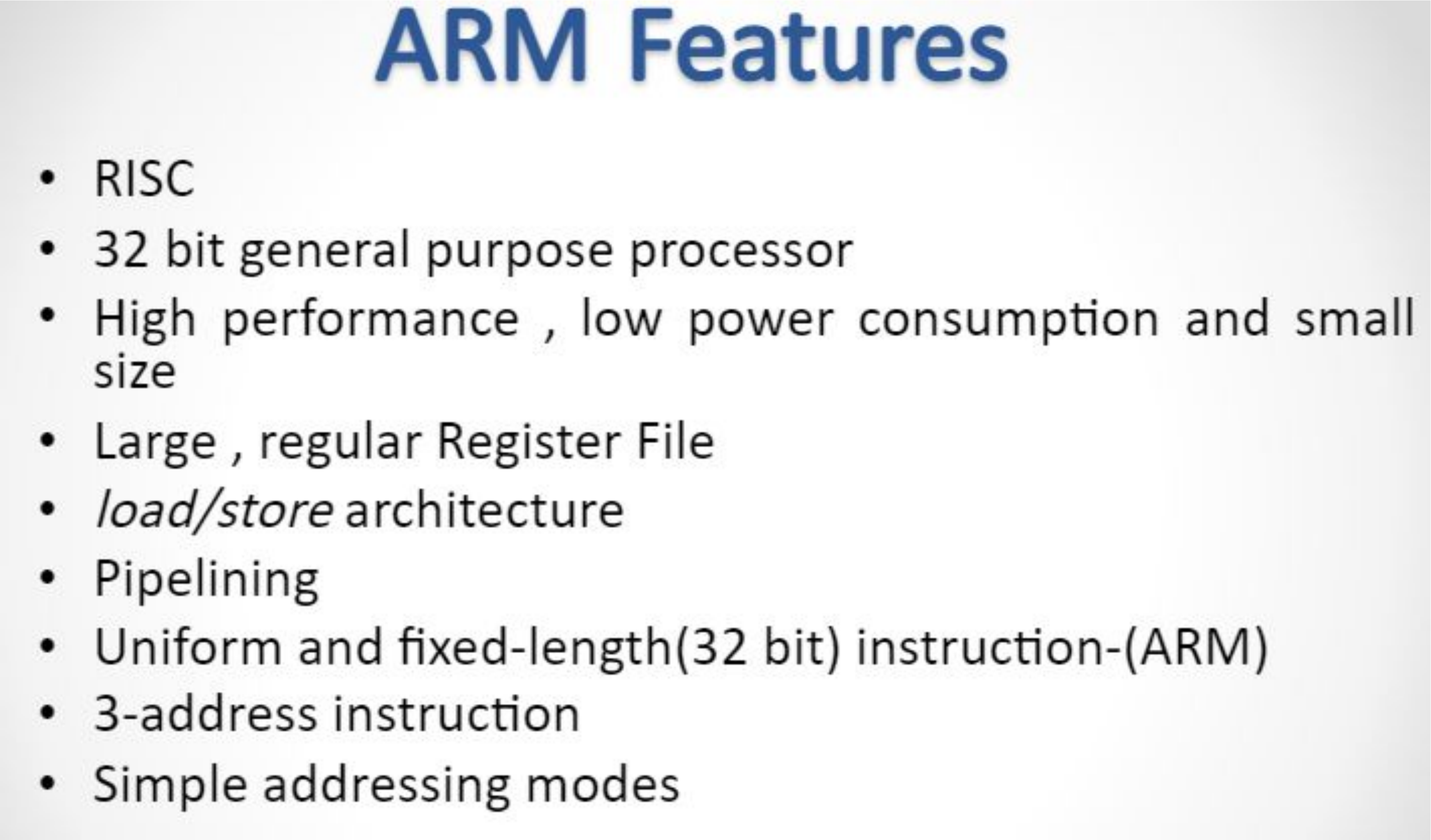
P2: prev instruct, call funcD, x=y*z, sum=x*2, call sub1(i,j), next instruct

Pn: prev instruct, do 10 i=1,N, alpha=w**3, zeta=C(i), 10 continue, next instruct





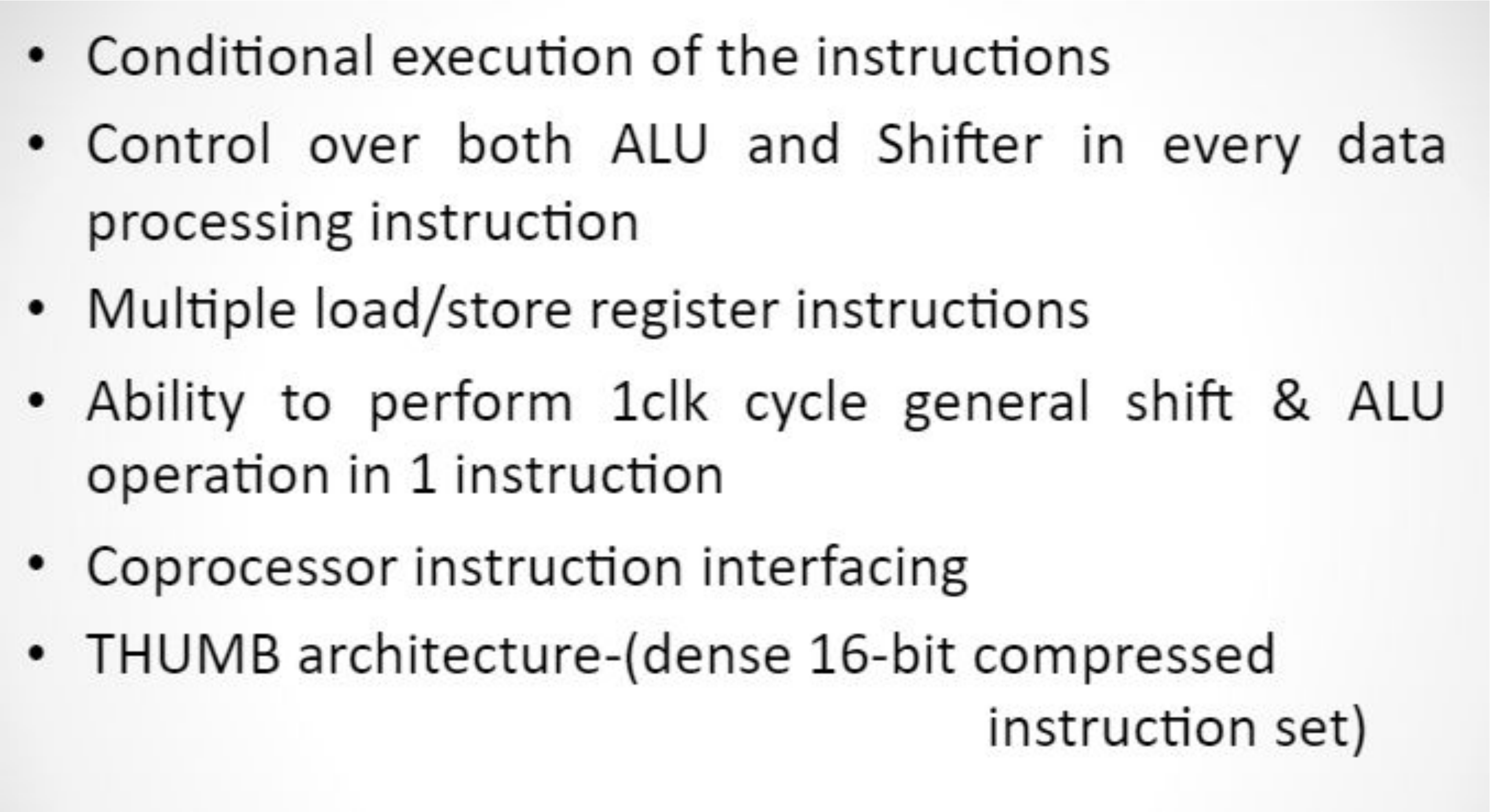
- ARM – ***A***dvanced ***R***isc ***M***achine
- T – The Thumb 16 bit instruction set.
- D – On chip Debug support.
- M – Enhanced Multiplier
- I – Embedded ICE (in-circuit Emulator) hardware to give break point and watch point support.



ARM Features

- RISC
- 32 bit general purpose processor
- High performance , low power consumption and small size
- Large , regular Register File
- *load/store* architecture
- Pipelining
- Uniform and fixed-length(32 bit) instruction-(ARM)
- 3-address instruction
- Simple addressing modes

ARM Features Contd…

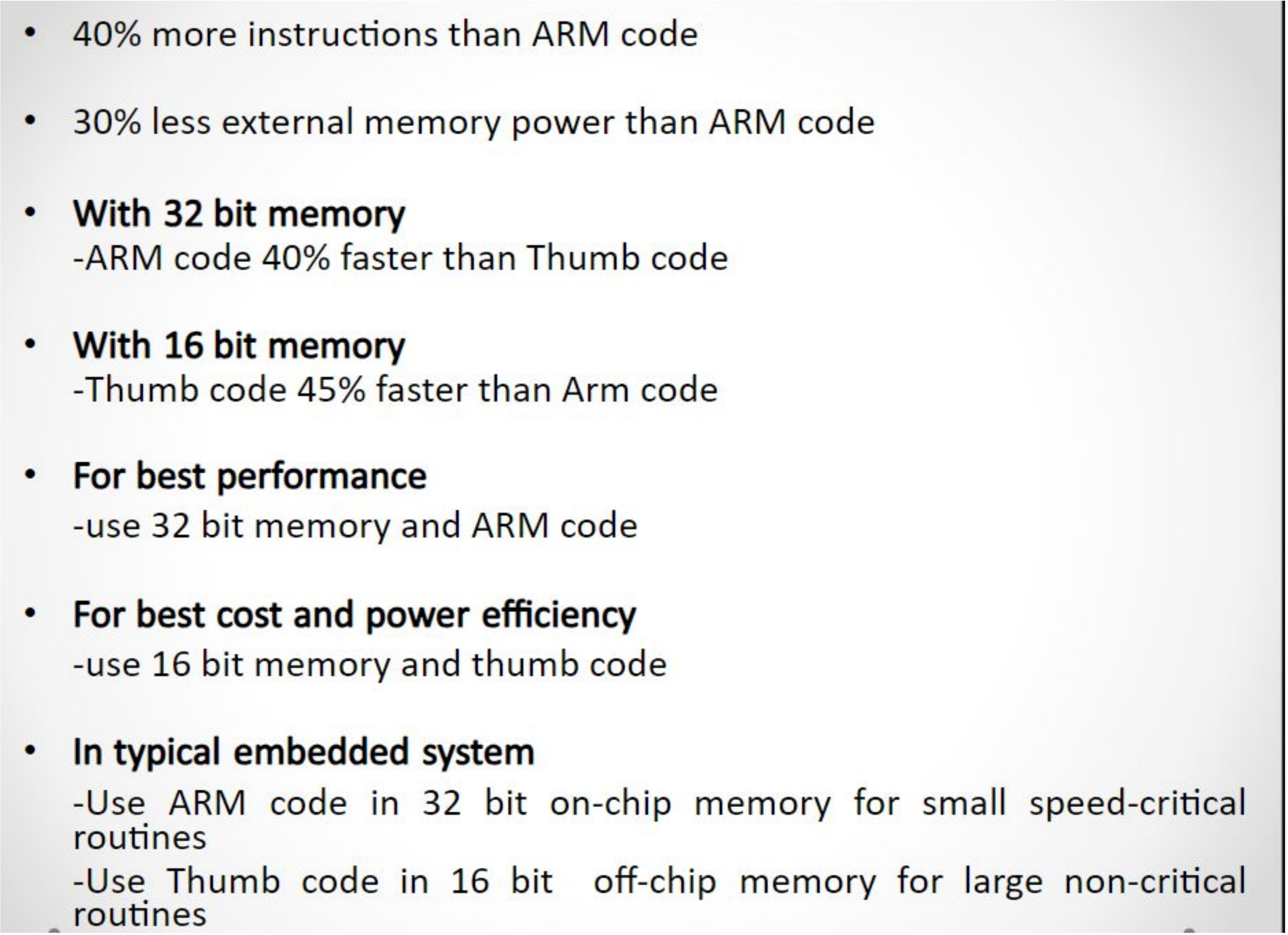


- Conditional execution of the instructions
- Control over both ALU and Shifter in every data processing instruction
- Multiple load/store register instructions
- Ability to perform 1clk cycle general shift & ALU operation in 1 instruction
- Coprocessor instruction interfacing
- THUMB architecture-(dense 16-bit compressed instruction set)

THUMB Instruction Set (T variant)

