ARM Processor Architecture

Jin-Fu Li

Department of Electrical Engineering

National Central University

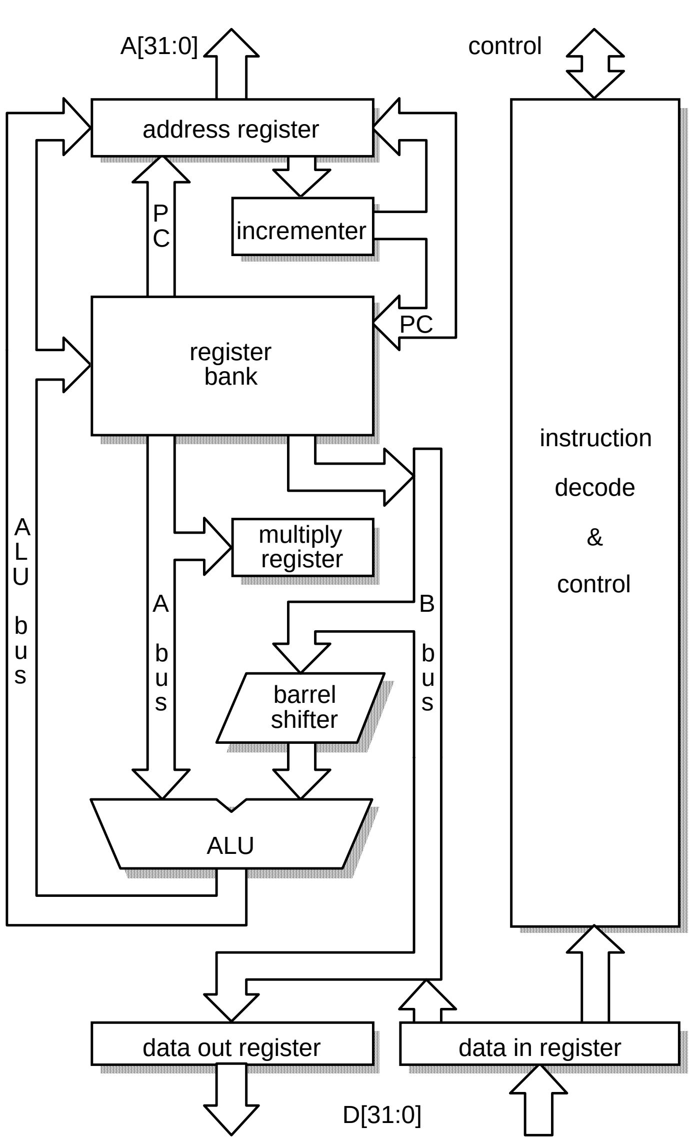
Adopted from National Chiao-Tung University
IP Core Design

Outline

- ARM Processor Core
- Memory Hierarchy
- Software Development
- Summary

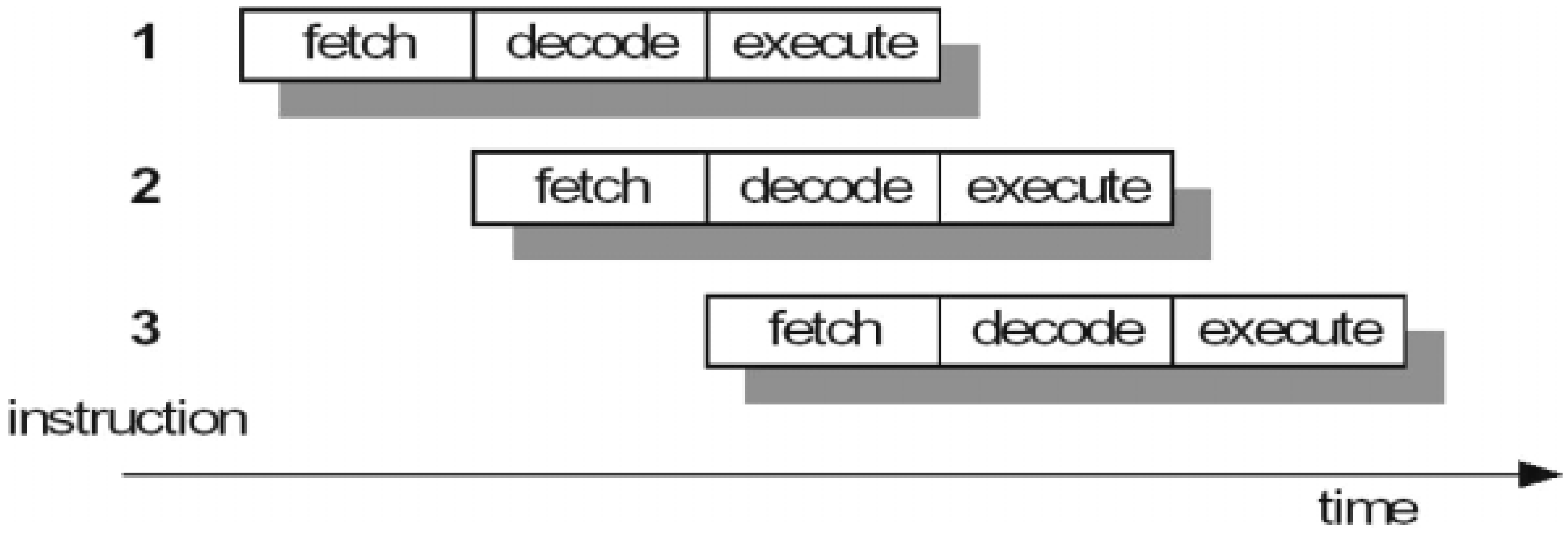
ARM Processor Core

3-Stage Pipeline ARM Organization



- ❑ Register Bank
  - – 2 read ports, 1 write ports, access any register
  - – 1 additional read port, 1 additional write port for r15 (PC)
- ❑ Barrel Shifter
  - – Shift or rotate the operand by any number of bits
- ❑ ALU
- ❑ Address register and incrementer
- ❑ Data Registers
  - – Hold data passing to and from memory
- ❑ Instruction Decoder and Control

3-Stage Pipeline (1/2)

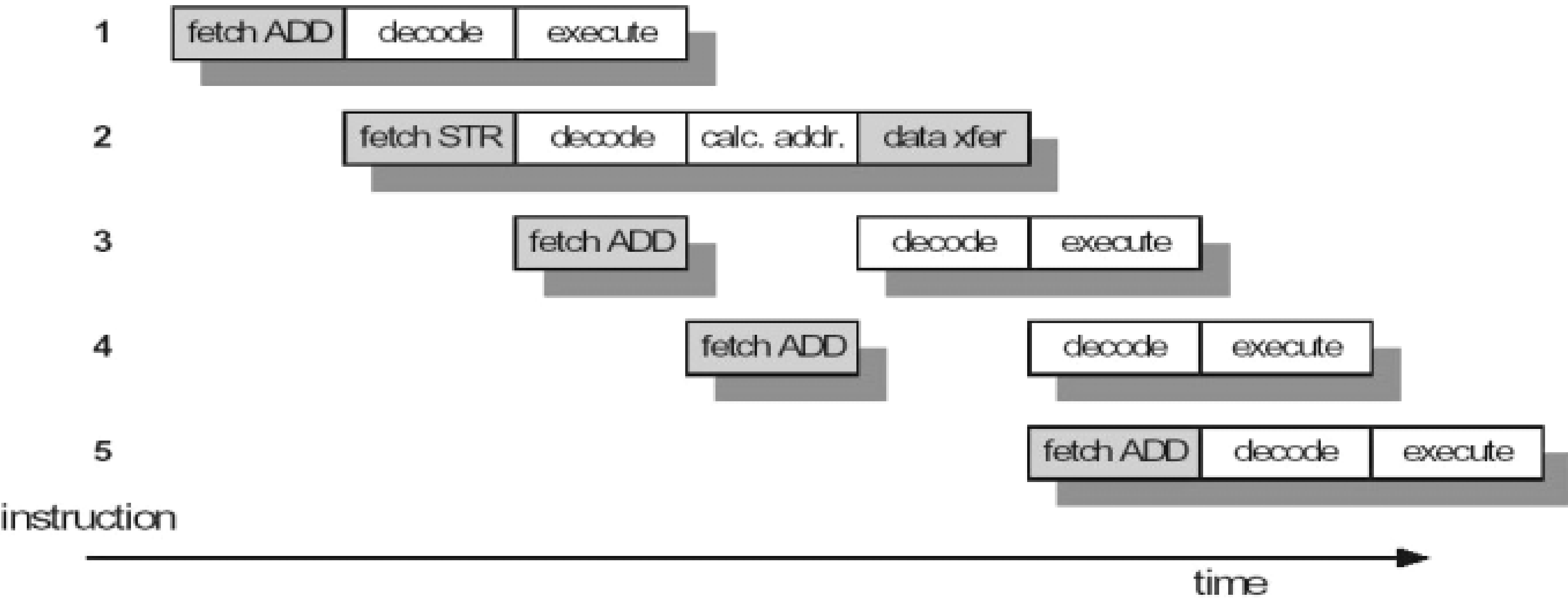


- **Fetch**
  - The instruction is fetched from memory and placed in the instruction pipeline
- **Decode**
  - The instruction is decoded and the datapath control signals prepared for the next cycle
- **Execute**
  - The register bank is read, an operand shifted, the ALU result generated and written back into destination register

3-Stage Pipeline (2/2)

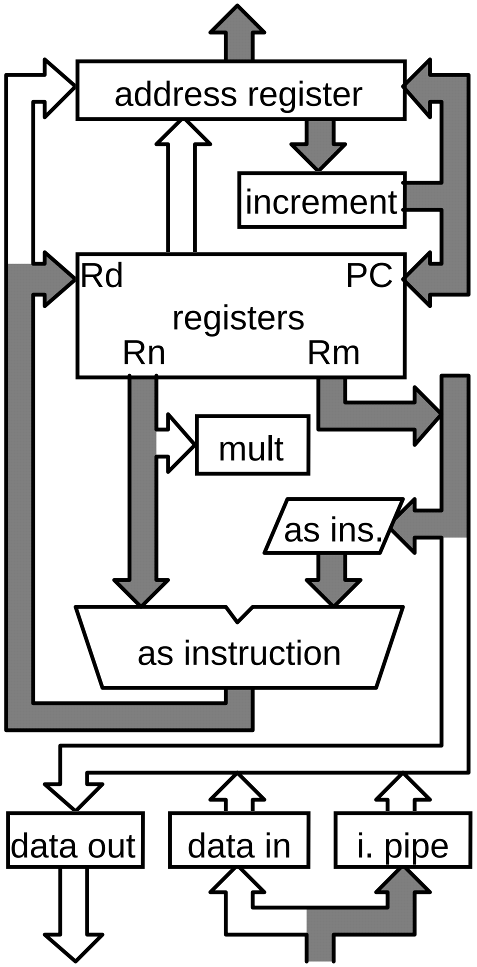
- At any time slice, 3 different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operations
- When the processor is executing data processing instructions , the latency = **3** cycles and the throughput = **1** instruction/cycle

Multi-cycle Instruction

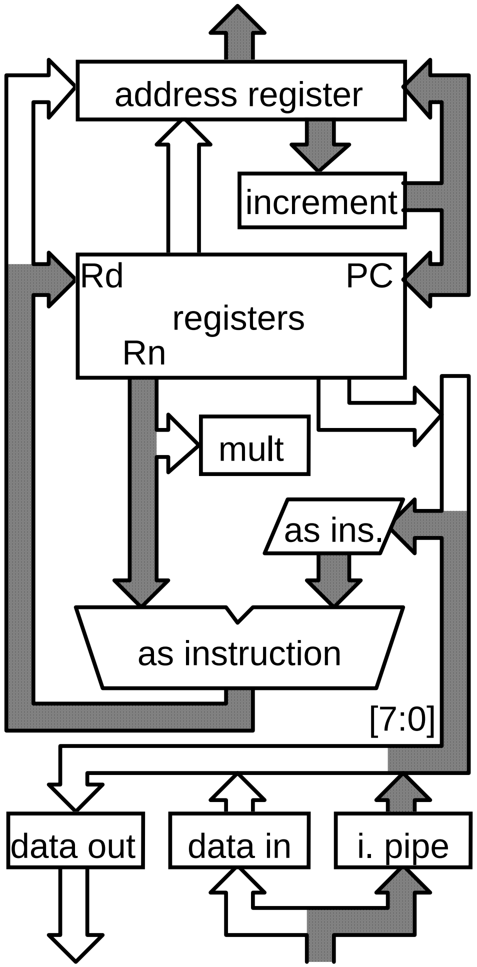


- ❑ Memory access (fetch, data transfer) in every cycle
- ❑ Datapath used in every cycle (execute, address calculation, data transfer)
- ❑ Decode logic generates the control signals for the data path use in next cycle (decode, address calculation)

Data Processing Instruction



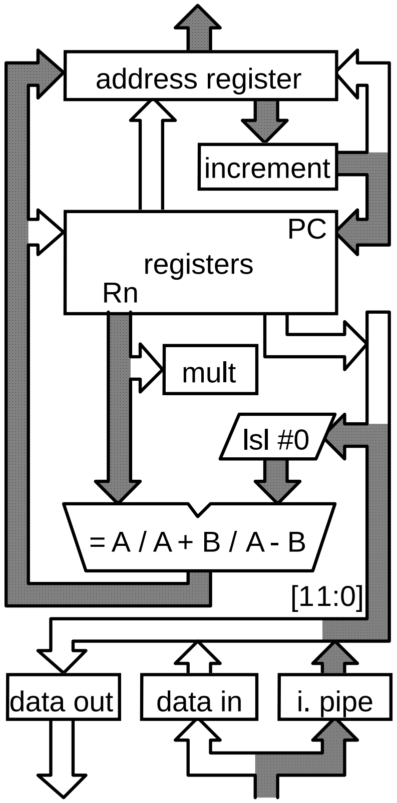
(a) register - register operations



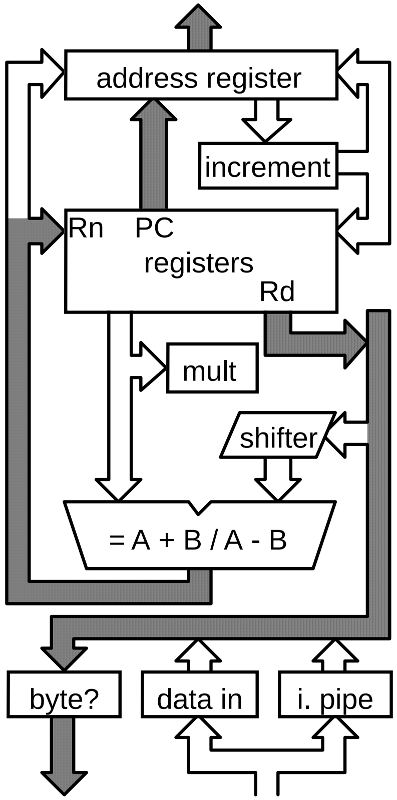
(b) register - immediate operations

- All operations take place in a single clock cycle

Data Transfer Instructions



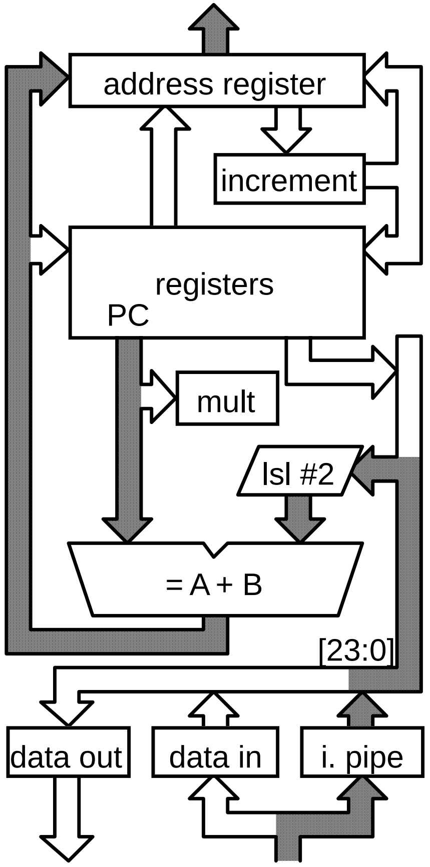
(a) 1st cycle - compute address



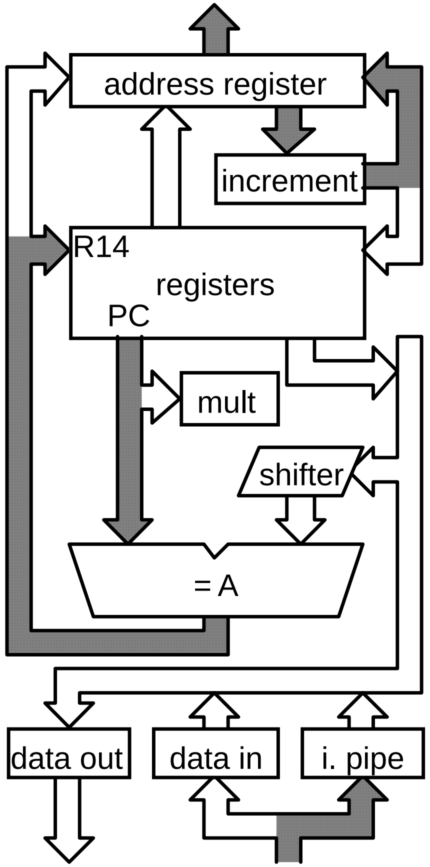
(b) 2nd cycle - store data & auto-index

- Computes a memory address similar to a data processing instruction
- Load instruction follow a similar pattern except that the data from memory only gets as far as the 'data in' register on the 2nd cycle and a 3rd cycle is needed to transfer the data from there to the destination register