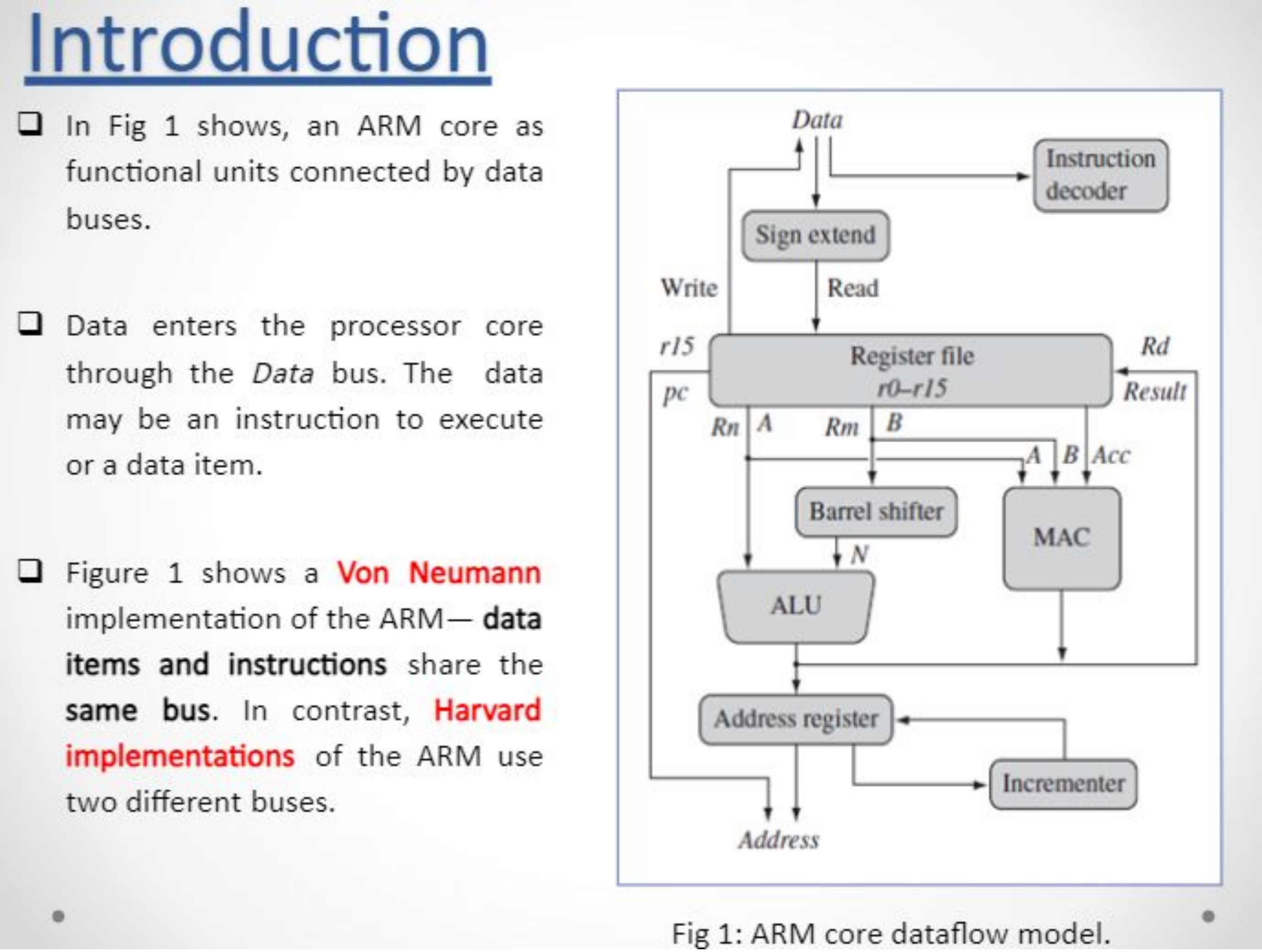
- re-encoded subset of ARM instruction
- Half the size of ARM instructions(16 bit)
- Greater code density
- On execution 16 bit thumb transparently decompressed to full 32 bit ARM without loss of performance
- Has all the advantages of 32 bit core
- Low performance in time-critical code
- Doesn’t include some instruction needed for exception handling

Thumb instruction set (T variant) Contd….



- 40% more instructions than ARM code
- 30% less external memory power than ARM code
- **With 32 bit memory**
  -ARM code 40% faster than Thumb code
- **With 16 bit memory**
  -Thumb code 45% faster than Arm code
- **For best performance**
  -use 32 bit memory and ARM code
- **For best cost and power efficiency**
  -use 16 bit memory and thumb code
- **In typical embedded system**
  -Use ARM code in 32 bit on-chip memory for small speed-critical routines
  -Use Thumb code in 16 bit off-chip memory for large non-critical routines

ARM Core dataflow model



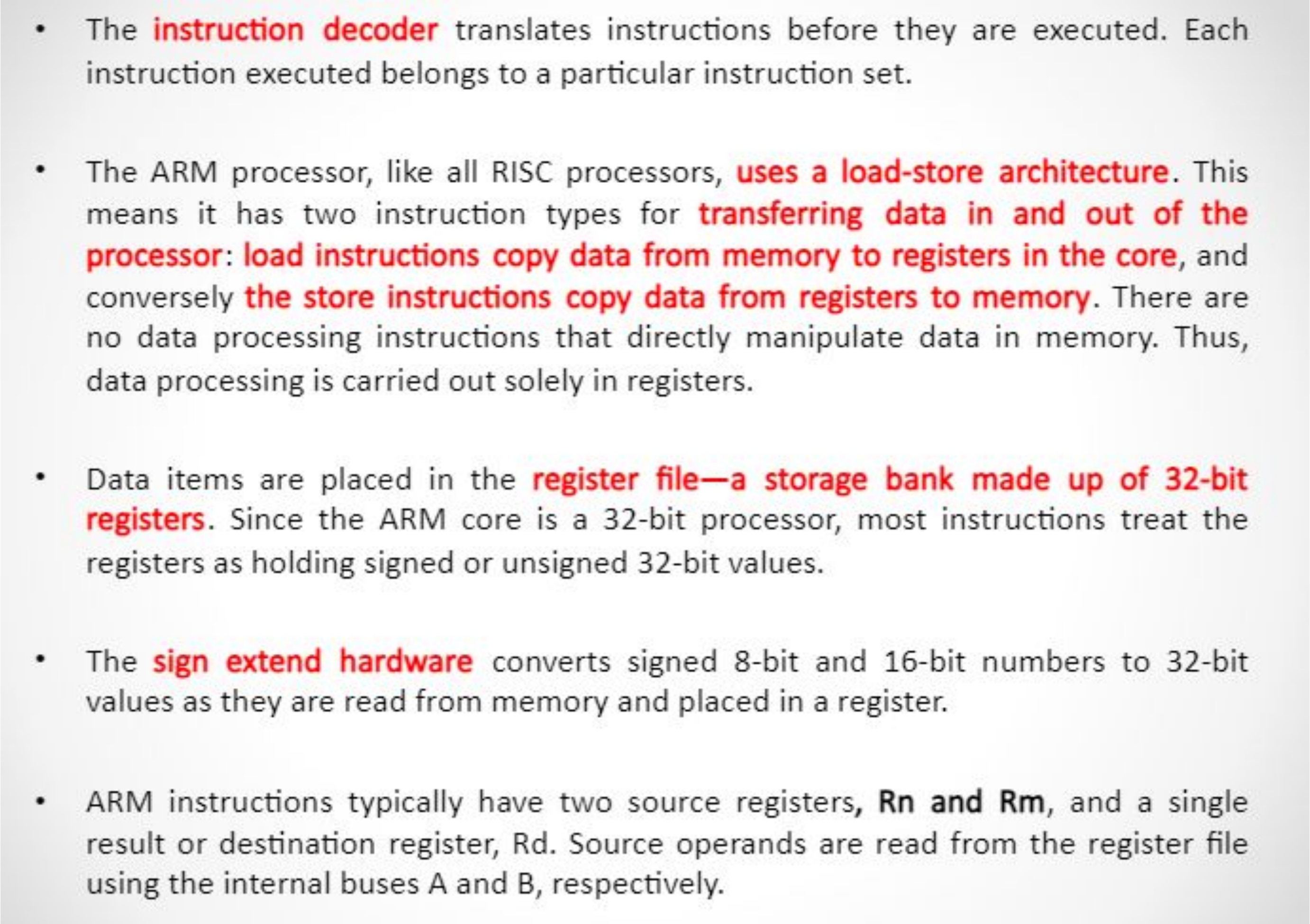
Introduction

- ❑ In Fig 1 shows, an ARM core as functional units connected by data buses.

- ❑ Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item.

- ❑ Figure 1 shows a **Von Neumann** implementation of the ARM— **data items and instructions** share the **same bus**. In contrast, **Harvard implementations** of the ARM use two different buses.

Fig 1: ARM core dataflow model.



- The **instruction decoder** translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

- The ARM processor, like all RISC processors, **uses a load-store architecture**. This means it has two instruction types for **transferring data in and out of the processor**: **load instructions copy data from memory to registers in the core**, and conversely **the store instructions copy data from registers to memory**. There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

- Data items are placed in the **register file—a storage bank made up of 32-bit registers**. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values.

- The **sign extend hardware** converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

- ARM instructions typically have two source registers**, Rn and Rm**, and a single result or destination register, Rd. Source operands are read from the register file using the internal buses A and B, respectively.

- The **ALU (arithmetic logic unit) or MAC (multiply-accumulate unit)** takes the register values ***Rn*** and ***Rm*** from the *A* and *B* buses and computes a result. Data processing instructions write the **result in *Rd*** directly to the register file. Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.

- **One important feature** of the ARM is that register **Rm** alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses. After passing through the functional units, the result in Rd is written back to the register file using the Result bus.

- For load and store instructions the **incremented** updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

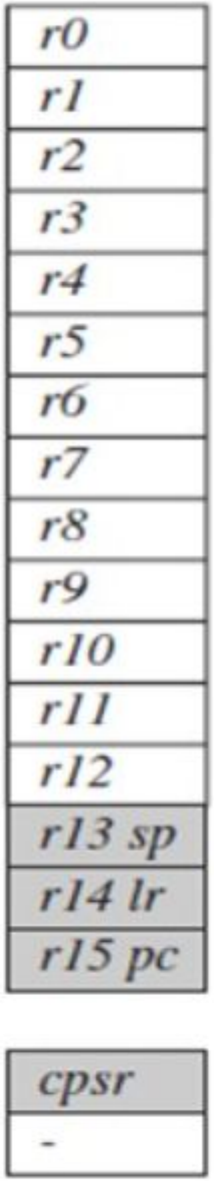
- The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

ARM Registers

- General-purpose registers hold either data or an address.
- Fig 2 shows the active registers available in user mode—a protected mode normally used when executing applications.

- The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

- In Fig 2, the shaded registers identify the assigned special-purpose registers:
- **Register *r13*** is traditionally used as the **stack pointer (sp)** and stores the head of the stack in the current processor mode.
- **Register r14** is called the **link register (*lr*)** and is where the core puts the return address whenever it calls a subroutine.
- **Register *r15*** is the **program counter (pc)** and contains the address of the next instruction to be fetched by the processor.



r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 sp
r14 lr
r15 pc

cpsr
-

Fig 2: ARM Register in ***user mode***.

- The processor can operate in seven different modes. All the registers shown are 32 bits in size.

- There are up to **18 active registers**: **16 data registers** and **2 processor status registers**. The data registers are visible to the programmer as ***r0*** to ***r15***.

- Depending upon the context, registers r13 and r14 can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change.

- In ARM state the registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers. However, there are instructions that treat r14 and r15 in a special way.

- In addition to the 16 data registers, there are two program status registers: ***cpsr*** and ***spsr*** (the **current and saved program status registers**, respectively).

Processor Modes

- First 5 bits are signifies as mode selection

- The processor mode determines which registers are active and the access rights to the ***cpsr*** register itself.

- Each processor mode is either **privileged** or **nonprivileged:** A **privileged** mode allows full read-write access to the **cpsr**. Conversely, a **nonprivileged** mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

- There are seven processor modes in total: **six privileged modes** (**abort**, **fast interrupt request**, **interrupt request**, **supervisor**, **system**, and **undefined**) and one **nonprivileged** mode (**user**).

- ***Abort modes:***
  - The processor enters *abort* mode when there is **a failed attempt to access memory.**

- ***Fast interrupt request* and *interrupt request* modes:**
  - These two modes correspond to the two interrupt levels available on the ARM processor.