Branch Instructions



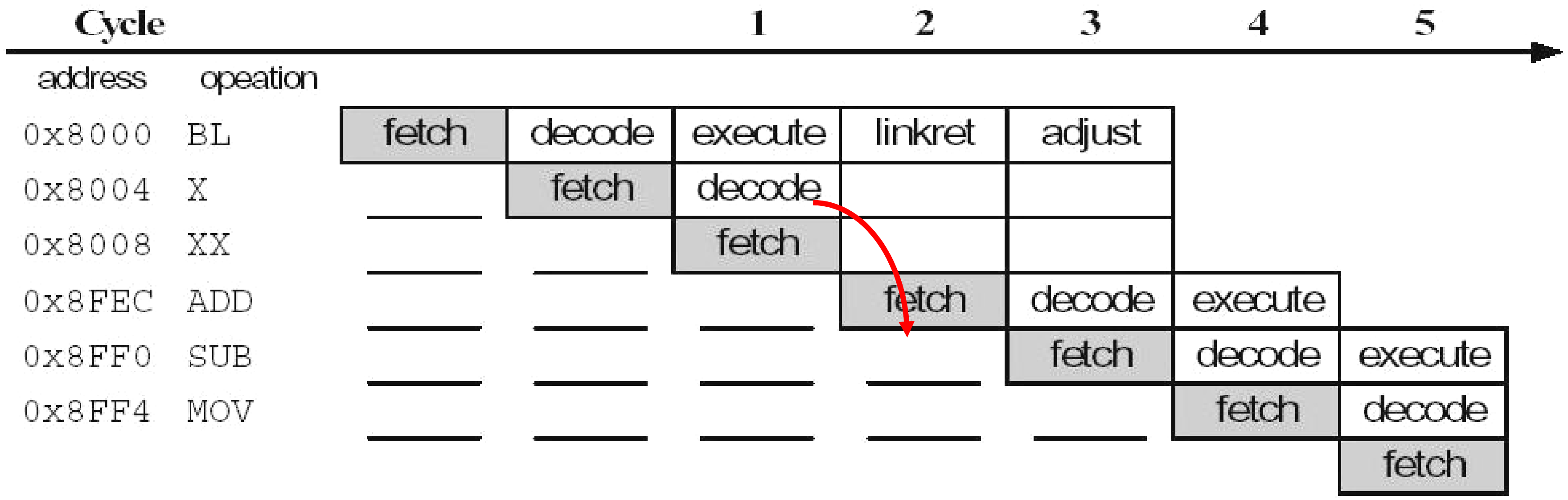
(a) 1st cycle - compute branch target



(b) 2nd cycle - save return address

- The third cycle, which is required to complete the pipeline refilling, is also used to mark the small correction to the value stored in the link register in order that is points directly at the instruction which follows the branch

Branch Pipeline Example

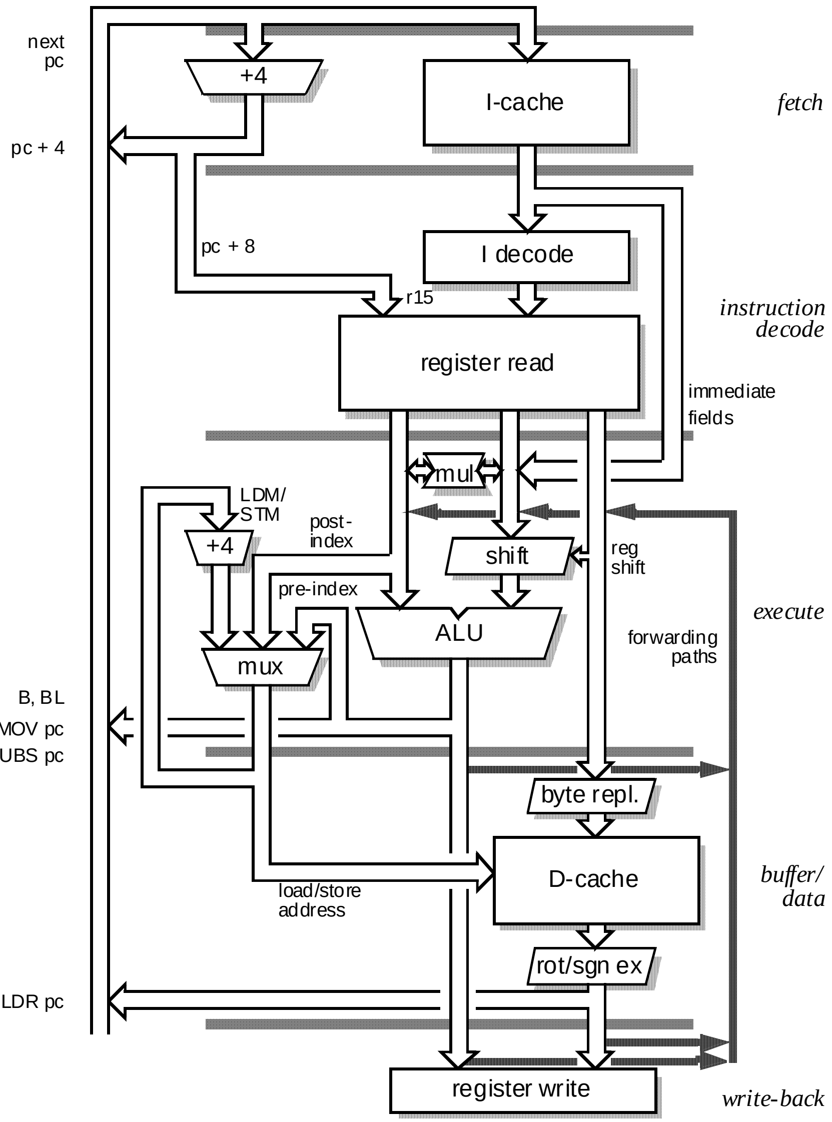


- ❑ Breaking the pipeline
- ❑ Note that the core is executing in the ARM state

5-Stage Pipeline ARM Organization

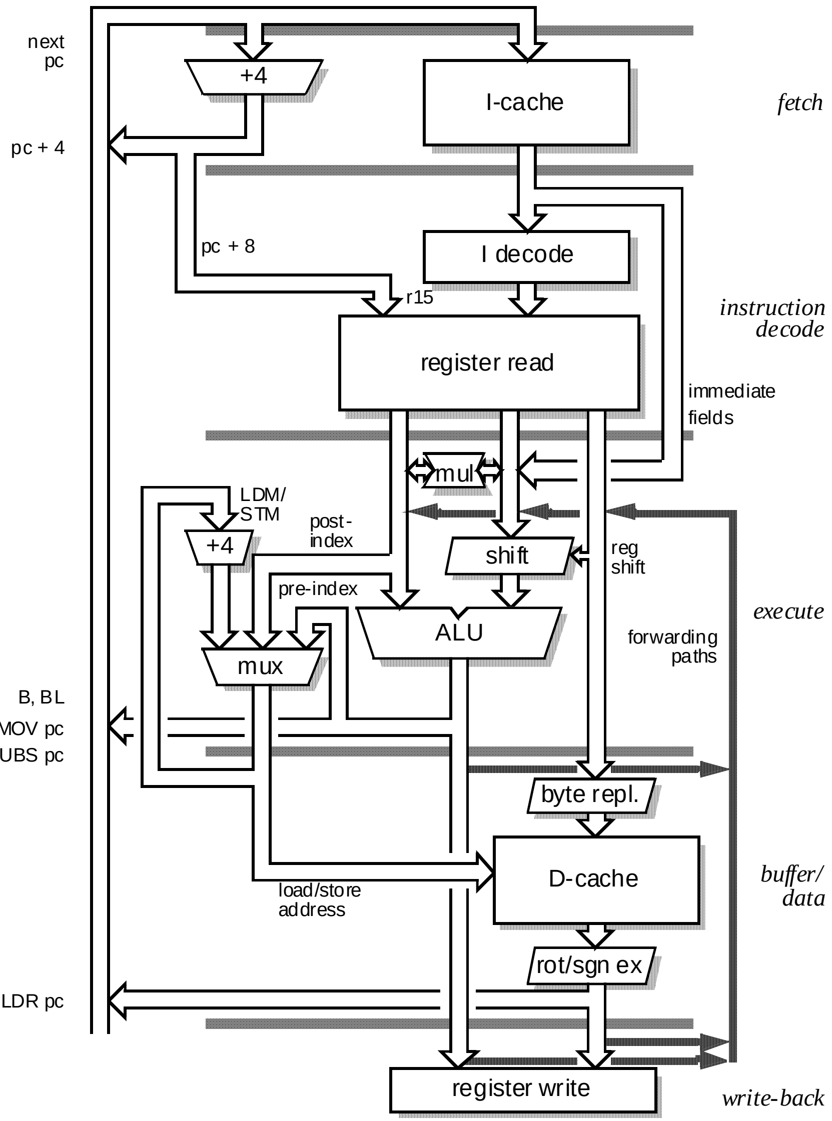
- $T_{prog} = N_{inst} * CPI / f_{clk}$
  – T_{prog}: the time that execute a given program
  – N_{inst}: the number of ARM instructions executed in the program => compiler dependent
  – CPI: average number of clock cycles per instructions => hazard causes pipeline stalls
  – f_{clk}: frequency
- Separate instruction and data memories => **5** stage pipeline
- Used in ARM9TDMI

5-Stage Pipeline Organization (1/2)



- Fetch
  - The instruction is fetched from memory and placed in the instruction pipeline
- Decode
  - The instruction is decoded and register operands read from the register files. There are 3 operand read ports in the register file so most ARM instructions can source all their operands in one cycle
- Execute
  - An operand is shifted and the ALU result generated. If the instruction is a load or store, the memory address is computed in the ALU

5-Stage Pipeline Organization (2/2)



- Buffer/Data
  - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle
- Write back
  - The result generated by the instruction are written back to the register file, including any data loaded from memory

Pipeline Hazards

- There are situations, called ***hazards***, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.
- There are three classes of hazards:
  - **Structural Hazards**: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - **Data Hazards**: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - **Control Hazards**: They arise from the pipelining of branches and other instructions that change the PC

Structural Hazards

- When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- If some combination of instructions cannot be accommodated because of a *resource conflict*, the machine is said to have a **structural hazard**.

Example

- A machine has shared a **single-memory** pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load), it will conflict with the instruction reference for a later instruction (instr 3):

Clock cycle number								
instr	1	2	3	4	5	6	7	8
load	IF	ID	EX	**MEM**	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				**IF**	ID	EX	MEM	WB

Solution (1/2)

- To resolve this, we *stall* the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number									
instr	1	2	3	4	5	6	7	8	9
load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				**stall**	**IF**	ID	EX	MEM	WB

Solution (2/2)

- Another solution is to use separate instruction and data memories.
- ARM is use **Harvard** architecture, so we do not have this hazard

Data Hazards

- **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

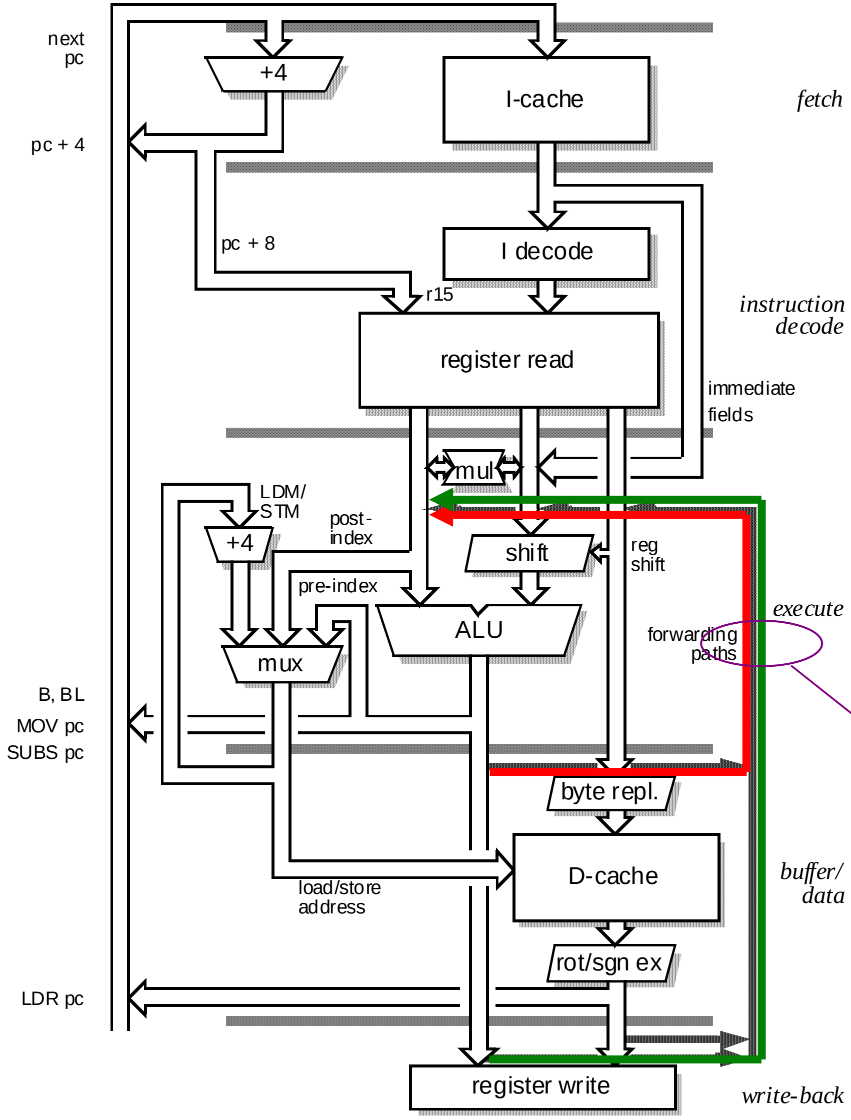
Clock cycle number										
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB			
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB		
OR	R8,R1,R9				IF	ID_{or}	EX	MEM	WB	
XOR	R10,R1,R11					IF	ID_{xor}	EX	MEM	WB

Forwarding

- The problem with data hazards, introduced by this sequence of instructions can be solved with a simple hardware technique called ***forwarding***.

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**		
SUB	R4,R5,R1		IF	ID_{sub}	EX	MEM	WB	
AND	R6,R1,R7			IF	ID_{and}	EX	MEM	WB

Forwarding Architecture



- Forwarding works as follows:
  - The ALU result from the EX/MEM register is always ***fed back*** to the ALU input latches.
  - If the forwarding hardware detects that the previous ALU operation has written the register corresponding to the source for the current ALU operation, ***control logic*** selects the forwarded result as the ALU input rather than the value read from the register file.

forwarding paths

Forward Data

Clock cycle number								
		1	2	3	4	5	6	7
ADD	R1,R2,R3	IF	ID	EX_{add}	MEM_{add}	WB		
SUB	R4,R5,R1		IF	ID	EX_{sub}	MEM	WB	
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB

- The first forwarding is for value of **R1** from **EX_{add}** to **EX_{sub}**. The second forwarding is also for value of **R1** from **MEM_{add}** to **EX_{and}**. This code now can be executed without stalls.
- Forwarding can be generalized to include passing the result directly to the functional unit that requires it: a result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Without Forward

Clock cycle number										
		1	2	3	4	5	6	7	8	9
ADD	R1,R2,R3	IF	ID	EX	MEM	**WB**				
SUB	R4,R5,R1		IF	*stall*	*stall*	ID$_{sub}$	EX	MEM	WB	
AND	R6,R1,R7			*stall*	*stall*	IF	ID$_{and}$	EX	MEM	WB

Data forwarding

- Data dependency arises when an instruction needs to use the result of one of its predecessors before the result has returned to the register file => pipeline hazards
- Forwarding paths allow results to be passed between stages as soon as they are available
- 5-stage pipeline requires each of the three source operands to be forwarded from any of the intermediate result registers
- Still one load stall

```
LDR rN, […]
ADD r2,r1,rN  ;use rN immediately
```

  – One stall
  – Compiler rescheduling

Stalls are required

		1	2	3	4	5	6	7	8
LDR	R1,@(R2)	IF	ID	EX	MEM	WB			
SUB	R4,R1,R5		IF	ID	EX_{sub}	MEM	WB		
AND	R6,R1,R7			IF	ID	EX_{and}	MEM	WB	
OR	R8,R1,R9				IF	ID	EXE	MEM	WB

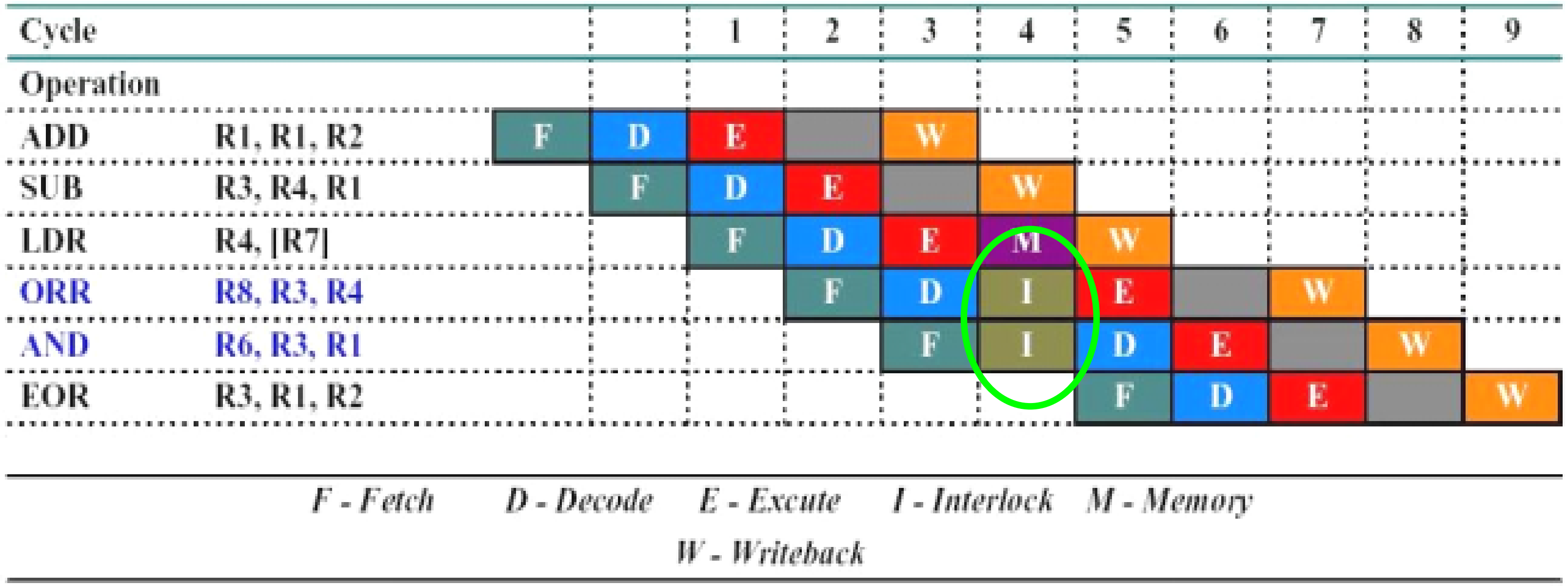
- The load instruction has a delay or latency that cannot be eliminated by forwarding alone.

The Pipeline with one Stall

		1	2	3	4	5	6	7	8	9
LDR	R1,@(R2)	IF	ID	EX	MEM	WB				
SUB	R4,R1,R5		IF	ID	*stall*	EX$_{sub}$	MEM	WB		
AND	R6,R1,R7			IF	*stall*	ID	EX	MEM	WB	
OR	R8,R1,R9				*stall*	IF	ID	EX	MEM	WB

- The only necessary forwarding is done for R1 from **MEM** to **EX$_{sub}$**.

LDR Interlock

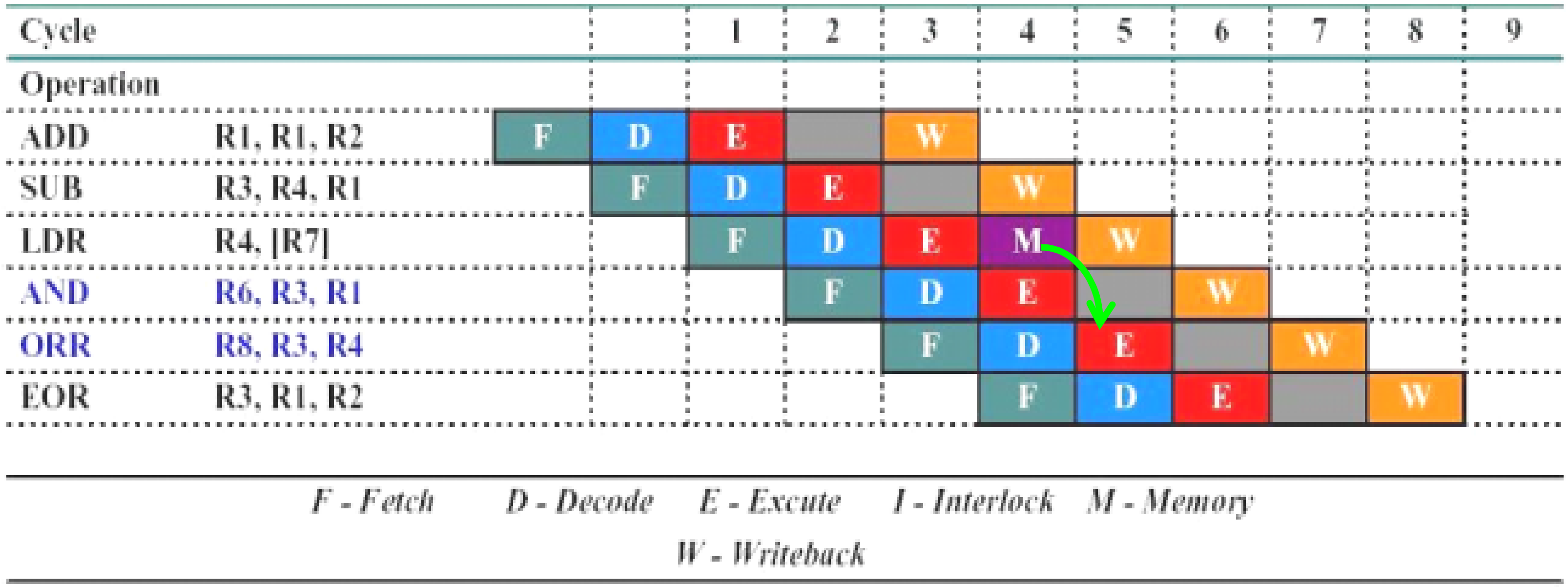


Cycle				1	2	3	4	5	6	7	8	9
Operation												
ADD	R1, R1, R2	F	D	E	▨	W						
SUB	R3, R4, R1		F	D	E	▨	W					
LDR	R4, [R7]			F	D	E	M	W				
ORR	R8, R3, R4				F	D	I	E	▨	W		
AND	R6, R3, R1					F	I	D	E	▨	W	
EOR	R3, R1, R2							F	D	E	▨	W

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*
W - Writeback

- In this example, it takes 7 clock cycles to execute 6 instructions, CPI of 1.2
- The LDR instruction immediately followed by a data operation using the same register cause an interlock

Optimal Pipelining

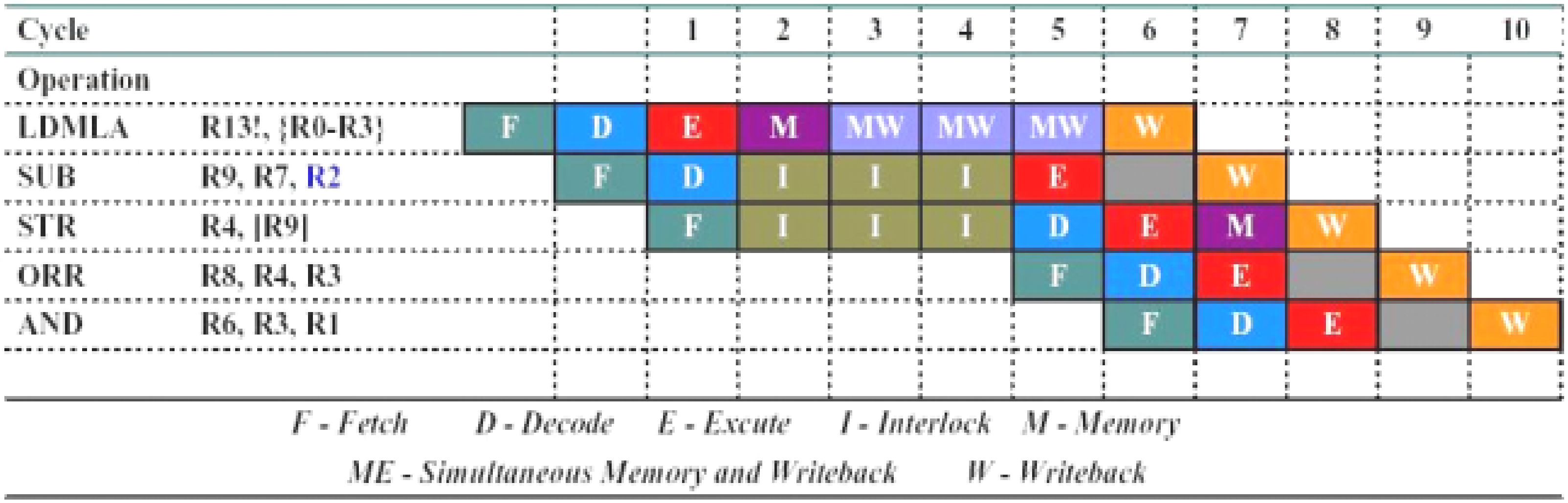


Cycle				1	2	3	4	5	6	7	8	9
Operation												
ADD	R1, R1, R2	F	D	E		W						
SUB	R3, R4, R1		F	D	E		W					
LDR	R4, [R7]			F	D	E	M	W				
AND	R6, R3, R1				F	D	E		W			
ORR	R8, R3, R4					F	D	E		W		
EOR	R3, R1, R2						F	D	E		W	

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*
W - Writeback

- In this example, it takes 6 clock cycles to execute 6 instructions, CPI of 1
- The LDR instruction does not cause the pipeline to interlock

LDM Interlock (1/2)



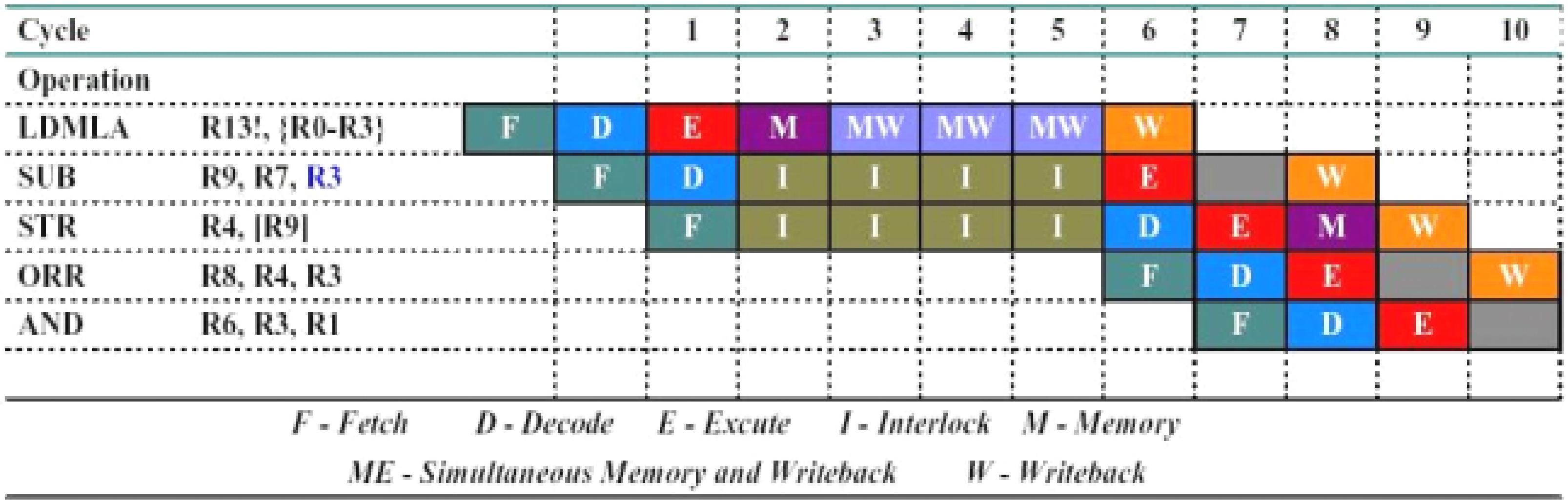
Cycle				1	2	3	4	5	6	7	8	9	10
Operation													
LDMLA	R13!, {R0-R3}	F	D	E	M	MW	MW	MW	W				
SUB	R9, R7, R2		F	D	I	I	I	E	▨	W			
STR	R4, [R9]			F	I	I	I	D	E	M	W		
ORR	R8, R4, R3							F	D	E	▨	W	
AND	R6, R3, R1								F	D	E	▨	W

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*

ME - Simultaneous Memory and Writeback *W - Writeback*

- ❑ In this example, it takes 8 clock cycles to execute 5 instructions, CPI of 1.6
- ❑ During the LDM there are parallel memory and writeback cycles

LDM Interlock (2/2)



Cycle				1	2	3	4	5	6	7	8	9	10
Operation													
LDMLA	R13!, {R0-R3}	F	D	E	M	MW	MW	MW	W				
SUB	R9, R7, R3		F	D	I	I	I	I	E	▨	W		
STR	R4, [R9]			F	I	I	I	I	D	E	M	W	
ORR	R8, R4, R3								F	D	E	▨	W
AND	R6, R3, R1									F	D	E	▨

F - Fetch *D - Decode* *E - Excute* *I - Interlock* *M - Memory*

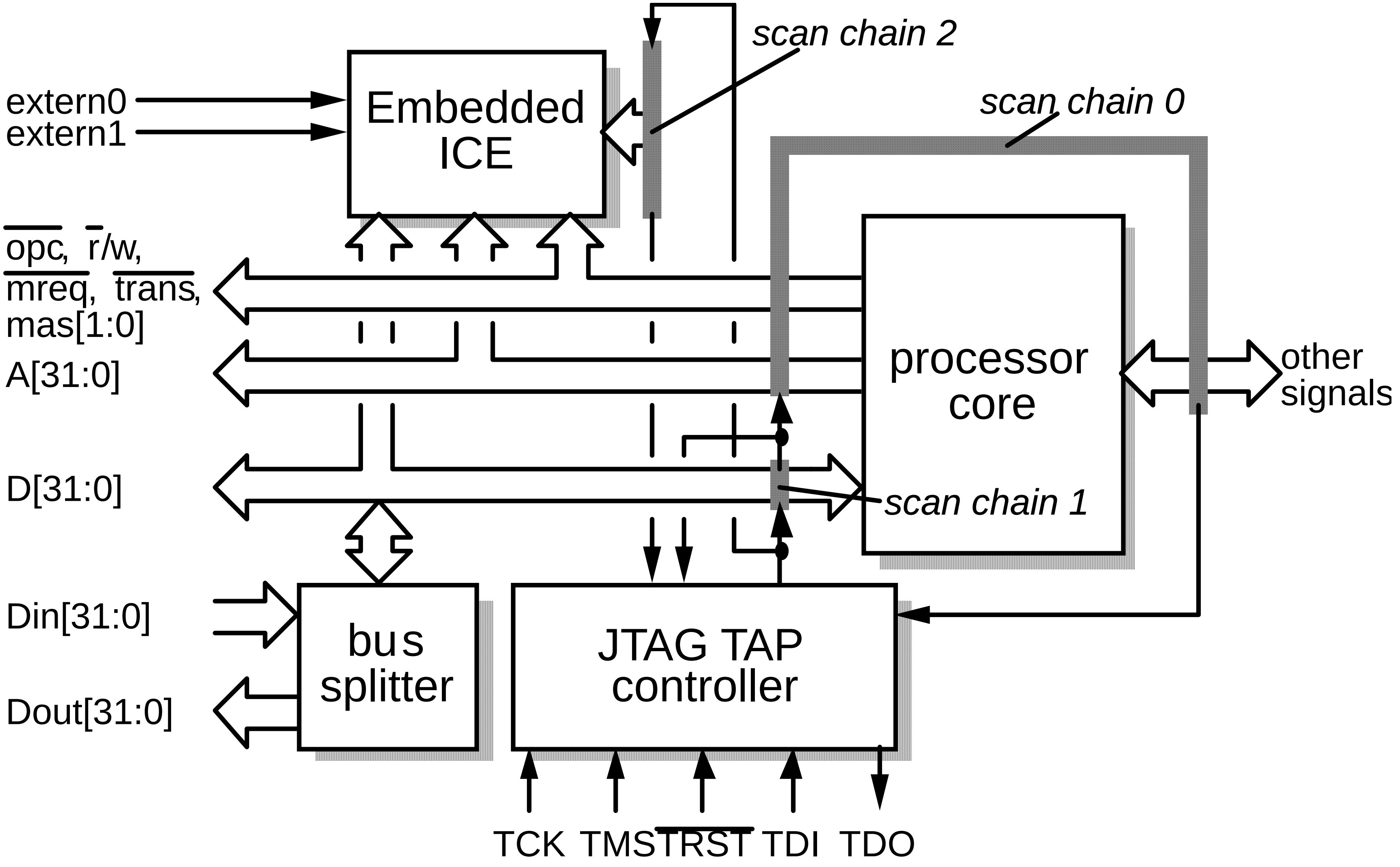
ME - Simultaneous Memory and Writeback *W - Writeback*