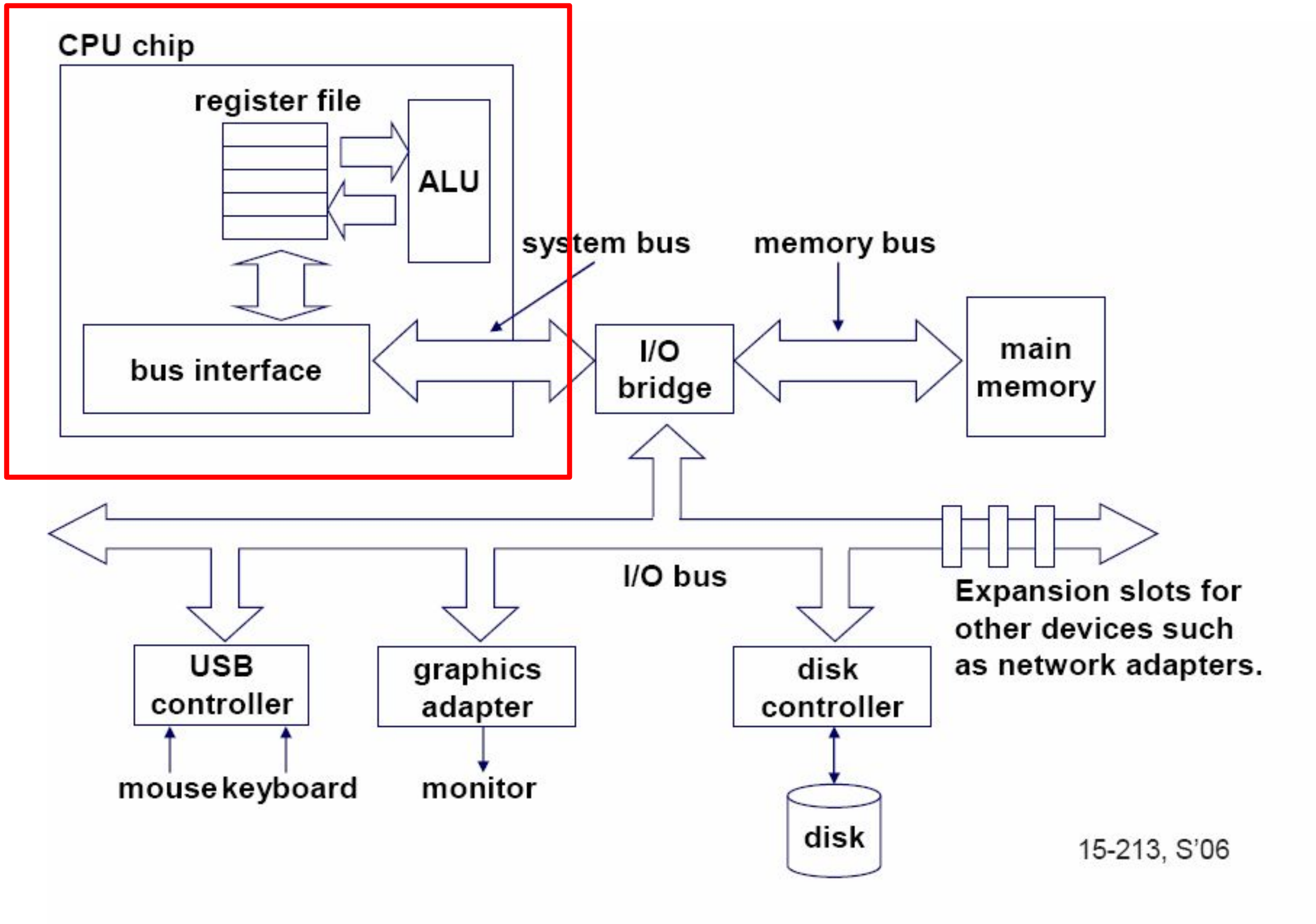
- ***Supervisor mode***
  - It is the mode that **the processor is in after reset** and is generally the mode that an operating system kernel operates in.

- ***System* mode**
  - It is a special version of user mode that allows **full read-write access** to the cpsr.

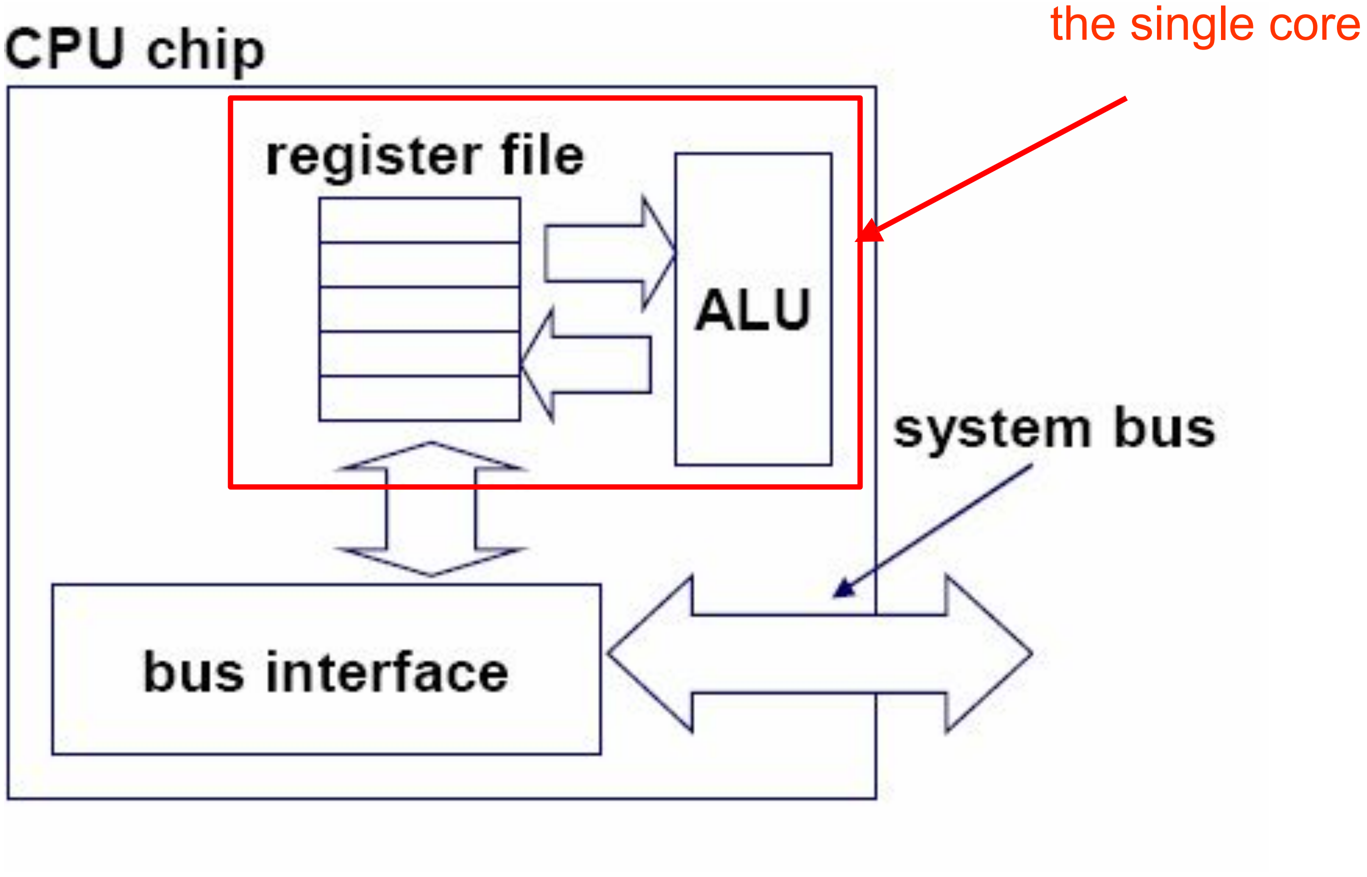
- ***Undefined* mode**
  - is used when the processor encounters an **instruction that is undefined** or not supported by the implementation.

- ***User mode***
  - is used for programs and applications.