- ❑ In this example, it takes 9 clock cycles to execute 5 instructions, CPI of 1.8
- ❑ The SUB incurs a further cycle of interlock due to it using the highest specified register in the LDM instruction

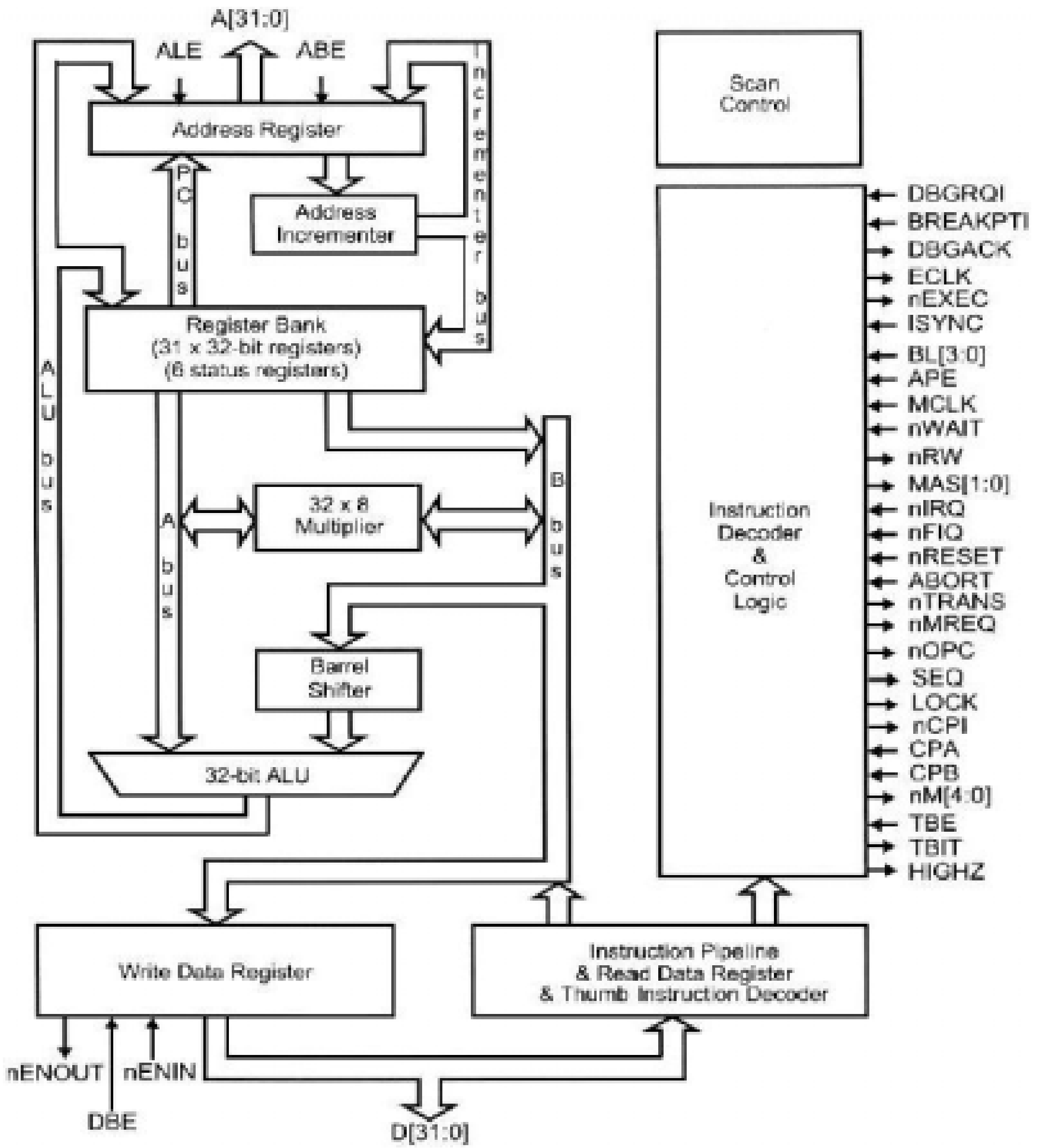
ARM7TDMI Processor Core

- ❑ Current low-end ARM core for applications like digital mobile phones
- ❑ TDMI
  - **T**: Thumb, 16-bit compressed instruction set
  - **D**: on-chip Debug support, enabling the processor to halt in response to a debug request
  - **M**: enhanced Multiplier, yield a full 64-bit result, high performance
  - **I**: EmbeddedICE hardware
- ❑ Von Neumann architecture
- ❑ 3-stage pipeline, CPI ~ 1.9

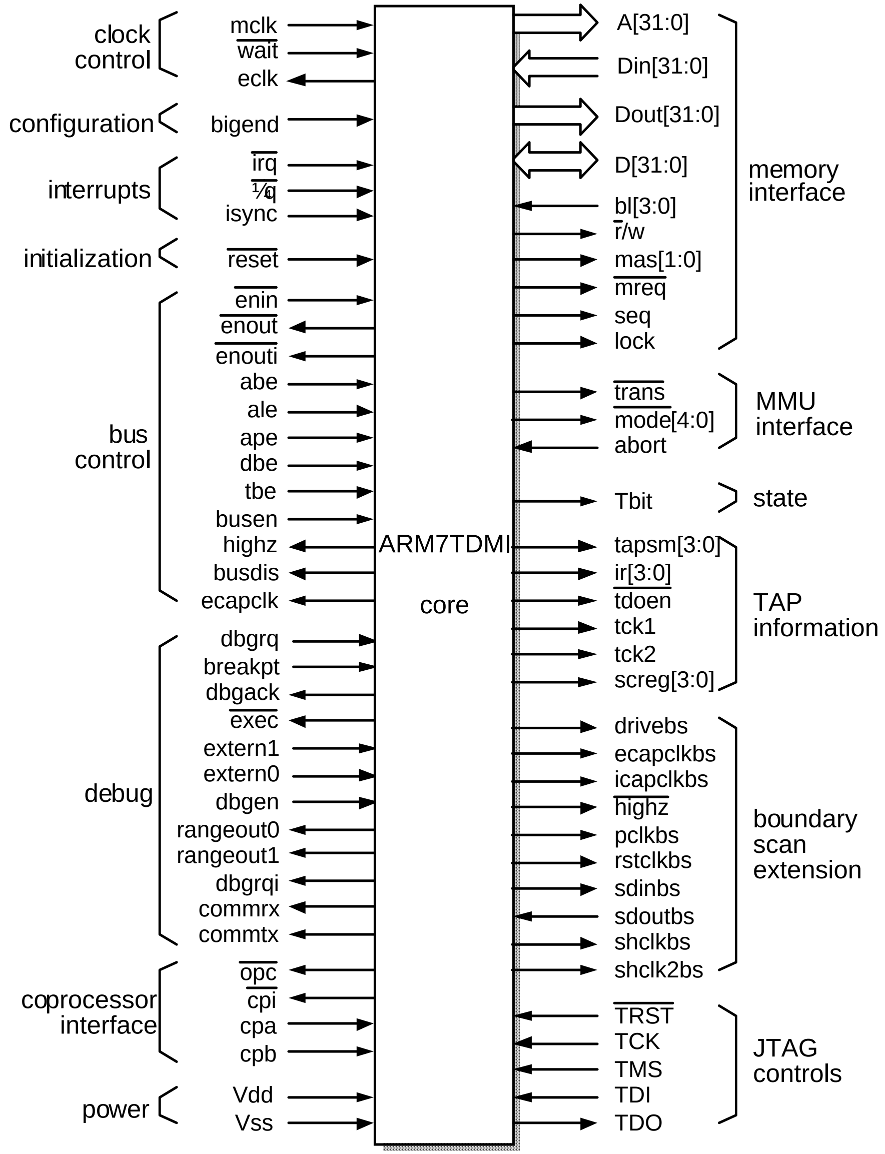
ARM7TDMI Block Diagram



ARM7TDMI Core Diagram



ARM7TDMI Interface Signals (1/4)



ARM7TDMI Interface Signals (2/4)

- Clock control
  - All state change within the processor are controlled by *mclk*, the memory clock
  - Internal clock = mclk AND \wait
  - eclk clock output reflects the clock used by the core
- Memory interface
  - 32-bit address A[31:0], bidirectional data bus D[31:0], separate data out Dout[31:0], data in Din[31:0]
  - \mreq indicates that the memory address will be sequential to that used in the previous cycle

mreq	**seq**	**Cycle**	**Use**
0	0	N	Non-sequential memory access
0	1	S	Sequential memory access
1	0	I	Internal cycle – bus and memory inactive
1	1	C	Coprocessor register transfer – memory inactive

ARM7TDMI Interface Signals (3/4)

- Lock indicates that the processor should keep the bus to ensure the atomicity of the read and write phase of a SWAP instruction
- \r/w, read or write
- mas[1:0], encode memory access size – byte, half – word or word
- bl[3:0], externally controlled enables on latches on each of the 4 bytes on the data input bus

❑ MMU interface

- \trans (translation control), 0: user mode, 1: privileged mode
- \mode[4:0], bottom 5 bits of the CPSR (inverted)
- Abort, disallow access

❑ State

- T bit, whether the processor is currently executing ARM or Thumb instructions

❑ Configuration

- Bigend, big-endian or little-endian

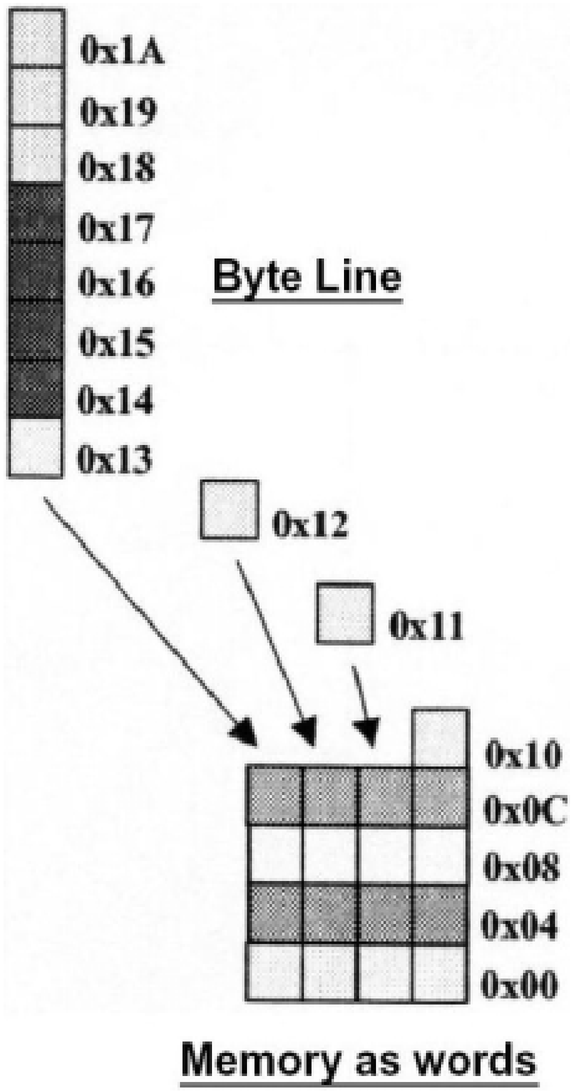
ARM7TDMI Interface Signals (4/4)

- Interrupt
  - \fiq, fast interrupt request, higher priority
  - \irq, normal interrupt request
  - isync, allow the interrupt synchronizer to be passed
- Initialization
  - \reset, starts the processor from a known state, executing from address 00000000_{16}
- ARM7TDMI characteristics

Process	0.35 um	**Transistors**	74,209	**MIPS**	60
Metal layers	3	**Core area**	2.1 mm^2	**Power**	87 mW
Vdd	3.3 V	**Clock**	0 to 66 MHz	**MIPS/W**	690

Memory Access

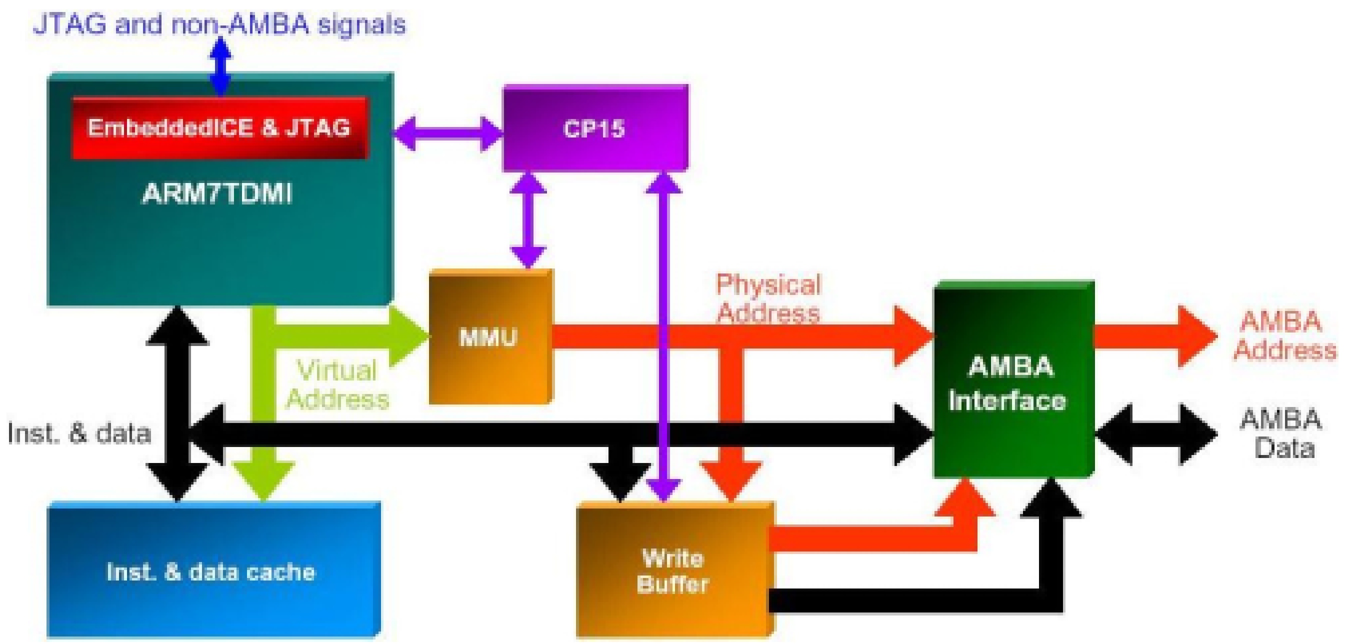
- The ARM7 is a Von Neumann, load/store architecture, i.e.,
  - Only 32 bit data bus for both inst. And data.
  - Only the load/store inst. (and SWP) access memory.
- Memory is addressed as a 32 bit address space
- Data type can be 8 bit bytes, 16 bit half-words or 32 bit words, and may be seen as a byte line folded into 4-byte words
- Words must be aligned to 4 byte boundaries, and half-words to 2 byte boundaries.
- Always ensure that memory controller supports all three access sizes



ARM Memory Interface

- Sequential (S cycle)
  - (nMREQ, SEQ) = (0, 1)
  - The ARM core requests a transfer to or from an address which is either the same, or one word or one-half-word greater than the preceding address.
- Non-sequential (N cycle)
  - (nMREQ, SEQ) = (0, 0)
  - The ARM core requests a transfer to or from an address which is unrelated to the address used in the preceding address.
- Internal (I cycle)
  - (nMREQ, SEQ) = (1, 0)
  - The ARM core does not require a transfer, as it performing an internal function, and no useful prefetching can be performed at the same time
- Coprocessor register transfer (C cycle)
  - (nMREQ, SEQ) = (1, 1)
  - The ARM core wished to use the data bus to communicate with a coprocessor, but does no require any action by the memory system.