Single-core computer

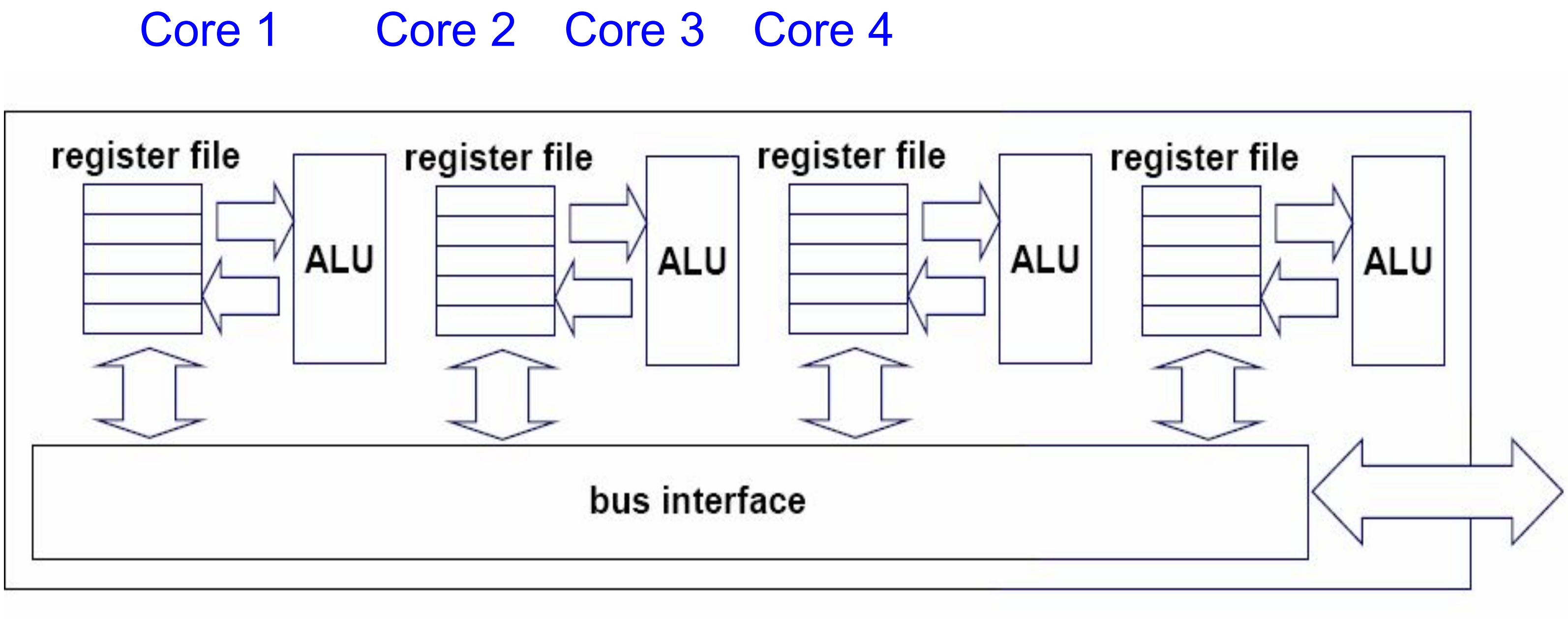


Single-core CPU chip



Multi-core architectures

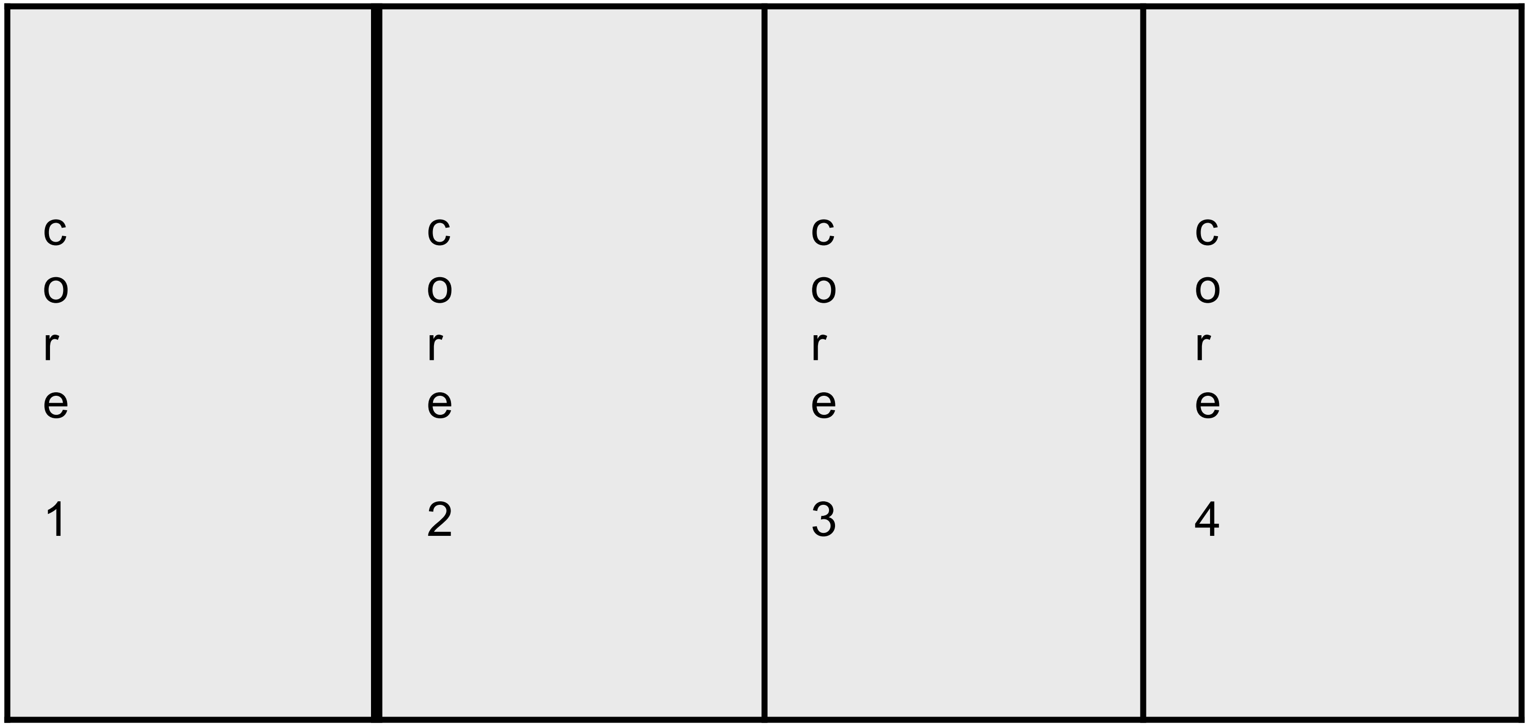
- Replicate multiple processor cores on a single die.



Multi-core CPU chip

Multi-core CPU chip

- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



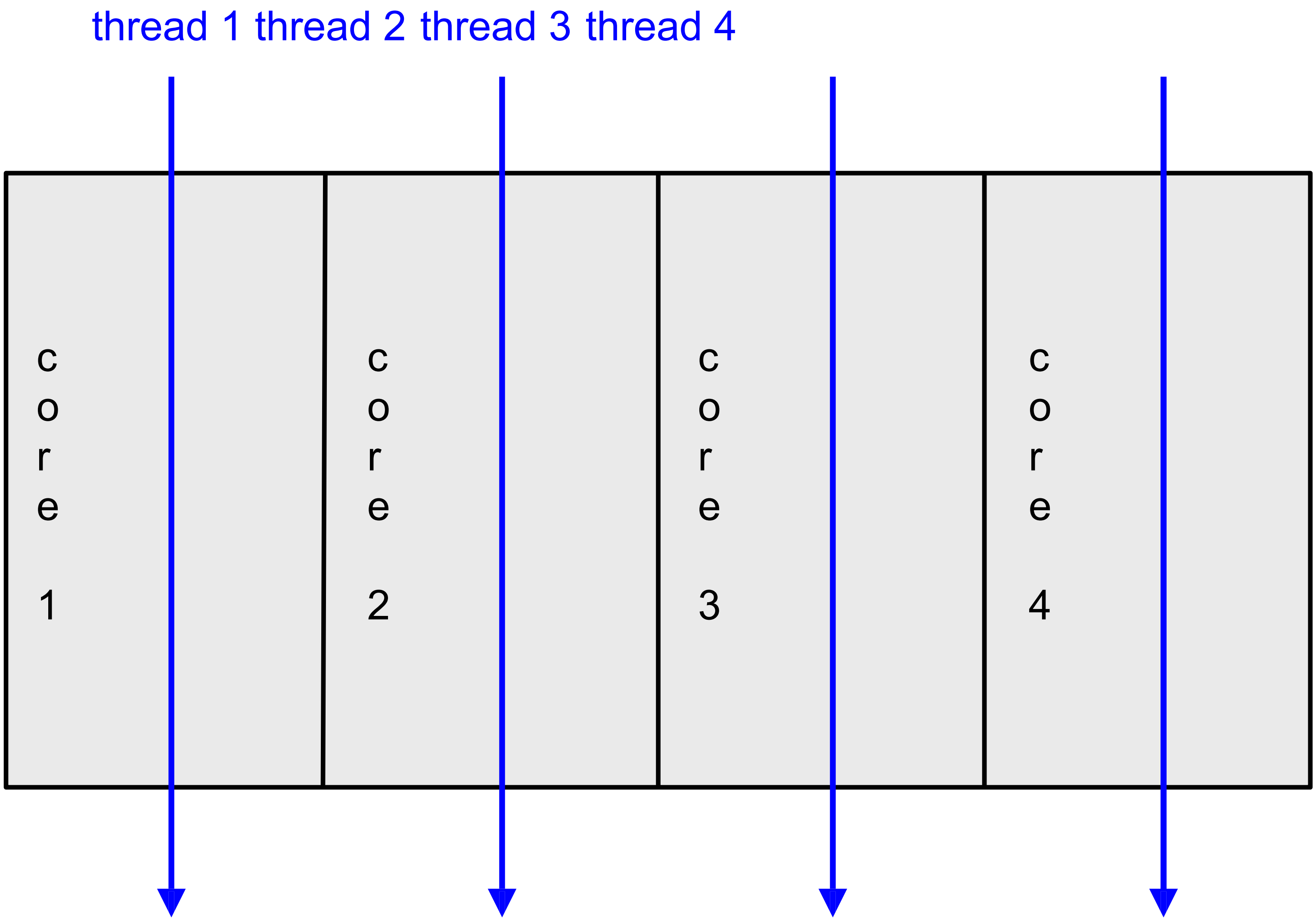
core 1

core 2

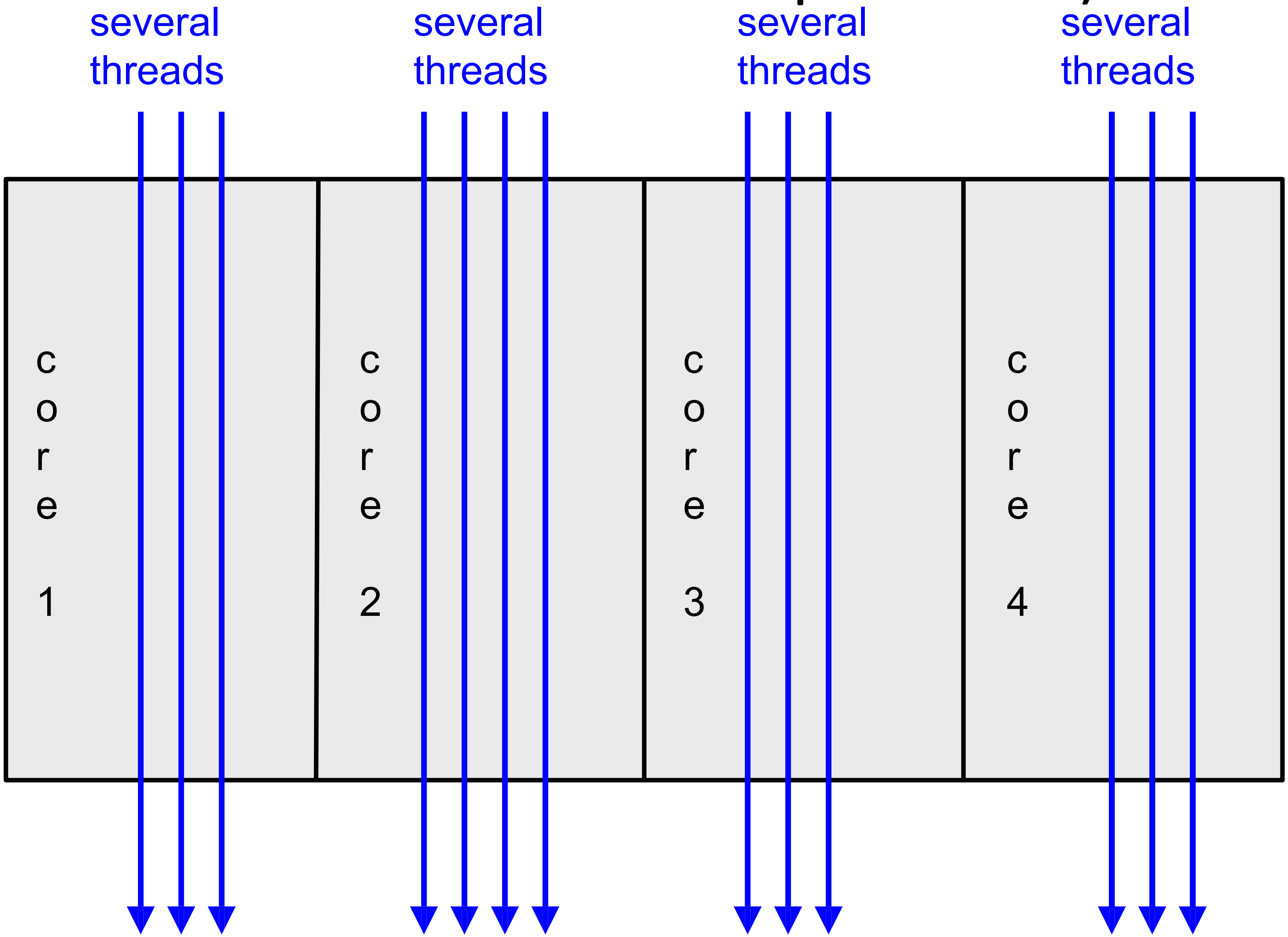
core 3

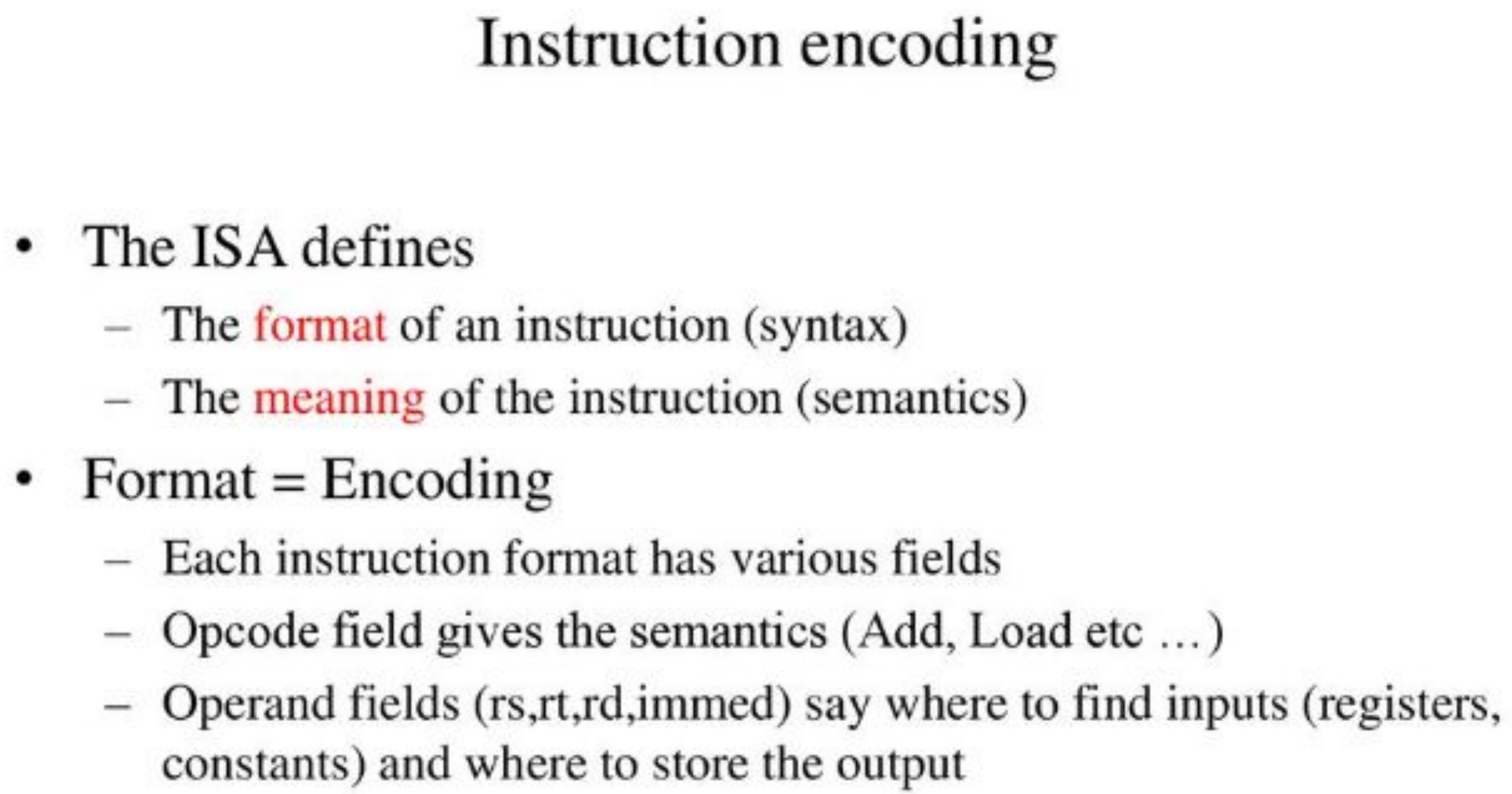
core 4

The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)





Instruction encoding

- The ISA defines
  - The format of an instruction (syntax)
  - The meaning of the instruction (semantics)
- Format = Encoding
  - Each instruction format has various fields
  - Opcode field gives the semantics (Add, Load etc …)
  - Operand fields (rs,rt,rd,immed) say where to find inputs (registers, constants) and where to store the output

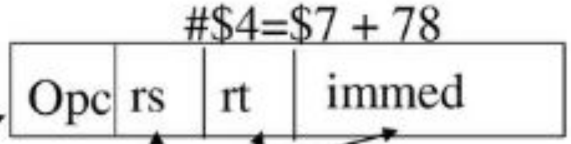
MIPS Instruction encoding

- MIPS = RISC hence
  - Few (3+) instruction formats
- R in RISC also stands for “Regular”
  - All instructions of the same length (32-bits = 4 bytes)
  - Formats are consistent with each other
    - Opcode always at the same place (6 most significant bits)
    - rd and rs always at the same place
    - immed always at the same place etc.

I-type (immediate) format

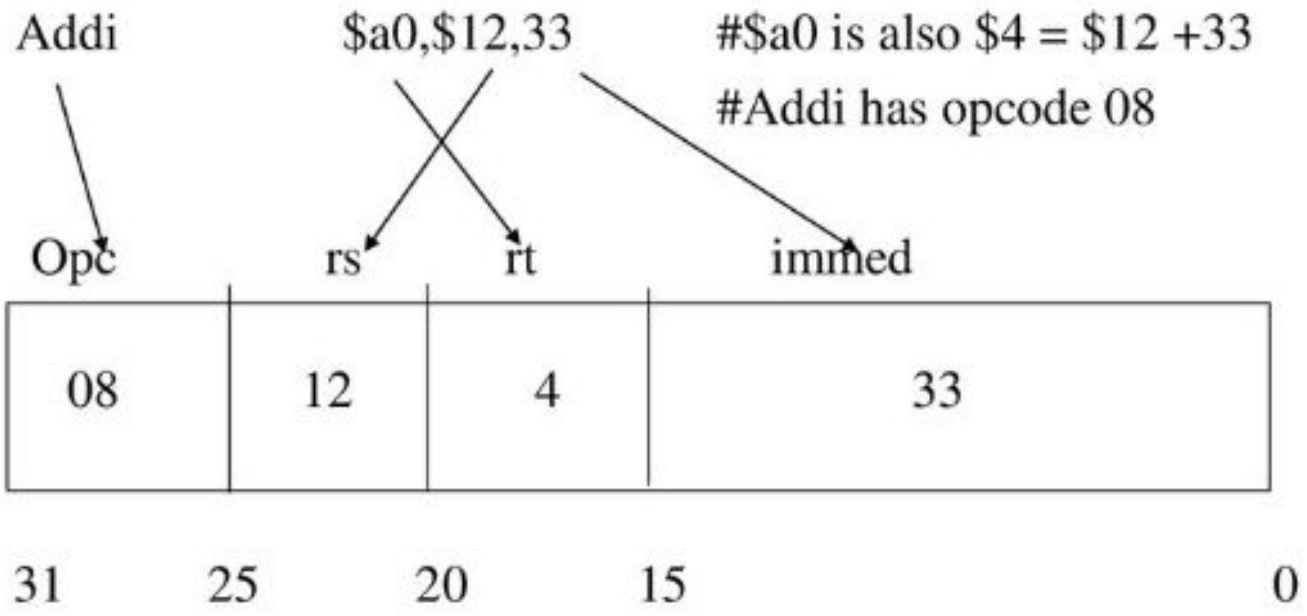
- An instruction with an immediate constant has the SPIM form:

Opcode	Operands	Comment
Addi	$4,$7,78	#$4=$7 + 78



- Encoding of the 32 bits:
  - Opcode is 6 bits
  - Since we have 32 registers, each register "name" is 5 bits
  - This leaves 16 bits for the immediate constant

I-type format example



Addi $a0,$12,33 #$a0 is also $4 = $12 +33

#Addi has opcode 08

Opc	rs	rt	immed
08	12	4	33

31 25 20 15 0

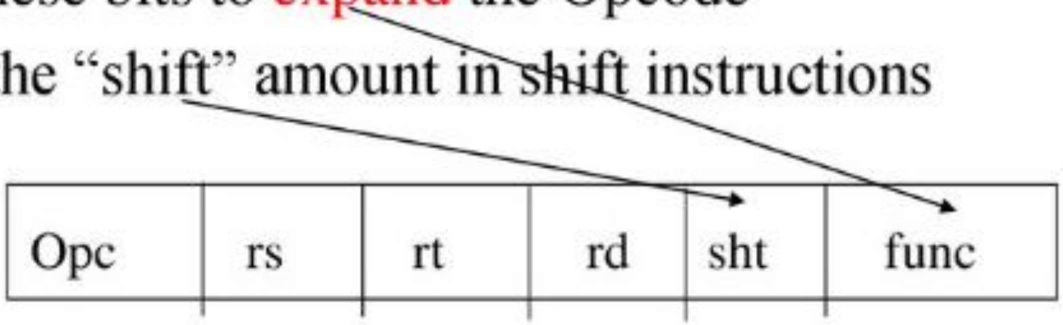
In hex: 21840021

Sign extension

- Internally the ALU (adder) deals with 32-bit numbers
- What happens to the 16-bit constant?
  - Extended to 32 bits
- If the Opcode says “unsigned” (e.g., Addiu)
  - Fill upper 16 bits with 0’s
- If the Opcode says “signed” (e.g., Addi)
  - Fill upper 16 bits with the msb of the 16 bit constant
    - i.e. fill with 0’s if the number is positive
    - i.e. fill with 1’s if the number is negative

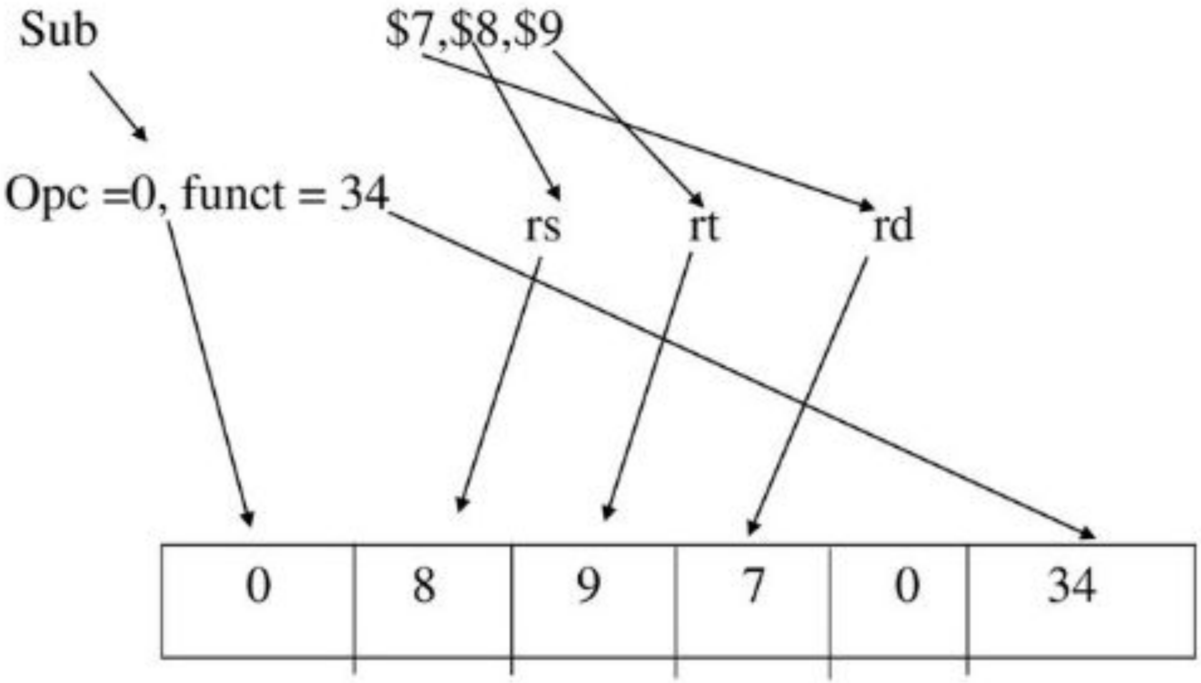
R-type (register) format

- Arithmetic, Logical, and Compare instructions require encoding 3 registers.
- Opcode (6 bits) + 3 registers (5.3 =15 bits) => 32 -21 = 11 "free" bits
- Use 6 of these bits to expand the Opcode
- Use 5 for the "shift" amount in shift instructions



Opc	rs	rt	rd	sht	func

R-type example

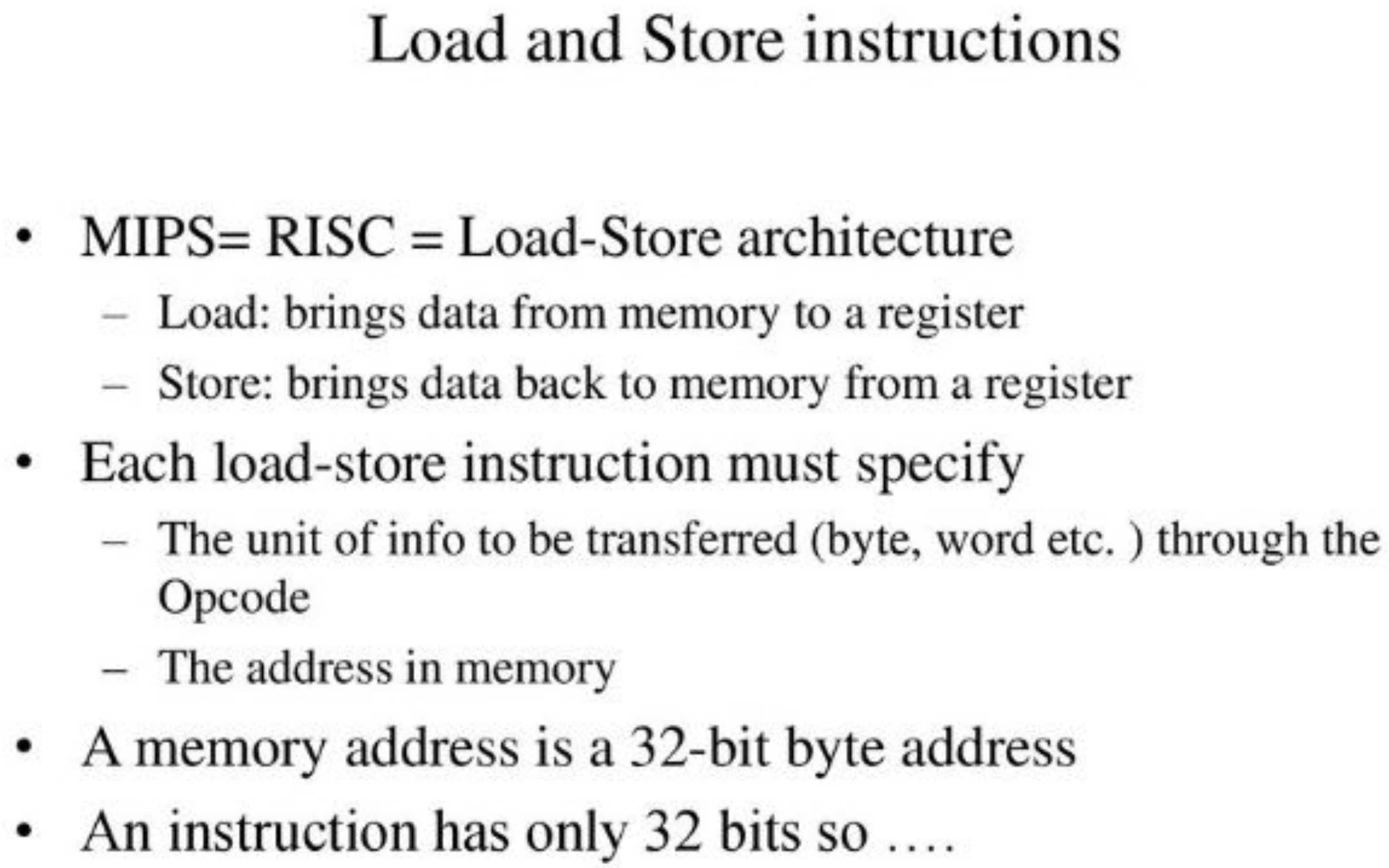


Sub $7,$8,$9

Opc =0, funct = 34

rs rt rd

0	8	9	7	0	34



Load and Store instructions

- MIPS= RISC = Load-Store architecture
  - Load: brings data from memory to a register
  - Store: brings data back to memory from a register
- Each load-store instruction must specify
  - The unit of info to be transferred (byte, word etc.) through the Opcode
  - The address in memory
- A memory address is a 32-bit byte address
- An instruction has only 32 bits so ….