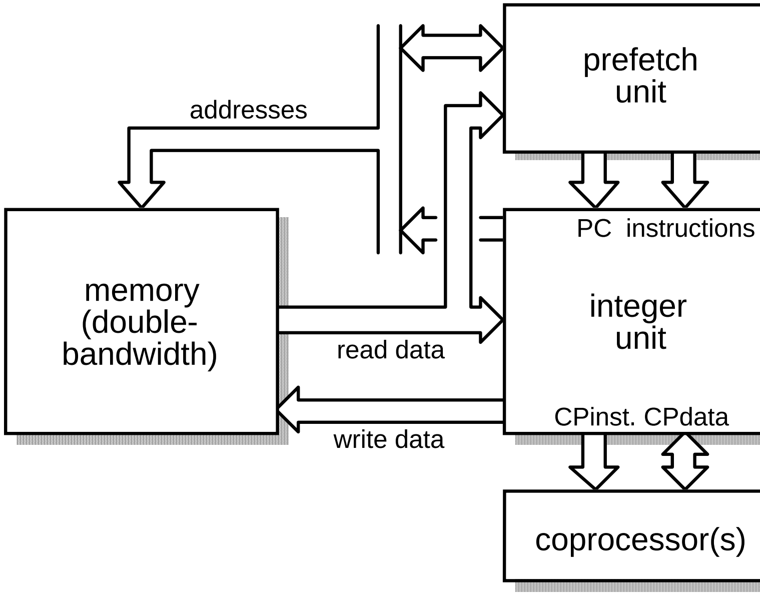
Cached ARM7TDMI Macrocells



- ARM710T
  - 8K unified write through cache
  - Full memory management unit supporting virtual memory
  - Write buffer
- ARM720T
  - As ARM 710T but with WinCE support
- ARM 740T
  - 8K unified write through cache
  - Memory protection unit
  - Write buffer

ARM8

- Higher performance than ARM7
  - By increasing the clock rate
  - By reducing the CPI
    - Higher memory bandwidth, 64-bit wide memory
    - Separate memories for instruction and data accesses
- $$\begin{tikzcd} \text{ARM 8} \arrow[r] \arrow[dr] & \text{ARM9TDMI} \\ & \text{ARM10TDMI} \end{tikzcd}$$
- Core Organization
  - The prefetch unit is responsible for fetching instructions from memory and buffering them (exploiting the double bandwidth memory)
  - It is also responsible for branch prediction and use static prediction based on the branch prediction (backward: predicted 'taken'; forward: predicted 'not taken')



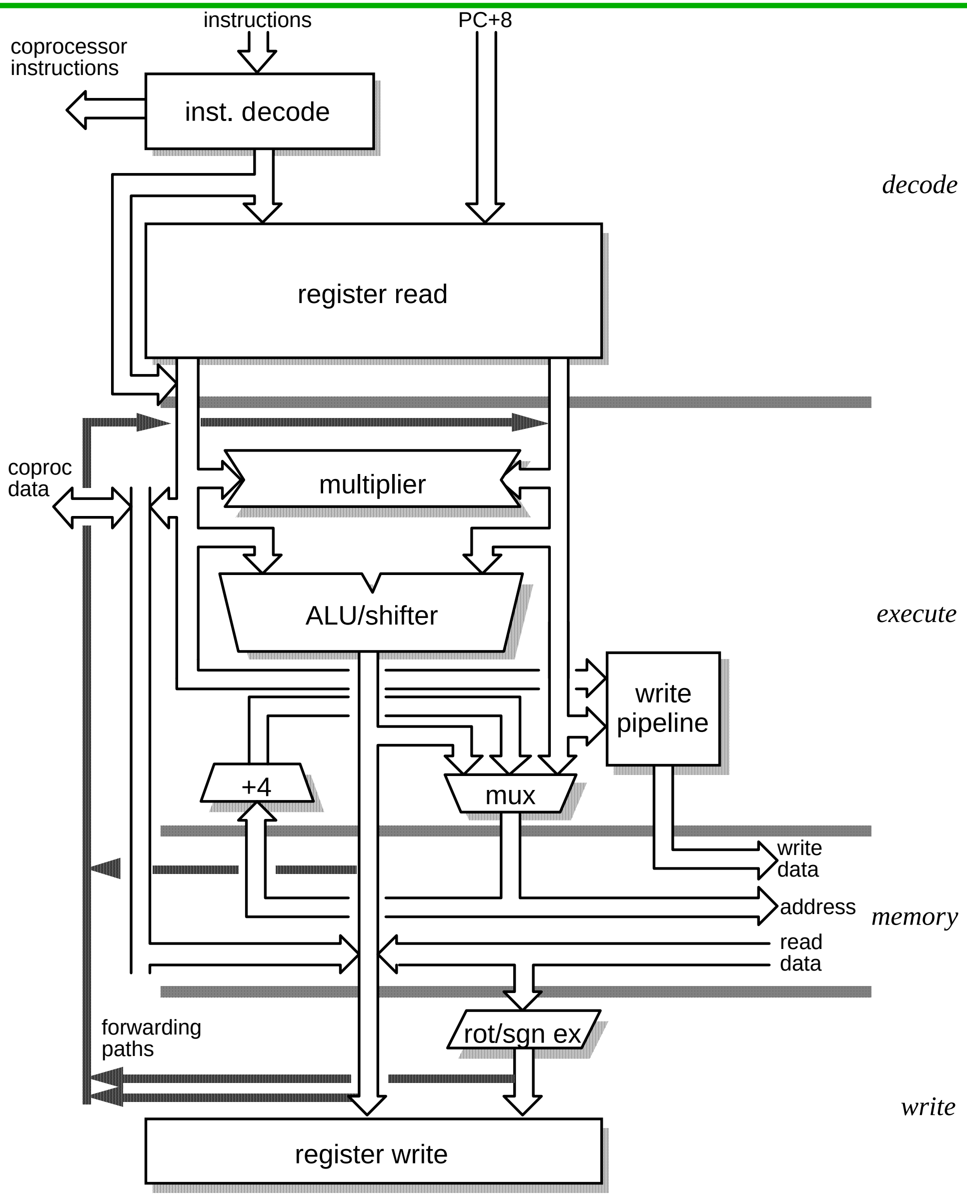
Pipeline Organization

- 5-stage, prefetch unit occupies the 1st stage, integer unit occupies the remainder

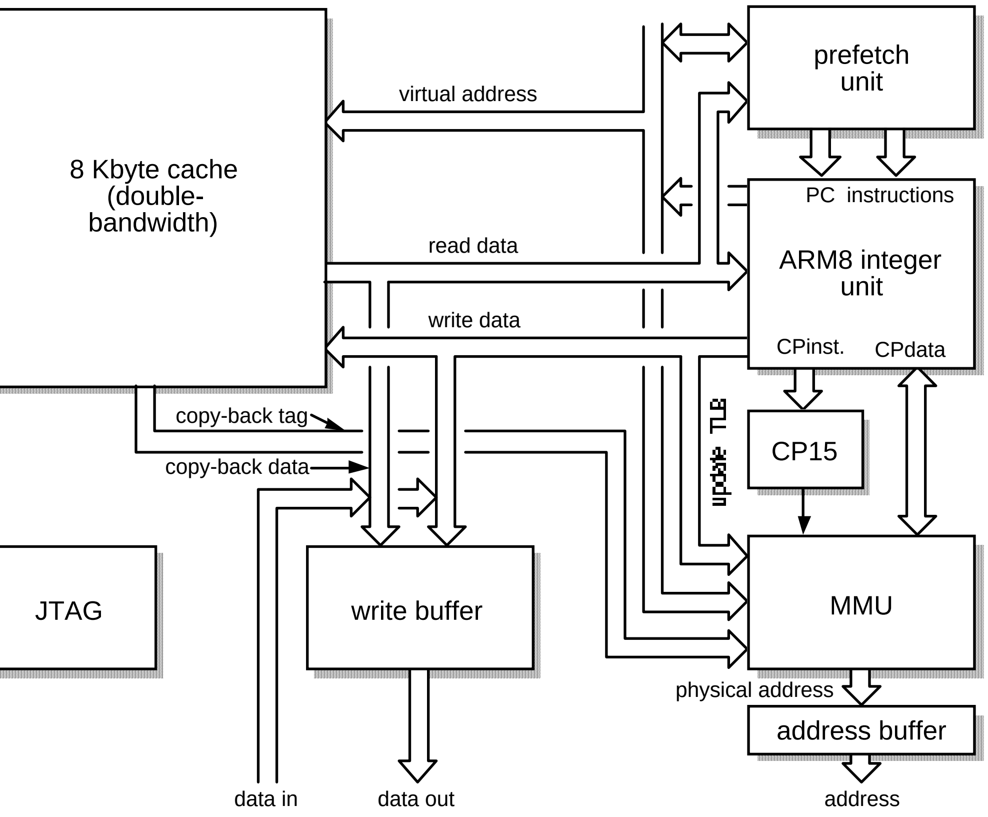
(1) Instruction prefetch $\longrightarrow$ **Prefetch Unit**

$$\left.\begin{array}{l}\text{(2) Instruction decode and register read}\\ \text{(3) Execute (shift and ALU)}\\ \text{(4) Data memory access}\\ \text{(5) Write back results}\end{array}\right\}\ \textbf{Integer Unit}$$

Integer Unit Organization



ARM8 Macrocell

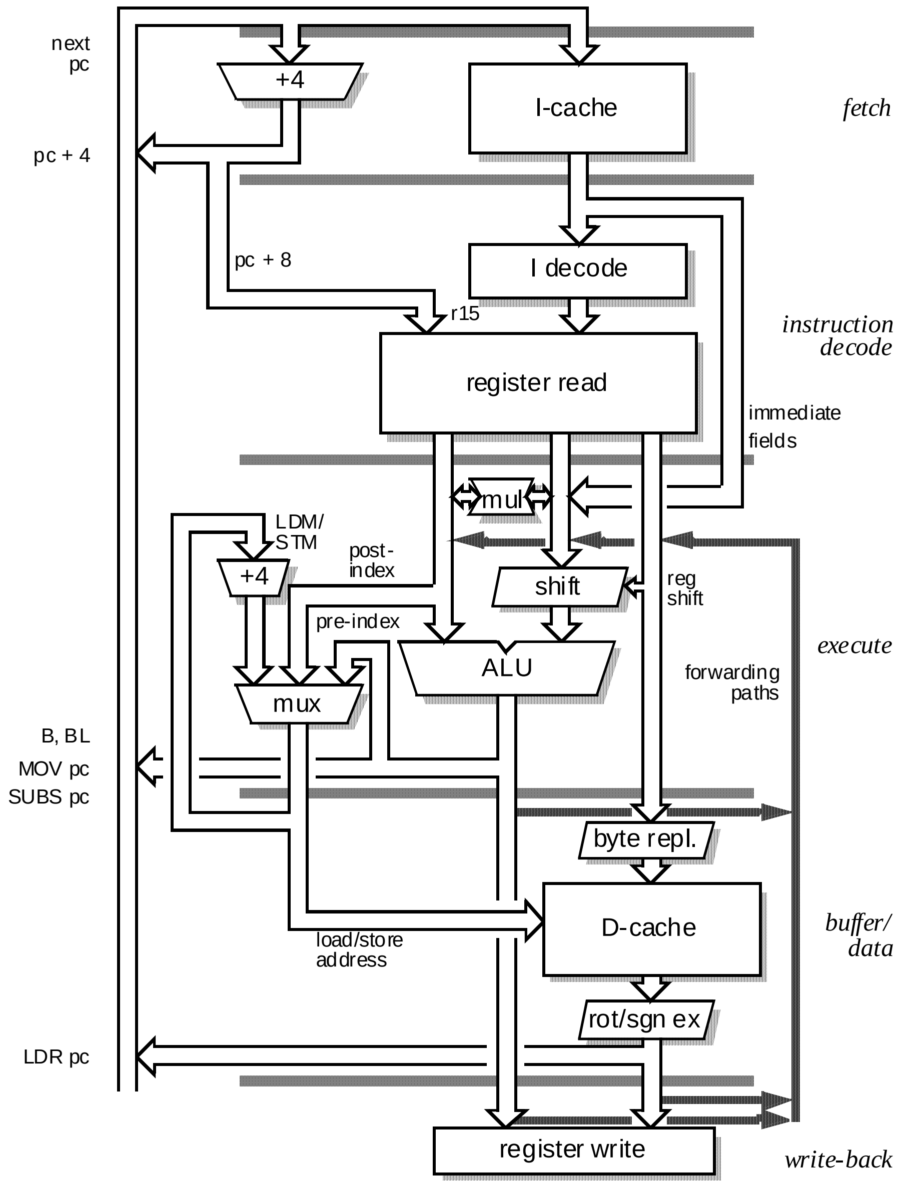


- ARM810
  - 8Kbyte unified instruction and data cache
  - Copy-back
  - Double-bandwidth
  - MMU
  - Coprocessor
  - Write buffer

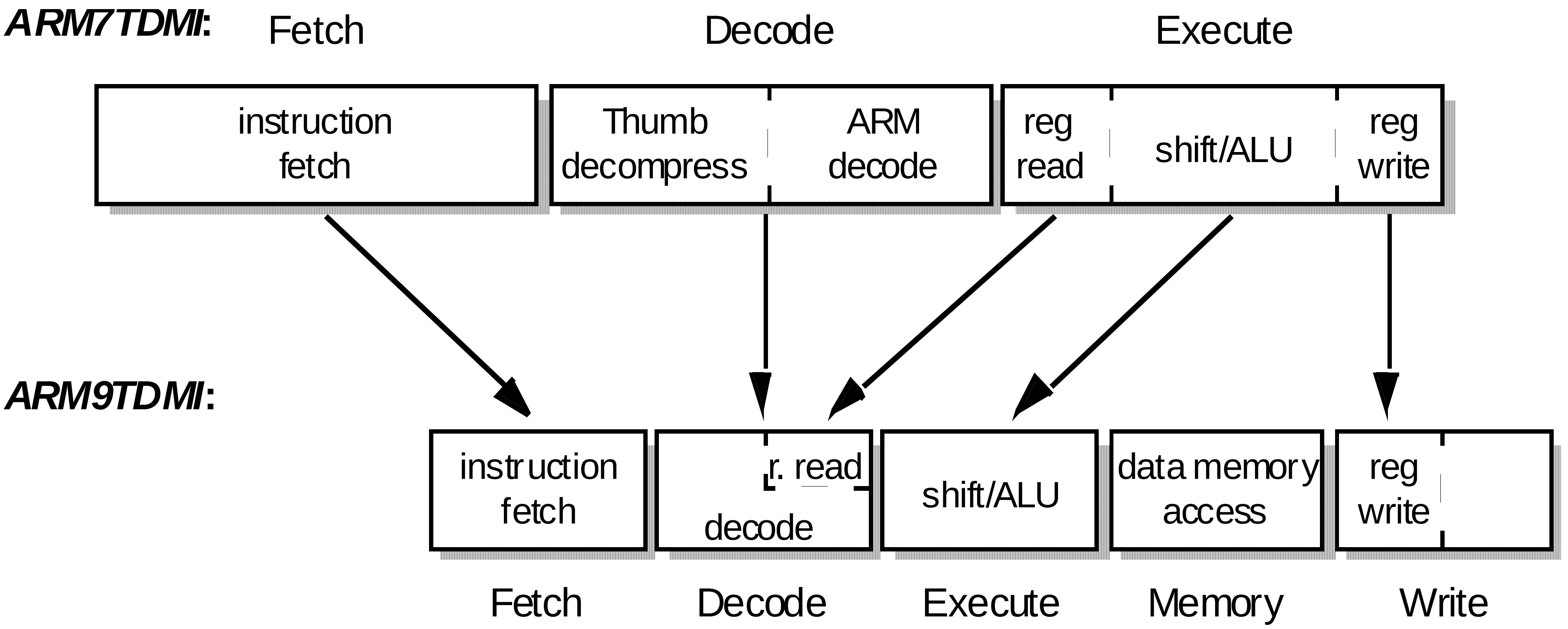
ARM9TDMI

- ❑ Harvard architecture
  - – Increases available memory bandwidth
    - • Instruction memory interface
    - • Data memory interface
  - – Simultaneous accesses to instruction and data memory can be achieved
- ❑ 5-stage pipeline
- ❑ Changes implemented to
  - – Improve CPI to ~1.5
  - – Improve maximum clock frequency