Addressing in Load/Store instructions

- The address will be the sum
  - of a *base* register (register rs)
  - a 16-bit *offset* (or displacement) which will be in the immed field and is added (as a signed number) to the contents of the base register
- Thus, one can address any byte within ± 32KB of the address pointed to by the contents of the base register.

Examples of load-store instructions

- Load word from memory:

```
Lw      rt,rs,offset            #rt = Memory[rs+offset]
```

- Store word to memory:

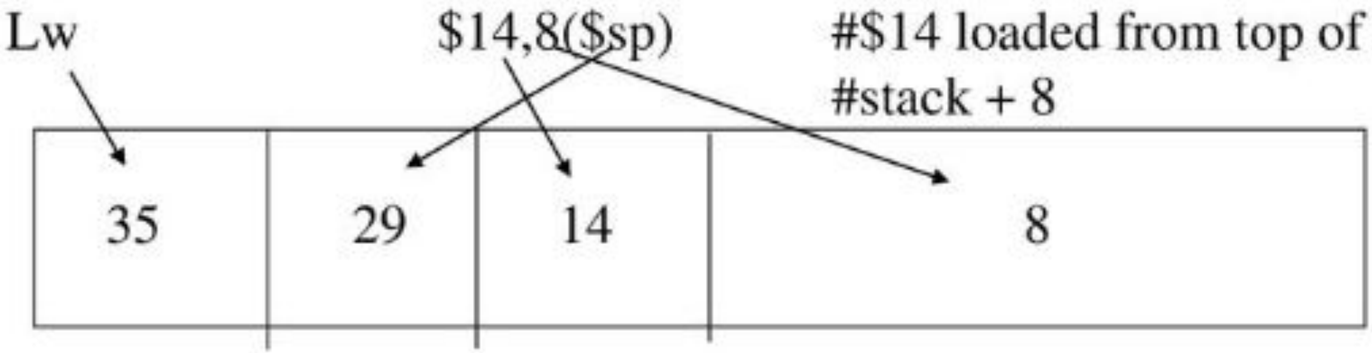
```
Sw      rt,rs,offset            #Memory[rs+offset]=rt
```

- For bytes (or half-words) only the lower byte (or half-word) of a register is addressable
  - For load you need to specify if it is sign-extended or not

```
Lb   rt,rs,offset               #rt =sign-ext( Memory[rs+offset])
Lbu  rt,rs,offset               #rt =zero-ext( Memory[rs+offset])
Sb   rt,rs,offset               #Memory[rs+offset]= least signif.
                                #byte of rt
```

Load-Store format

- Need for
  - Opcode (6 bits)
  - Register destination (for Load) and source (for Store) : rt
  - Base register: rs
  - Offset (immed field)
- Example



Lw $14,8($sp) #$14 loaded from top of #stack + 8

35	29	14	8

Loading small constants in a register

- If the constant is small (i.e., can be encoded in 16 bits) use the immediate format with Li (Load immediate)

```
Li        $14,8        #$14 = 8
```

- But, there is no opcode for Li!
- Li is a *pseudoinstruction*
  - It's there to help you
  - SPIM will recognize it and transform it into Addi (with sign-extension)

```
Addi      $14,$0,8        #$14 = $0+8
```

Loading large constants in a register

- If the constant does not fit in 16 bits (e.g., an address)
- Use a two-step process
  - Lui (load upper immediate) to load the upper 16 bits; it will zero out automatically the lower 16 bits
  - Use Ori for the lower 16 bits (but not Li, why?)
- Example: Load the constant 0x1B234567 in register $t0

```
Lui        $t0,0x1B23        #note the use of hex constants
  Ori      $t0,$t0,0x4567
```

How to address memory in assembly language

- Problem: how do I put the base address in the right register and how do I compute the offset
- Method 1 (most recommended). Let the assembler do it!

```
                .data               #define data section
xyz:            .word    1          #reserve room for 1 word at address xyz
                ........            #more data
                .text               #define program section
                 .....              # some lines of code
                lw    $5, xyz       # load contents of word at add. xyz in $5
```

- In fact the assembler generates:

```
      Lw      $5, offset ($gp)         #$gp is register 28
```

Generating addresses

- Method 2. Use the pseudo-instruction La (Load address)

```
La   $6,xyz          #$6 contains address of xyz
Lw   $5,0($6)        #$5 contains the contents of xyz
```

  - La is in fact Lui followed by Addi
  - This method can be useful to traverse an array after loading the base address in a register
- Method 3
  - If you know the address (i.e. a constant) use Li or Lui + Addi

Difference between Memory mapped I/O and I/O mapped I/O

	Memory Mapped Input/Output	Input/Output Mapped Input/Output
1.	Each port is treated as a memory location.	Each port is treated as an independent unit.
2.	CPU’s memory address space is divided between memory and input/output ports.	Separate address spaces for memory and input/output ports.
3.	Single instruction can transfer data between memory and port.	Two instruction are necessary to transfer data between memory and port.
4.	Data transfer is by means of instruction like MOVE.	Each port can be accessed by means of IN or OUT instructions.

Program Controlled I/O

- Program controlled I/O is one in which the processor repeatedly checks a status flag to achieve the required synchronization between processor & I/O device.
- The processor polls the device.
- It is useful in small low speed systems where hardware cost must be minimized.
- It requires that all input/output operators be executed under the direct control of the CPU.
- The transfer is between CPU registers(accumulator) and a buffer register connected to the input/output device.
- The i/o device does not have direct access to main memory.
- A data transfer from an input/output device to main memory requires the execution of several instructions by the CPU, including an input instruction to transfer a word from the input/output device to the CPU and a store instruction to transfer a word from CPU to main memory.
- One or more additional instructions may be needed for address communication and data word counting.

Typical Program controlled instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare	CMP
Test(by ADDing)	TST

Case study: ARM 5 and ARM 7 Architecture

Data Sizes and Instruction Sets

- **The ARM is a 32-bit architecture.**

- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

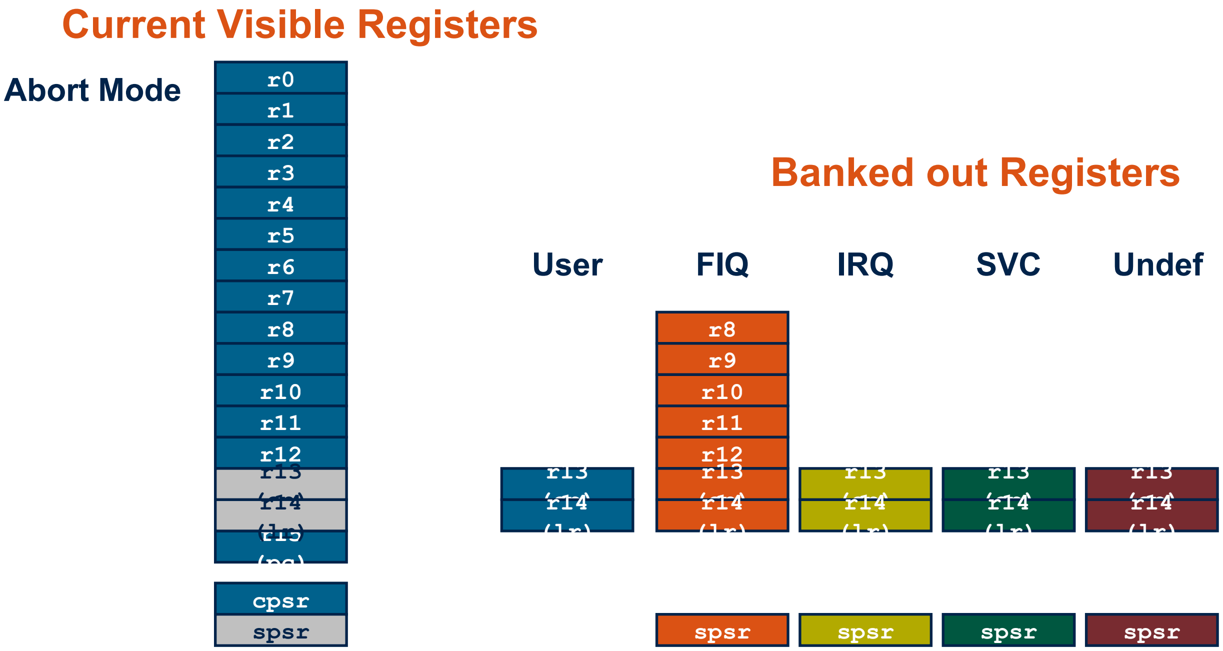
- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- **Jazelle cores can also execute Java bytecode**

Processor Modes

- **The ARM has seven basic operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

The ARM Register Set



Current Visible Registers

Abort Mode

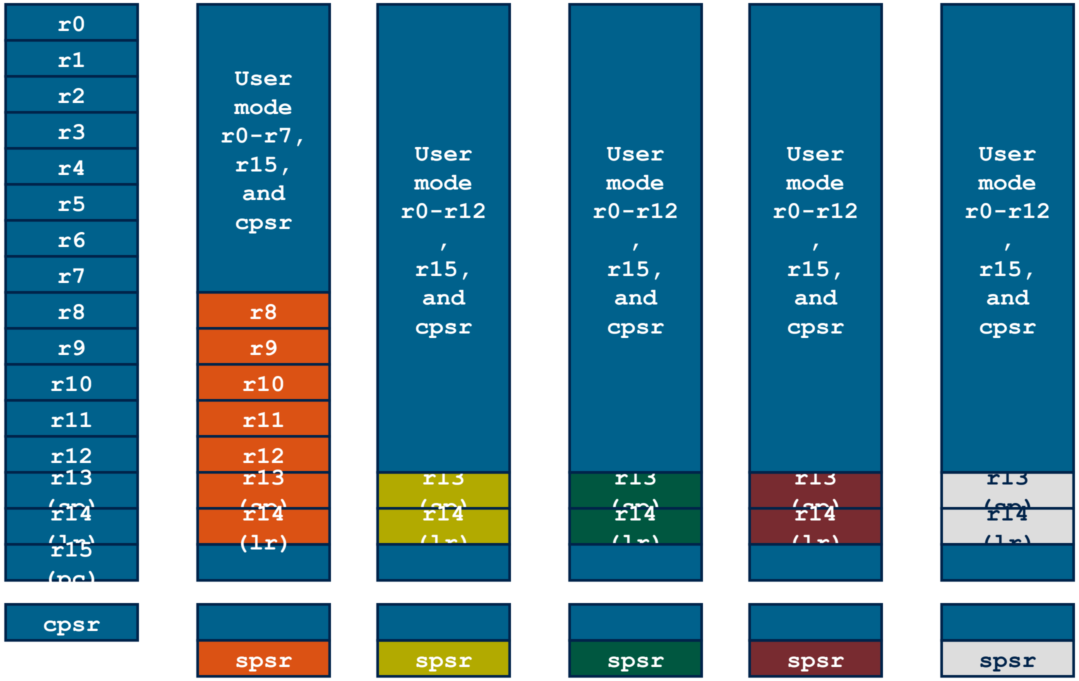
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Register Organization Summary



r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr

User mode r0-r7, r15, and cpsr
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

spsr

User mode r0-r12 , r15, and cpsr
r13 (sp)
r14 (lr)

spsr

User mode r0-r12 , r15, and cpsr
r13 (sp)
r14 (lr)

spsr

User mode r0-r12 , r15, and cpsr
r13 (sp)
r14 (lr)

spsr

User mode r0-r12 , r15, and cpsr
r13 (sp)
r14 (lr)

spsr

Thumb state
Low registers

Thumb state
High registers

Note: System mode uses the User mode register set

The Registers

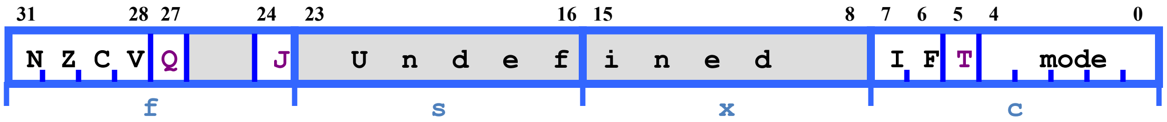
- **ARM has 37 registers all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- **The current processor mode governs which of several banks is accessible. Each mode can access**
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

Privileged modes (except System) can also access

- a particular spsr (saved program status register)

Program Status Registers


31 28 27 24 23 16 15 8 7 6 5 4 0

N Z C V Q J Undefined I F T mode

f s x c

- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed

- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred

- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.

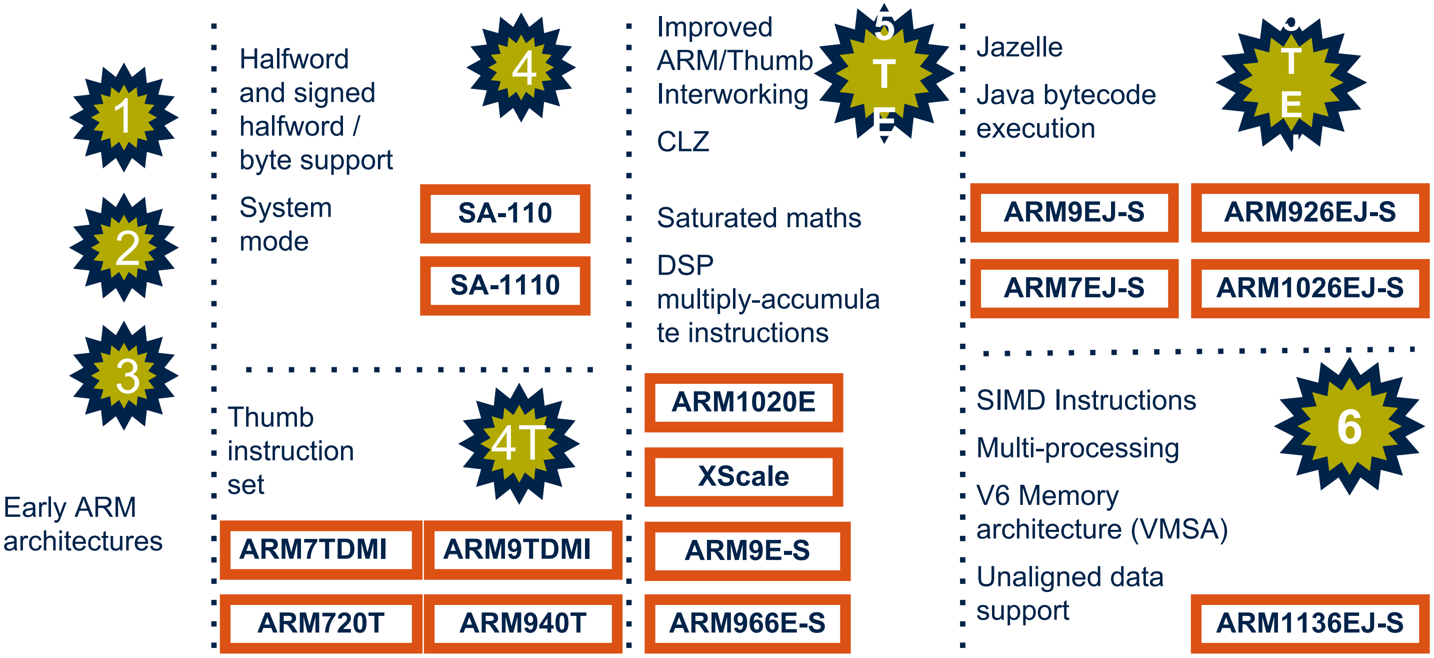
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state

- Mode bits
  - Specify the processor mode

Program Counter (r15)

- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).
- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).
- **When the processor is executing in Jazelle state:**
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Development of the ARM Architecture



1

2

3

Early ARM architectures

4

Halfword and signed halfword / byte support

System mode

SA-110

SA-1110

4T

Thumb instruction set

ARM7TDMI

ARM9TDMI

ARM720T

ARM940T

5
T
E

Improved ARM/Thumb Interworking

CLZ

Saturated maths

DSP multiply-accumulate instructions

ARM1020E

XScale

ARM9E-S

ARM966E-S

T
E

Jazelle

Java bytecode execution

ARM9EJ-S

ARM926EJ-S

ARM7EJ-S

ARM1026EJ-S

6

SIMD Instructions

Multi-processing

V6 Memory architecture (VMSA)

Unaligned data support

ARM1136EJ-S

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
CMP     r3,#0
   BEQ    skip
   ADD    r0,r1,r2
skip
```

```
CMP     r3,#0
   ADDNE  r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using “S”. CMP does not need “S”.

```
loop
   …
   SUBS  r1,r1,#1
   BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
        CMP         r0,#0
        MOVEQ       r0,#1
        BLEQ        func
```

- Set the flags, then use various condition codes

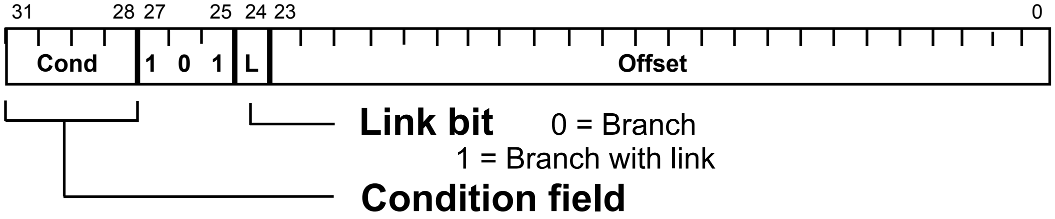
```
if (a==0) x=0;
if (a>0)  x=1;
        CMP         r0,#0
        MOVEQ       r1,#0
        MOVGT       r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
        CMP         r0,#4
        CMPNE       r0,#10
        MOVEQ       r1,#0
```

Branch instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Data processing Instructions

- Consist of :
  - Arithmetic: `ADD ADC SUB SBC RSB RSC`
  - Logical: `AND ORR EOR BIC`
  - Comparisons: `CMP CMN TST TEQ`
  - Data movement: `MOV MVN`

- These instructions only work on registers, NOT memory.

- Syntax:

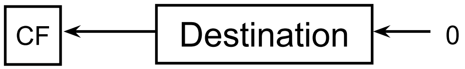
```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.

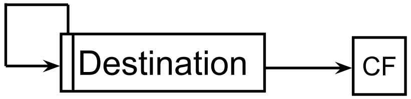
The Barrel Shifter

LSL : Logical Left Shift



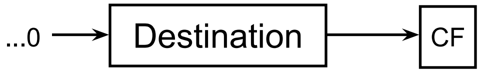
Multiplication by a power of 2

ASR: Arithmetic Right Shift



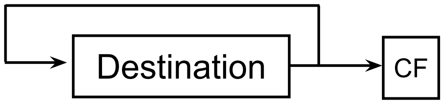
Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right



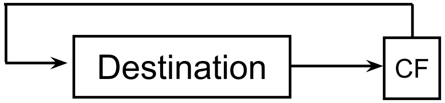
Division by a power of 2

ROR: Rotate Right



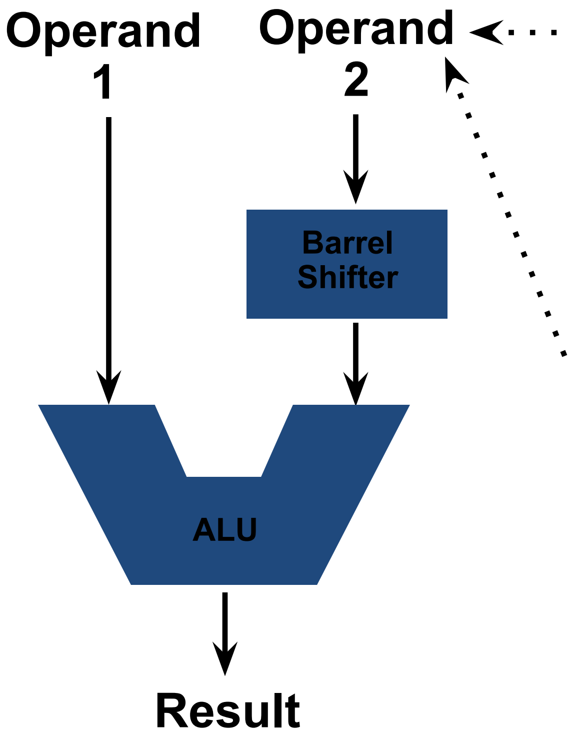
Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended



Single bit rotate with wrap around
from CF to MSB

Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation

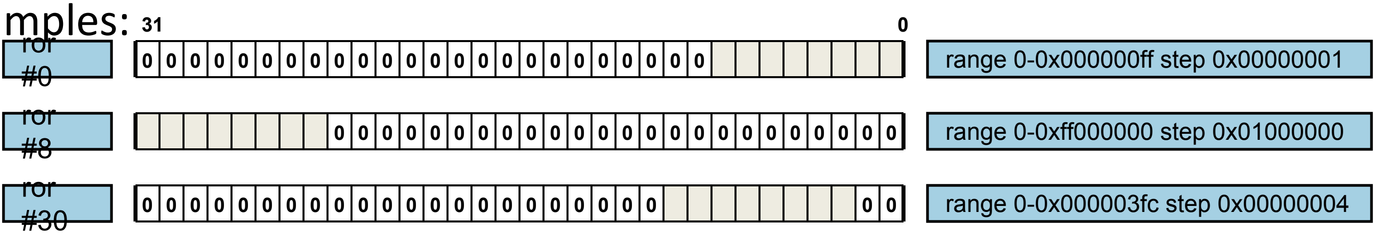
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Immediate constants

- Examples:



31 0

ror #0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | range 0-0x000000ff step 0x00000001

ror #8 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | range 0-0xff000000 step 0x01000000

ror #30 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | range 0-0x000003fc step 0x00000004

- The assembler converts immediate values to the rotate form:
  - `MOV r0,#4096          ; uses 0x40 ror 26`
  - `ADD r1,r2,#0xFF0000  ; uses 0xFF ror 16`

- The bitwise complements can also be formed using MVN:
  - `MOV r0, #0xFFFFFFFF  ; assembles to MVN r0,#0`

- Values that cannot be generated in this way will cause an error.

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR rd, =const`
- This will either:
  - Produce a `MOV` or `MVN` instruction to generate the value (if possible).

  or
  - Generate a `LDR` instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- For example

```
-   LDR r0,=0xFF              =>    MOV r0,#0xFF

-   LDR r0,=0x55555555        =>    LDR r0,[PC,#Imm12]
                                            …
                                            …
                                            DCD 0x55555555
```

- This is the recommended way of loading constants into a register

Multiply

- Syntax:
  - MUL{<cond>}{S} Rd, Rm, Rs   Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn   Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S}   RdLo, RdHi, Rm, Rs   RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs   RdHi,RdLo := (Rm*Rs)+RdHi,RdLo
- Cycle time
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for “long”
- Above are “general rules” - refer to the TRM for the core you are using for the exact details

Single register data transfer

LDR STR	Word	
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Memory system must support all access sizes

- Syntax:
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>

e.g. **LDREQB**

Address accessed

- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

    ```
    LDR r0,[r1,#8]
    ```

  - A register, optionally shifted by an immediate value

    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```

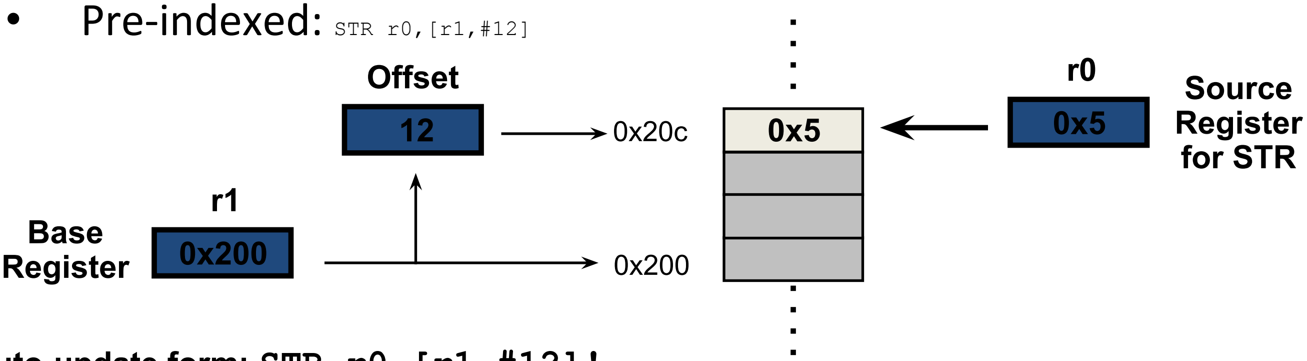
- This can be either added or subtracted from the base register:

```
LDR r0,[r1,#-8]
LDR r0,[r1,-r2]
LDR r0,[r1,-r2,LSL#2]
```

- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- Choice of *pre-indexed* or *post-indexed* addressing

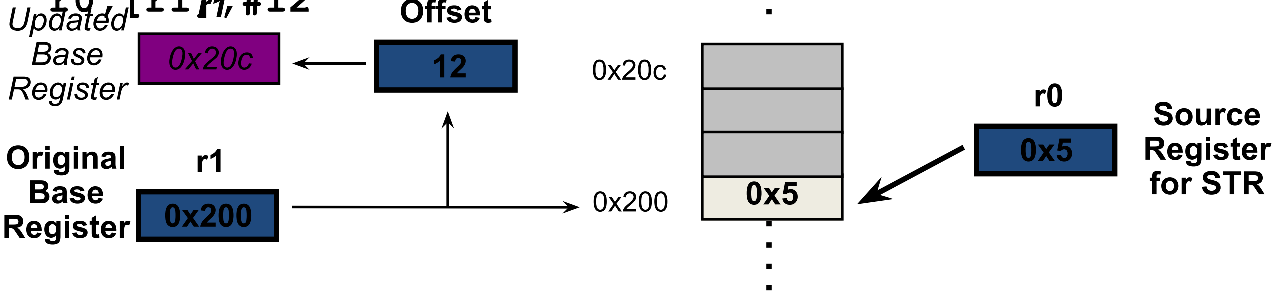
Pre or Post Indexed Addressing?

- Pre-indexed: `STR r0,[r1,#12]`



Auto-update form: `STR r0,[r1,#12]!`

- **Post-indexed: `STR r0, [r1], #12`**



LDM / STM operation

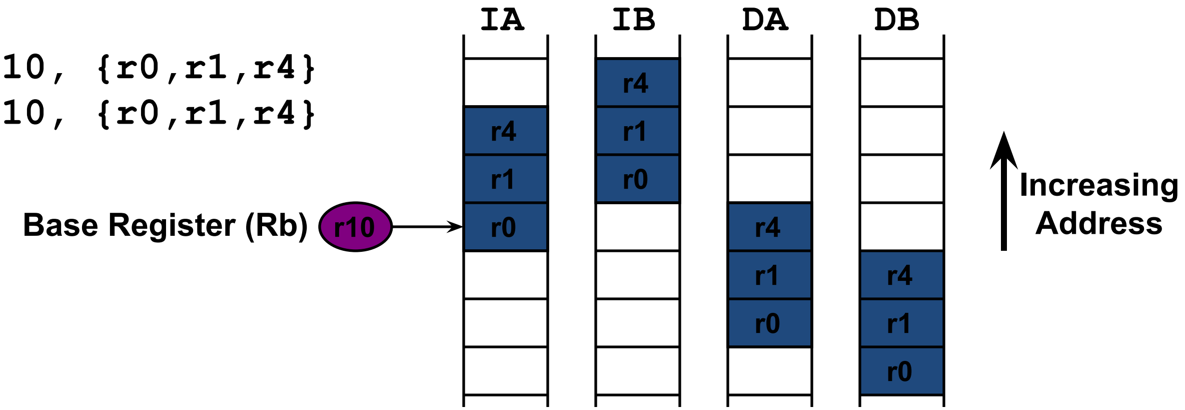
- Syntax:

  <LDM|STM>{<cond>}<addressing_mode> Rb{!}, <register list>

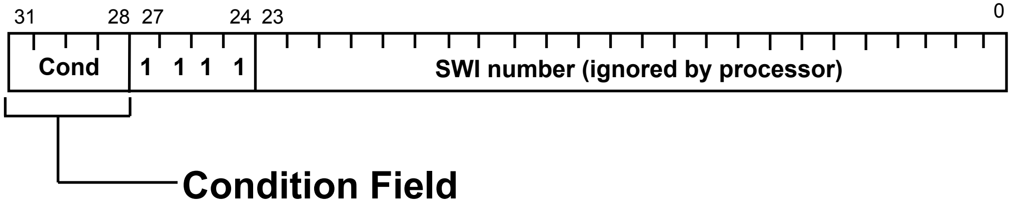
- 4 addressing modes:

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```

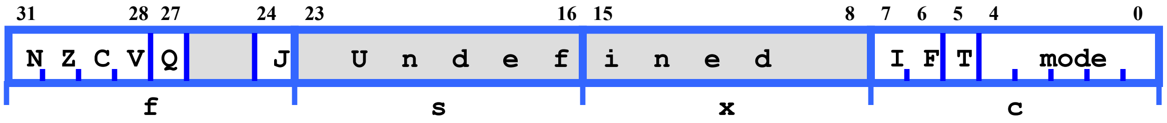


Software Interrupt (SWI)



- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
  - `SWI{<cond>} <SWI number>`

PSR Transfer Instructions



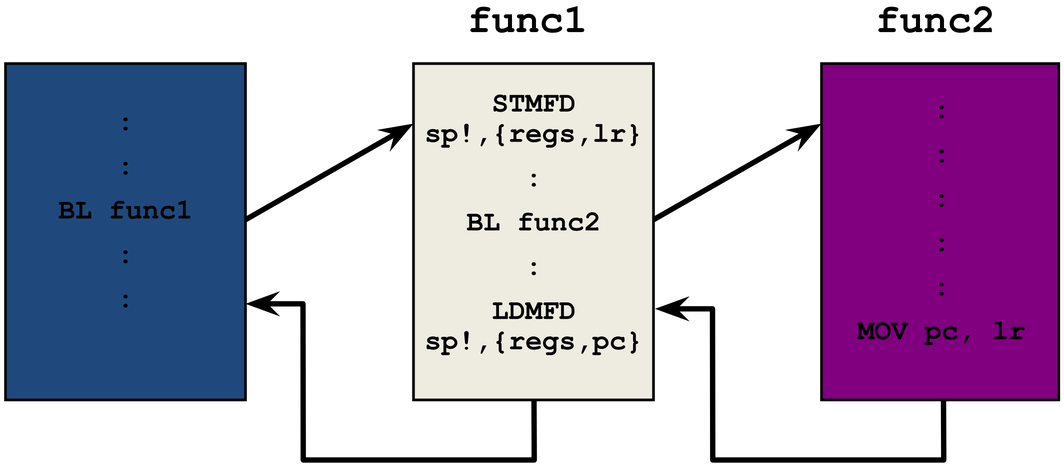
- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.
- Syntax:
  - `MRS{<cond>} Rd,<psr>           ; Rd = <psr>`
  - `MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm`

where
- `<psr> = CPSR or SPSR`
- `[_fields] = any combination of 'fsxc'`

- Also an immediate form
  - `MSR{<cond>} <psr_fields>,#Immediate`
- In User Mode, all bits can be read but only the condition flags (_f) can be written.

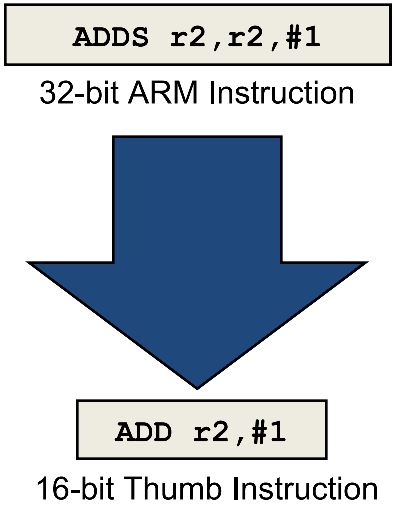
ARM Branches and Subroutines

- B <label>
  - PC relative. ±32 Mbyte range.
- BL <subroutine>
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Thumb

- Thumb is a 16-bit instruction set
  - Optimised for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
  - Switch between ARM and Thumb using **`BX`** instruction



`ADDS r2,r2,#1`

32-bit ARM Instruction

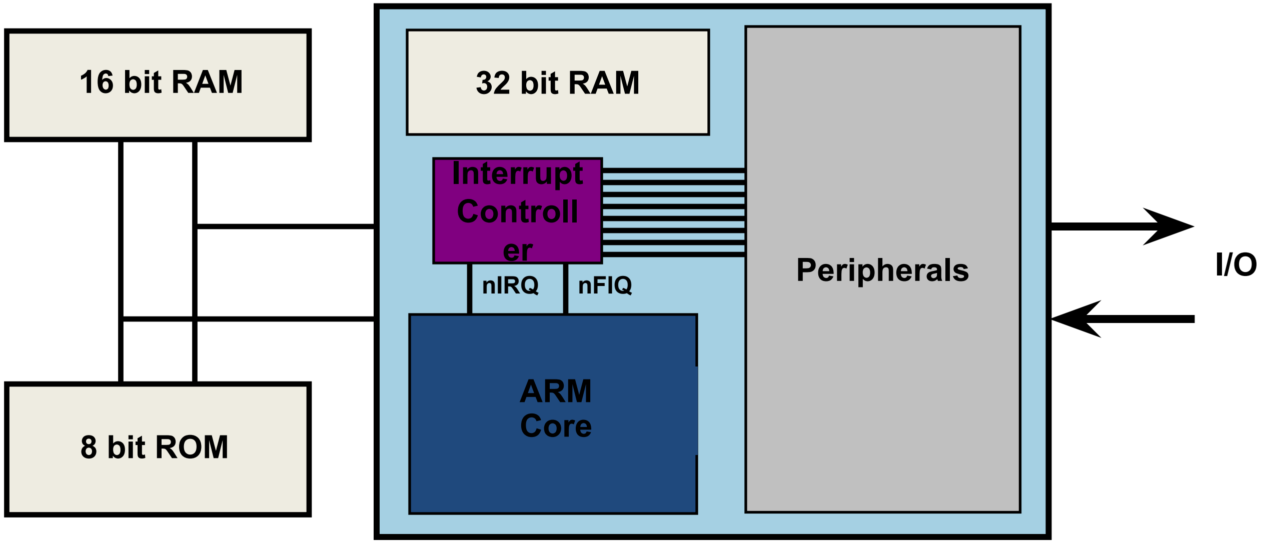
`ADD r2,#1`

16-bit Thumb Instruction

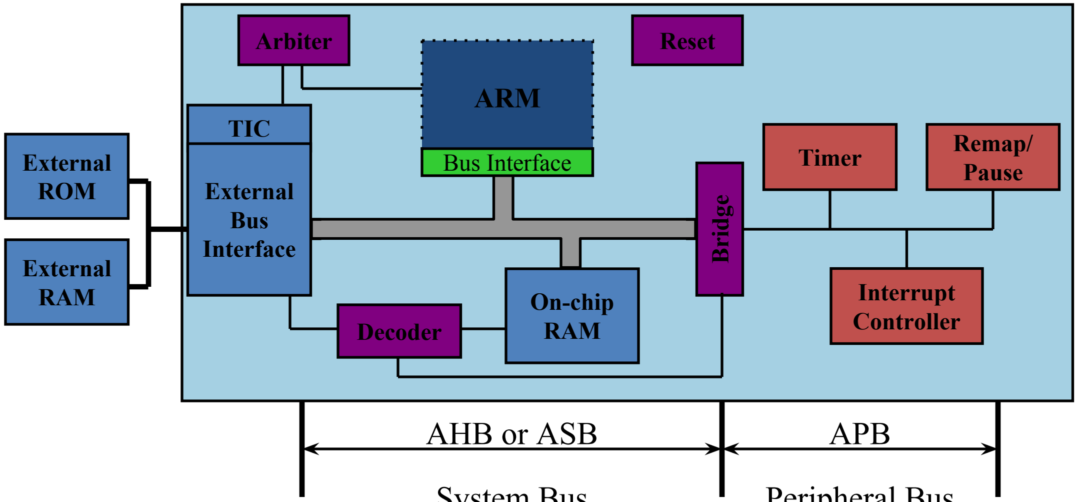
For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Example ARM-based System



AMBA



- AMBA
  - Advanced Microcontroller Bus Architecture
- ADK
  - Complete AMBA Design Kit
- ACT
  - AMBA Compliance Testbench
- PrimeCell
  - ARM’s AMBA compliant peripherals