ARM9TDMI Organization



ARM9TDMI Pipeline Operations (1/2)



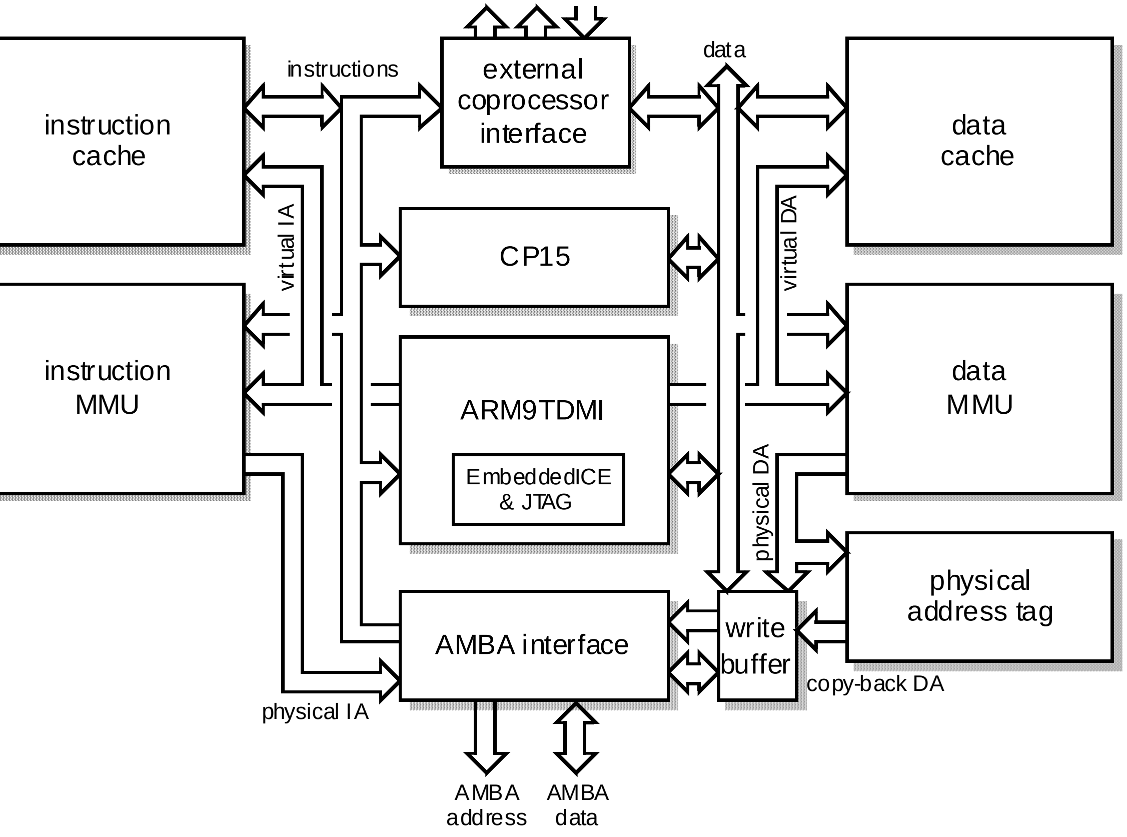
Not sufficient slack time to translate Thumb instructions into ARM instructions and then decode, instead the hardware decode both ARM and Thumb instructions directly

ARM9TDMI Pipeline Operations (2/2)

- Coprocessor support
  - Coprocessors: floating-point, digital signal processing, special-purpose hardware accelerator
- On-chip debugger
  - Additional features compared to ARM7TDMI
    - Hardware single stepping
    - Breakpoint can be set on exceptions
- ARM9TDMI characteristics

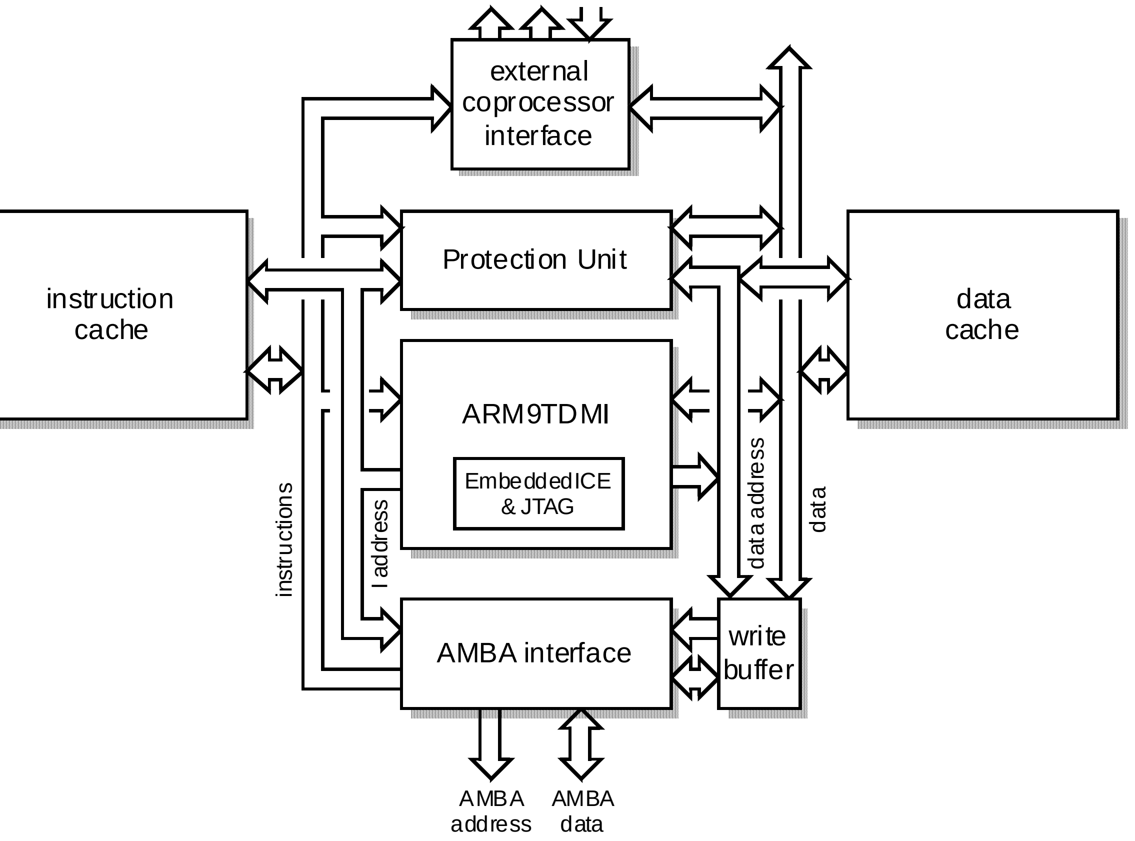
Process	0.25 um	**Transistors**	110,000	**MIPS**	220
Metal layers	3	**Core area**	2.1 mm^2	**Power**	150 mW
Vdd	2.5 V	**Clock**	0 to 200 MHz	**MIPS/W**	1500

ARM9TDMI Macrocells (1/2)



- ARM920T
  - 2 ☓ 16K caches
  - Full memory management unit supporting virtual addressing and memory protection
  - Write buffer

ARM9TDMI Macrocells (2/2)



- ARM 940T
  - 2 ☓ 4K caches
  - Memory protection Unit
  - Write buffer

ARM9E-S Family Overview

- ARM9E-S is based on an ARM9TDMI with the following extensions:
  - Single cycle 32*6 multiplier implementation
  - EmbeddedICE logic RT
  - Improved ARM/Thumb interworking
  - New 32*16 and 16*16 multiply instructions
  - New count leading zero instruction
  - New saturated math instructions

Architecture v5TE

- ARM946E-S
  - ARM9E-S core
  - Instruction and data caches, selectable sizes
  - Instruction and data RAMs, selectable sizes
  - Protection unit
  - AHB bus interface

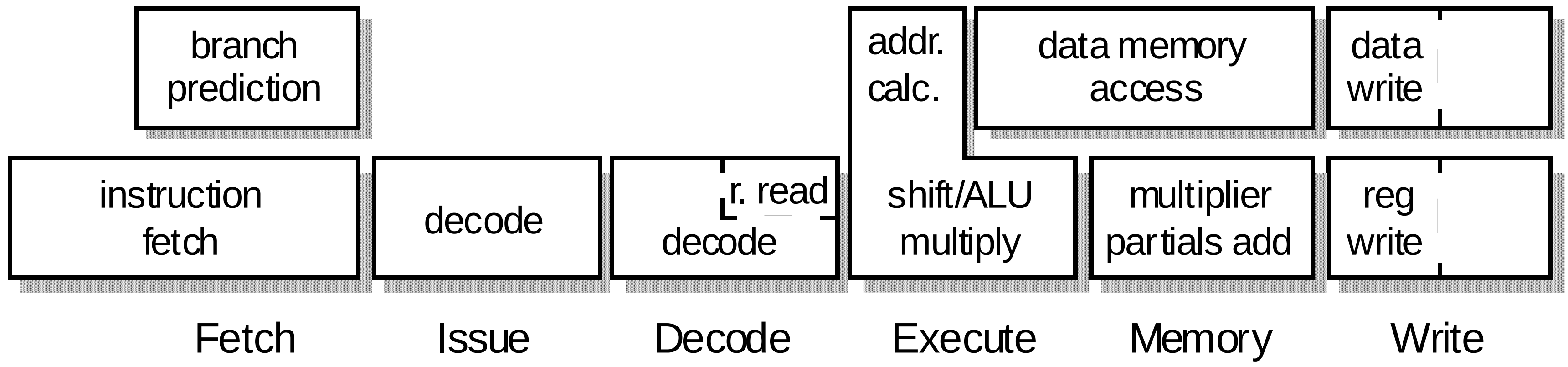
ARM10TDMI (1/2)

- ❑ Current high-end ARM processor core
- ❑ Performance on the same IC process

$$\begin{tikzcd}\text{ARM10TDMI} & \text{ARM9TDMI} \arrow[l, "\times 2"'] & \text{ARM7TDMI} \arrow[l, "\times 2"']\end{tikzcd}$$

- ❑ 300MHz, 0.25µm CMOS
- ❑ Increase clock rate

ARM10TDMI



ARM10TDMI (2/2)

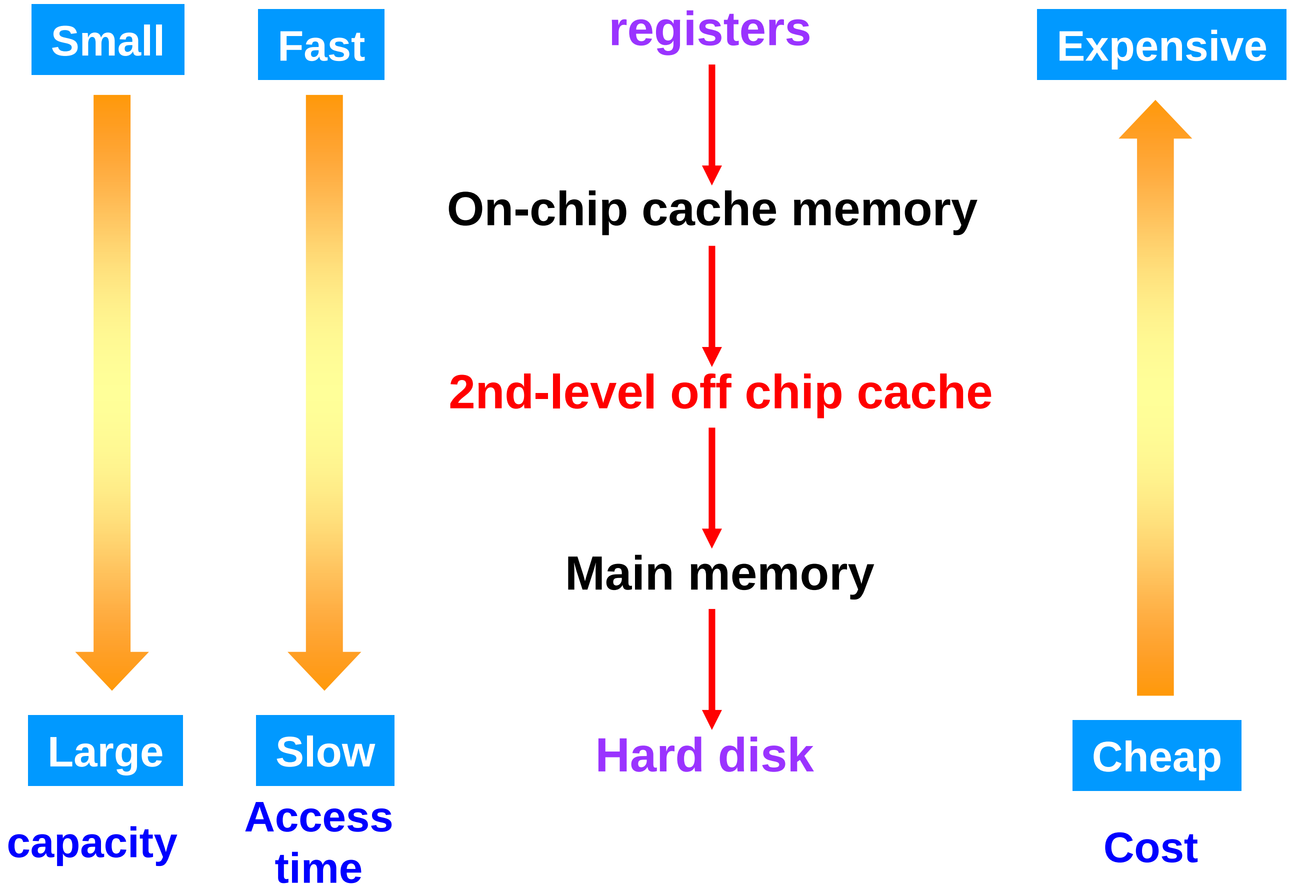
- Reduce CPI
  - Branch prediction
  - Non-blocking load and store execution
  - 64-bit data memory → transfer 2 registers in each cycle

ARM1020T Overview

- ❑ Architecture v5T
  - ARM1020E will be v5TE
- ❑ CPI ~ 1.3
- ❑ 6-stage pipeline
- ❑ Static branch prediction
- ❑ 32KB instruction and 32KB data caches
  - ‘hit under miss’ support
- ❑ 64 bits per cycle LDM/STM operations
- ❑ EmbeddedICE Logic RT-II
- ❑ Support for new VFPv1 architecture
- ❑ ARM10200 test chip
  - ARM1020T
  - VFP10
  - SDRAM memory interface
  - PLL

Memory Hierarchy

Memory Size and Speed



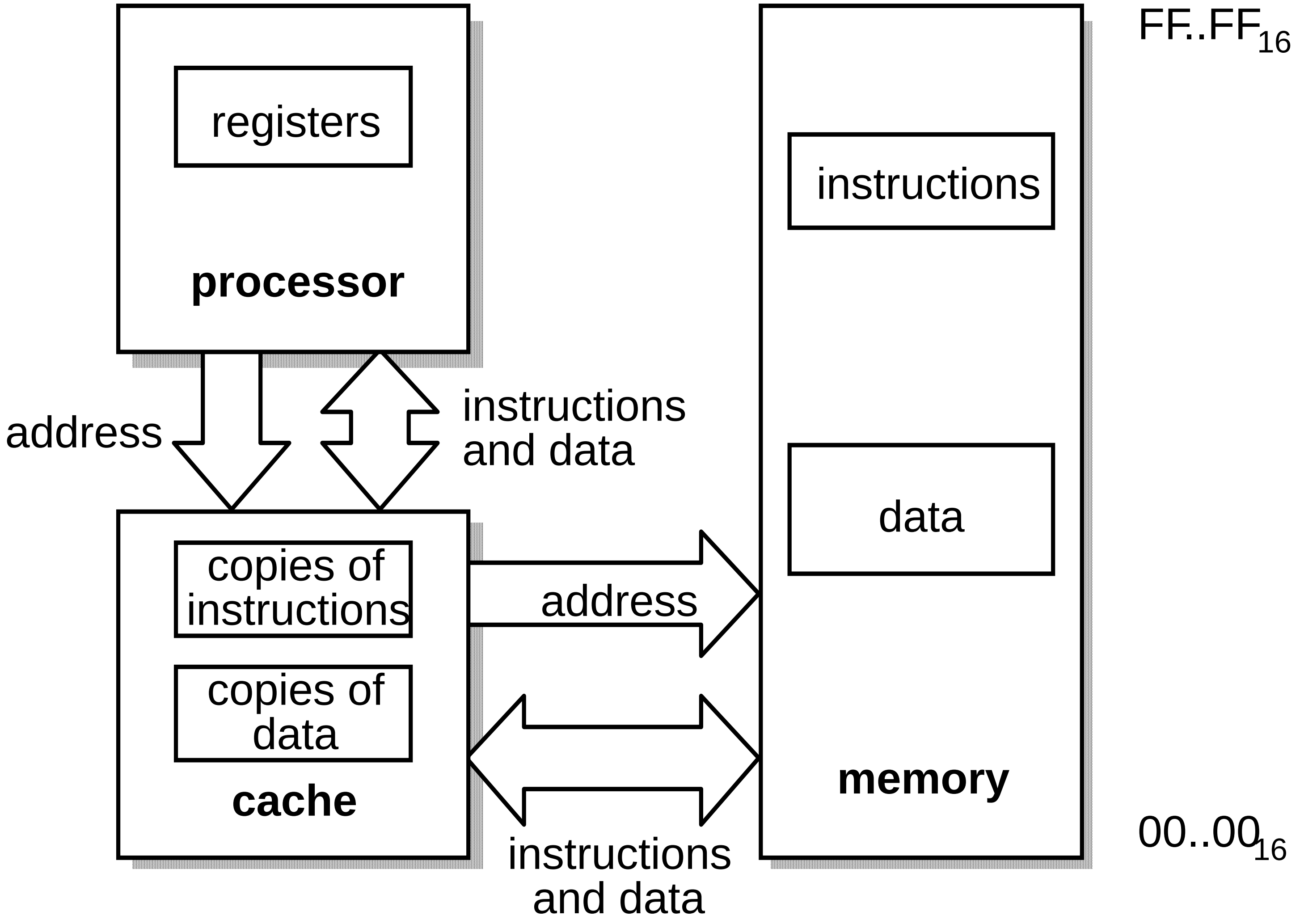
Caches (1/2)

- A cache memory is a small, very fast memory that retains copies of recently used memory values.
- It usually implemented on the same chip as the processor.
- Caches work because programs normally display the property of **locality**, which means that at any particular time they tend to execute the same instruction many times on the same areas of data.
- An access to an item which is in the cache is called a **hit**, and an access to an item which is not in the cache is a **miss**.

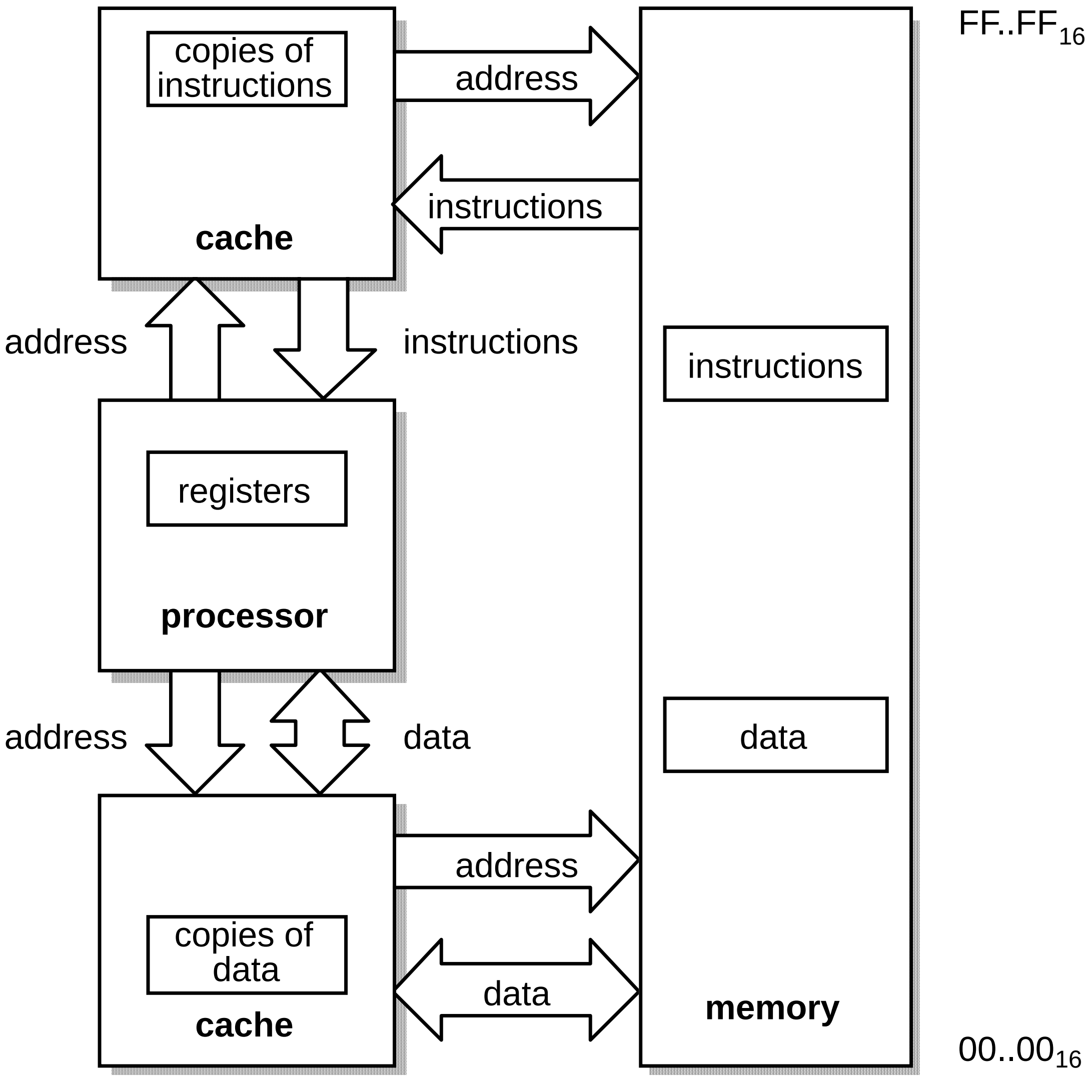
Caches (2/2)

- A processor can have one of the following two organizations:
  - A unified cache
    - This is a single cache for both instructions and data
  - Separate instruction and data caches
    - This organization is sometimes called a **modified Harvard** architectures

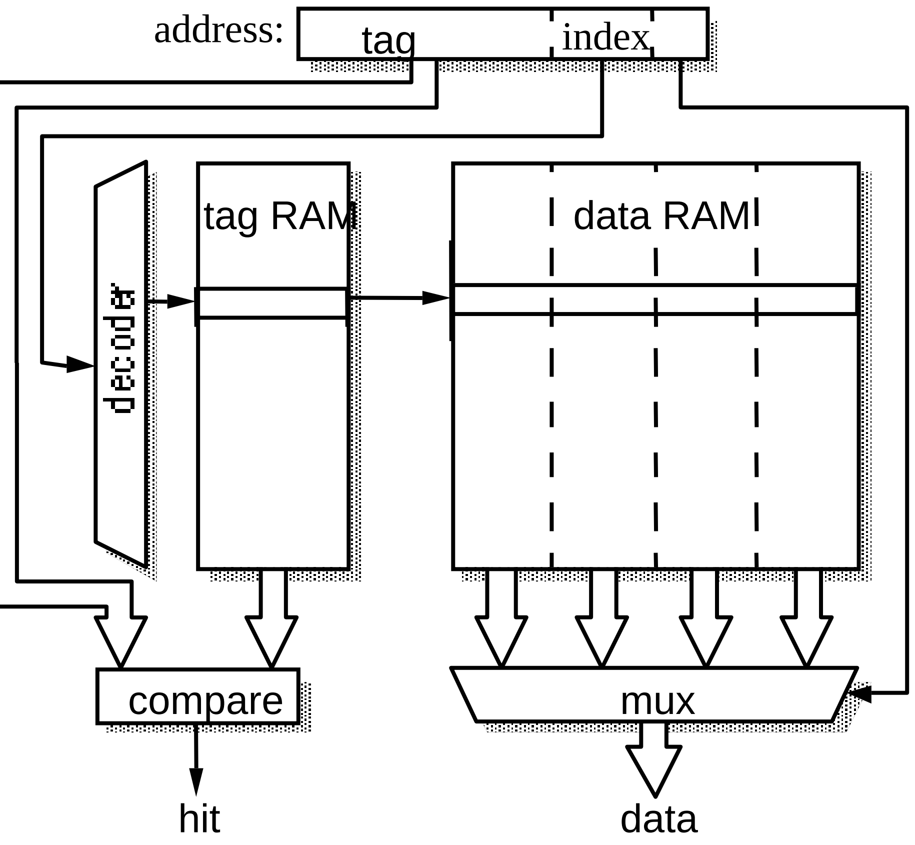
Unified instruction and data cache



Separate data and instruction caches

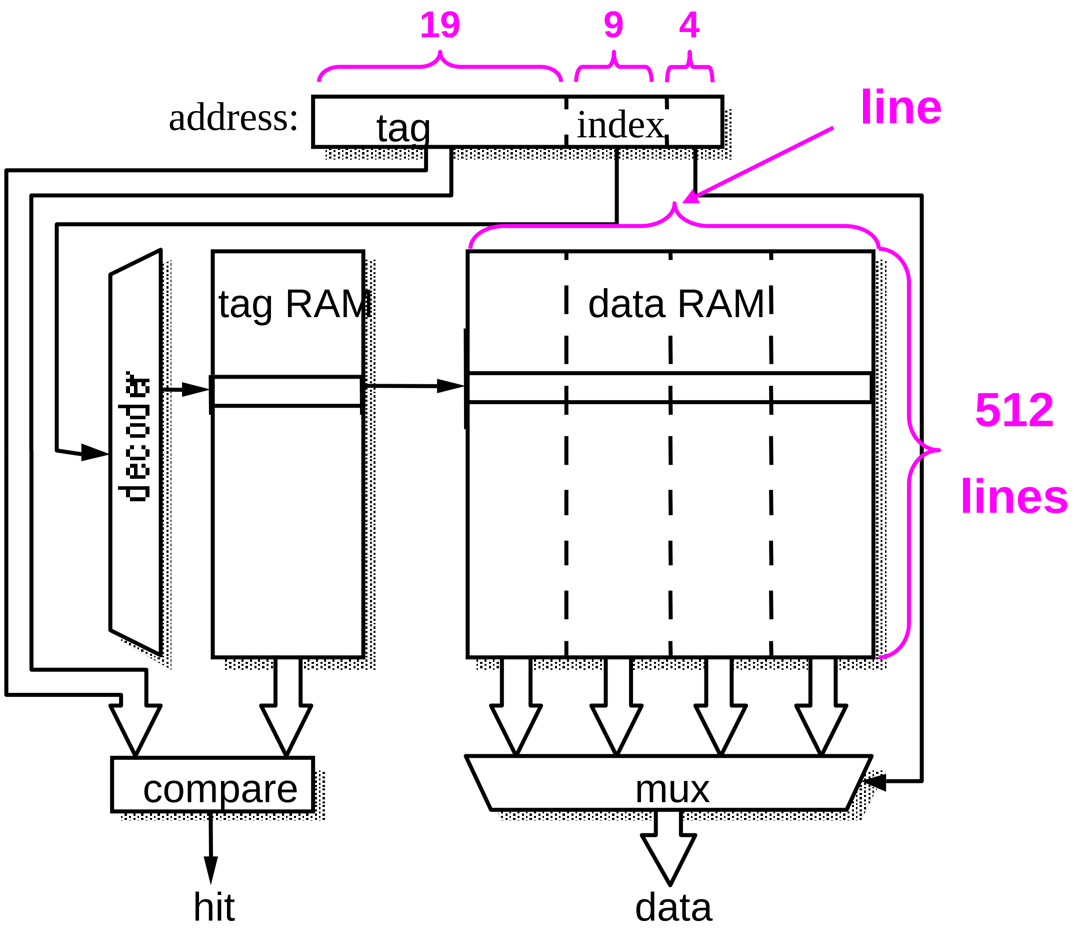


The direct-mapped cache



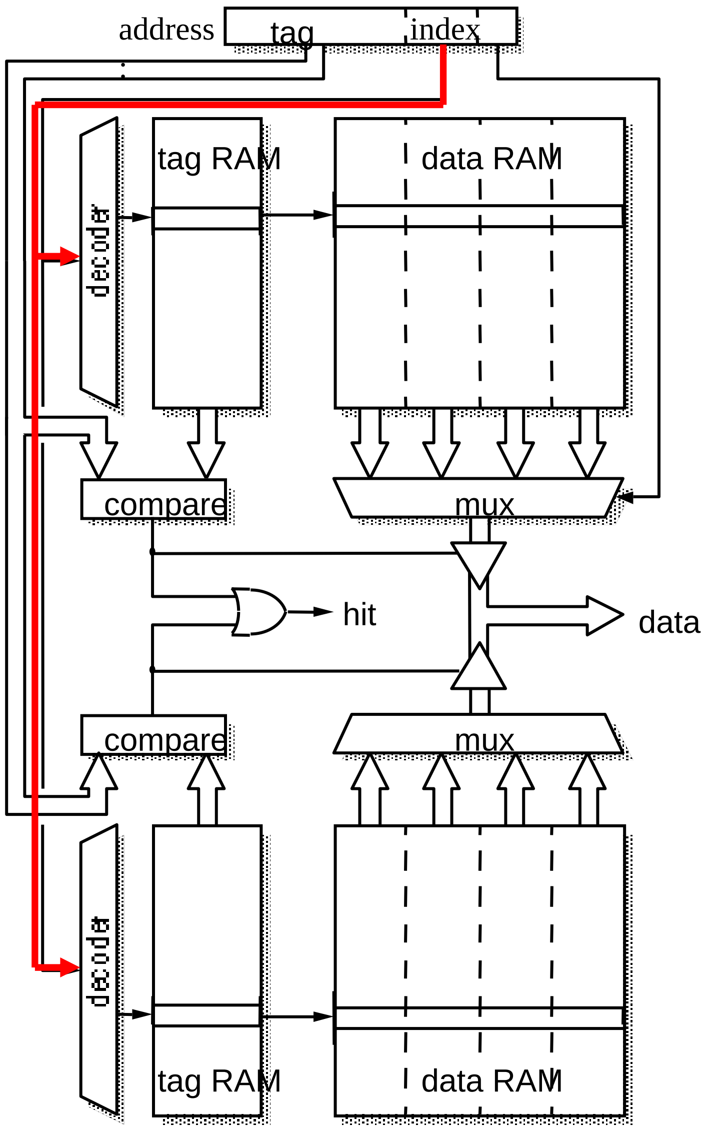
- ❑ The index address bits are used to access the cache entry
- ❑ The top address bit are then compared with the stored tag
- ❑ If they are equal, the item is in the cache
- ❑ The lowest address bit can be used to access the desired item with in the line.

Example



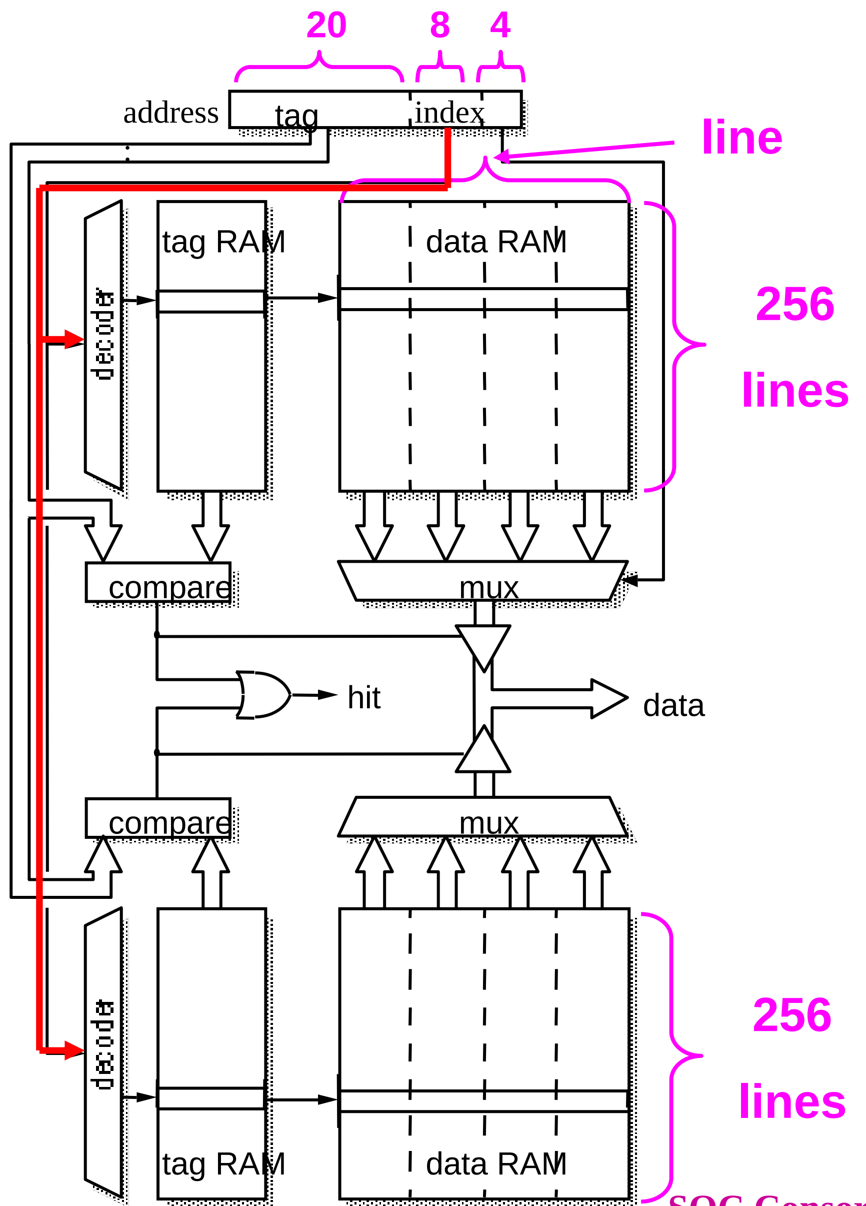
- The 8Kbytes of data in 16-byte lines. There would therefore be 512 lines
- A 32-bit address:
  - 4 bits to address bytes within the line
  - 9 bits to select the line
  - 19-bit tag

The set-associative cache



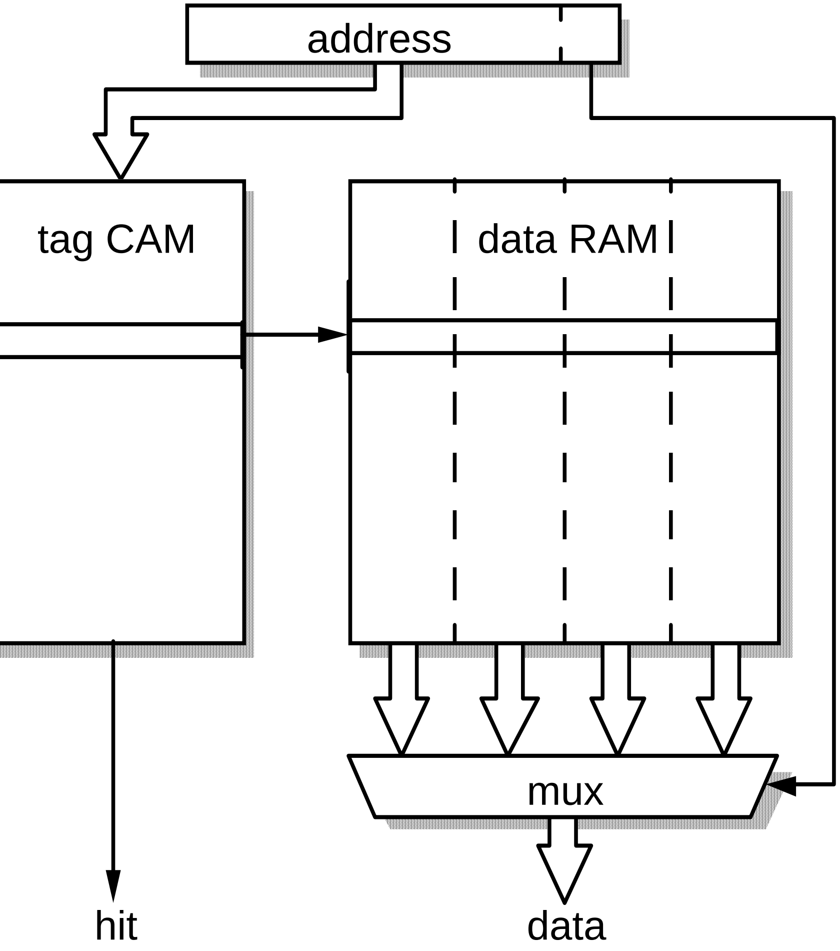
- A 2-way set-associative cache
- This form of cache is effectively two direct-mapped caches operating in parallel.

Example



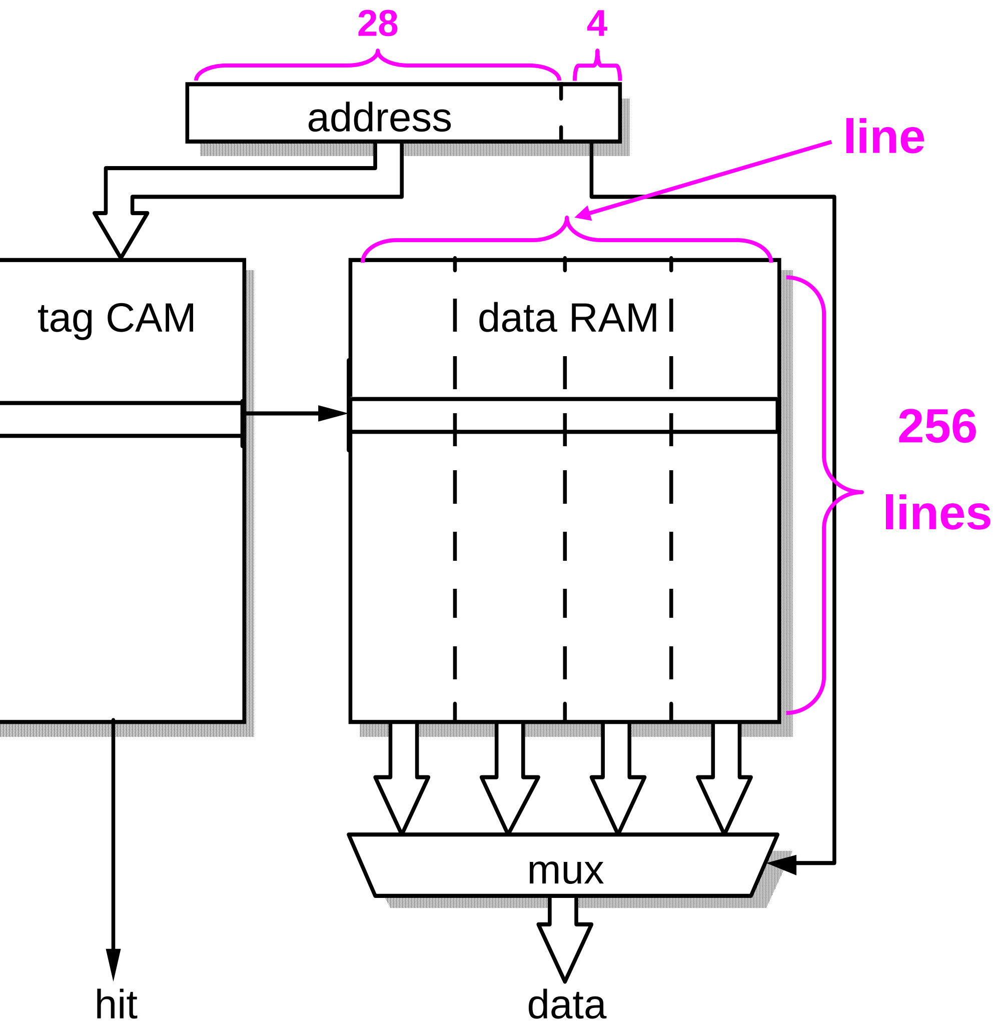
- The 8Kbytes of data in 16-byte lines. There would therefore be 256 lines in each half of the cache
- A 32-bit address:
  - 4 bits to address bytes within the line
  - 8 bits to select the line
  - 20-bit tag

Fully associative cache



- A **CAM** (Content Addressed Memory) cell is a RAM cell with an inbuilt comparator, so a CAM based tag store can perform a parallel search to locate an address in any location
- The address bit are compared with the stored tag
- If they are equal, the item is in the cache
- The lowest address bit can be used to access the desired item with in the line.

Example



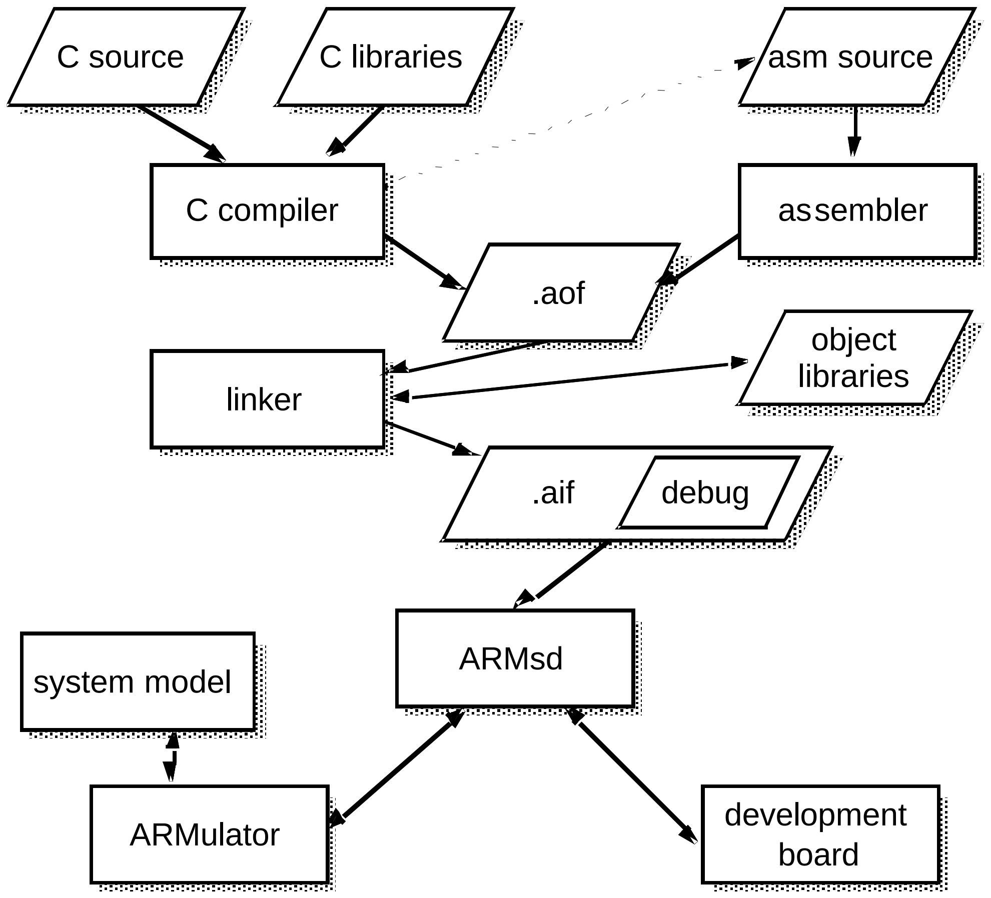
- The 8Kbytes of data in 16-byte lines. There would therefore be 512 lines
- A 32-bit address:
  - 4 bits to address bytes within the line
  - 28-bit tag

Write Strategies

- ❑ **Write-through**
  - – All write operations are passed to main memory
- ❑ **Write-through with buffered write**
  - – All write operations are still passed to main memory and the cache updated as appropriate, but instead of slowing the processor down to main memory speed the write address and data are stored in a **write buffer** which can accept the write information at high speed.
- ❑ **Copy-back (write-back)**
  - – No kept coherent with main memory

Software Development

ARM Tools



aof: ARM object format

aif: ARM image format

- ARM software development – *ADS*
- ARM system development – *ICE and trace*
- ARM-based SoC development – *modeling, tools, design flow*

ARM Development Suite (ADS), ARM Software Development Toolkit (SDT) (1/3)

- Develop and debug C/C++ or assembly language program
- *armcc* ARM C compiler

armcc	ARM C compiler
armcpp	ARM C++ compiler
tcc	Thumb C compiler
tcpp	Thumb C++ compiler
armasm	ARM and Thumb assembler
armlink	ARM linker
armsd	ARM and Thumb symbolic debugger

ARM Development Suite (ADS), ARM Software Development Toolkit (SDT) (2/3)

❑ *.aof*	ARM object format file
.aif	ARM image format file

❑ The .aif file can be built to include the debug tables

– ARM symbolic debugger, ARMsd

❑ ARMsd can load, run and debug programs either on hardware such as the ARM development board or using the software emulation of the ARM

❑ AXD (ARM eXtended Debugger)

– ARM debugger for Windows and Unix with graphics user interface

– Debug C, C++, and assembly language source

CodeWarrior IDE

– Project management tool for windows

ARM Development Suite (ADS), ARM Software Development Toolkit (SDT) (3/3)

- Utilities

armprof	ARM profiler
Flash downloader	download binary images to Flash memory on a development board

- Supporting software
  - *ARMulator* ARM core simulator
    - Provide instruction accurate simulation of ARM processors and enable ARM and Thumb executable programs to be run on non-native hardware
    - Integrated with the ARM debugger
  - *Angle* ARM debug monitor
    - Run on target development hardware and enable you to develop and debug applications on ARM-based hardware

ARM C Compiler

- ❑ Compiler is compliant with the ANSI standard for C
- ❑ Supported by the appropriate library of functions
- ❑ Use ARM Procedure Call Standard, APCS for all external functions
  - – For procedure entry and exit
- ❑ May produce assembly source output
  - – Can be inspected, hand optimized and then assembled sequentially
- ❑ Can also produce Thumb codes

Linker

- Take one or more object files and combine them
- Resolve symbolic references between the object files and extract the object modules from libraries
- Normally the linker includes debug tables in the output file

ARM Symbolic Debugger

- A front-end interface to debug program running either under emulator (on the ARMulator) or remotely on a ARM development board (via a serial line or through JTAG test interface)
- ARMsd allows an executable program to be loaded into the ARMulator or a development board and run. It allows the setting of
  - Breakpoints, addresses in the code
  - Watchpoints, memory address if accessed as data address
    - Cause exception to halt so that the processor state can be examined

ARM Emulator (1/2)

- ARMulator is a suite of programs that models the behavior of various ARM processor cores in software on a host system
- It operates at various levels of accuracy
  - – Instruction accuracy
  - – Cycle accuracy
  - – Timing accuracy
    - Instruction count or number of cycles can be measured for a program
    - Performance analysis
- Timing accuracy model is used for cache, memory management unit analysis, and so on

ARM Emulator (2/2)

- ARMulator supports a C library to allow complete C programs to run on the simulated system
- To run software on ARMulator, through ARM symbolic debugger or ARM GUI debuggers, AXD
- It includes
  - Processor core models which can emulate any ARM core
  - A memory interface which allows the characteristics of the target memory system to be modeled
  - A coprocessor interface that supports custom coprocessor models
  - An OS interface that allows individual system calls to be handled

ARM Development Board

- A circuit board including an ARM core (e.g. ARM7TDMI), memory component, I/O and electrically programmable devices
- It can support both hardware and software development before the final application-specific hardware is available

Summary (1/2)

- ARM7TDMI
  - Von Neumann architecture
  - 3-stage pipeline
  - CPI ~ 1.9
- ARM9TDMI, ARM9E-S
  - Harvard architecture
  - 5-stage pipeline
  - CPI ~ 1.5
- ARM10TDMI
  - Harvard architecture
  - 6-stage pipeline
  - CPI ~ 1.3

Summary (2/2)

- Cache
  - Direct-mapped cache
  - Set-associative cache
  - Fully associative cache
- Software Development
  - CodeWarrior
  - AXD

References

[1] http://twins.ee.nctu.edu.tw/courses/ip_core_02/index.html

[2] **ARM System-on-Chip Architecture** by S.Furber, Addison Wesley Longman: ISBN 0-201-67519-6.

[3] www.arm.com