Chapter 2

ARM Processor Core and Instruction Sets

Prof. Tian-Sheuan Chang

Outline

- Processor programming model
- 32-bit instruction set
- 16-bit instruction set
- ARM processor core
- Software development

Processor Programming Model

ARM Ltd

- ARM was originally developed at Acron Computer Limited of Cambridge, England between 1983 and 1985
  - 1980, RISC concept at Stanford and Berkeley universities
  - first RISC processor for commercial use
- 1990 Nov, ARM Ltd was founded
- ARM cores

  licensed to partners who fabricate and sell to customers
- Technologies assist to design in the ARM application
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc…

ARM Architecture vs. Berkeley RISC

- Features used
  - load/store architecture
  - fixed-length 32-bit instructions
  - 3-address instruction formats
- Features unused
  - register windows ⇒ costly
    use shadow registers in ARM
  - delayed branch ⇒ not well to superscalar
    badly with branch prediction
  - single-cycle execution of all instructions
    most single-cycle
  - memory access
    multiple cycles when no separate data and instruction memory support
    auto-indexing addressing modes

Data Size and Instruction set

- ARM processor is a 32-bit architecture
- Most ARM's implement two instruction sets
  - 32-bit ARM instruction set
  - 16-bit Thumb instruction set

Data Types

- ARM processor supports 6 data types
  - 8-bits signed and unsigned bytes
  - 16-bits signed and unsigned half-words, aligned on 2-byte boundaries
  - 32-bits signed and unsigned words, aligned on 4-byte boundaries
- ARM instructions are all 32-bit words, word-aligned

  Thumb instructions are half-words, aligned on 2-byte boundaries
- Internally all ARM operations are on 32-bit operands; the shorter data types are only supported by data transfer instructions. When a byte is loaded form memory, it is zero- or sign-extended to 32 bits
- ARM coprocessor supports floating-point values

Programming Model

- Each instruction can be viewed as performing a defined transformation of the states
  - visible registers
  - invisible registers
  - system memory
  - user memory

Processor Modes

- ARM has seven basic operating modes
- Mode changes by software control or external interrupts

CPRS[4:0]	Mode	Use	Registers
10000	User	Normal user code	User
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irp
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system	user

Privileged Modes

- Most programs operate in user mode. ARM has other privileges operating modes which are used to handle exceptions, supervisor calls (software interrupts), and system mode
- More access rights to memory systems and coprocessors
- Current operating mode is defined by CPSR[4:0]

Supervisor Mode

- Having some protective privileges
- System-level function (transaction with the outside world) can be accessed through specified supervisor calls
- Usually implemented by software interrupt (SWI)

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 bank is accessible

  each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (stack pointer, SP) and r14 (link register, LR)
  - the program counter, r15 (PC)
  - the current program status register, CPSR

  privileged modes (except System) can access
  - a particular SPSR (saved program status register)

Register Banking

Current Visible Registers

User Mode

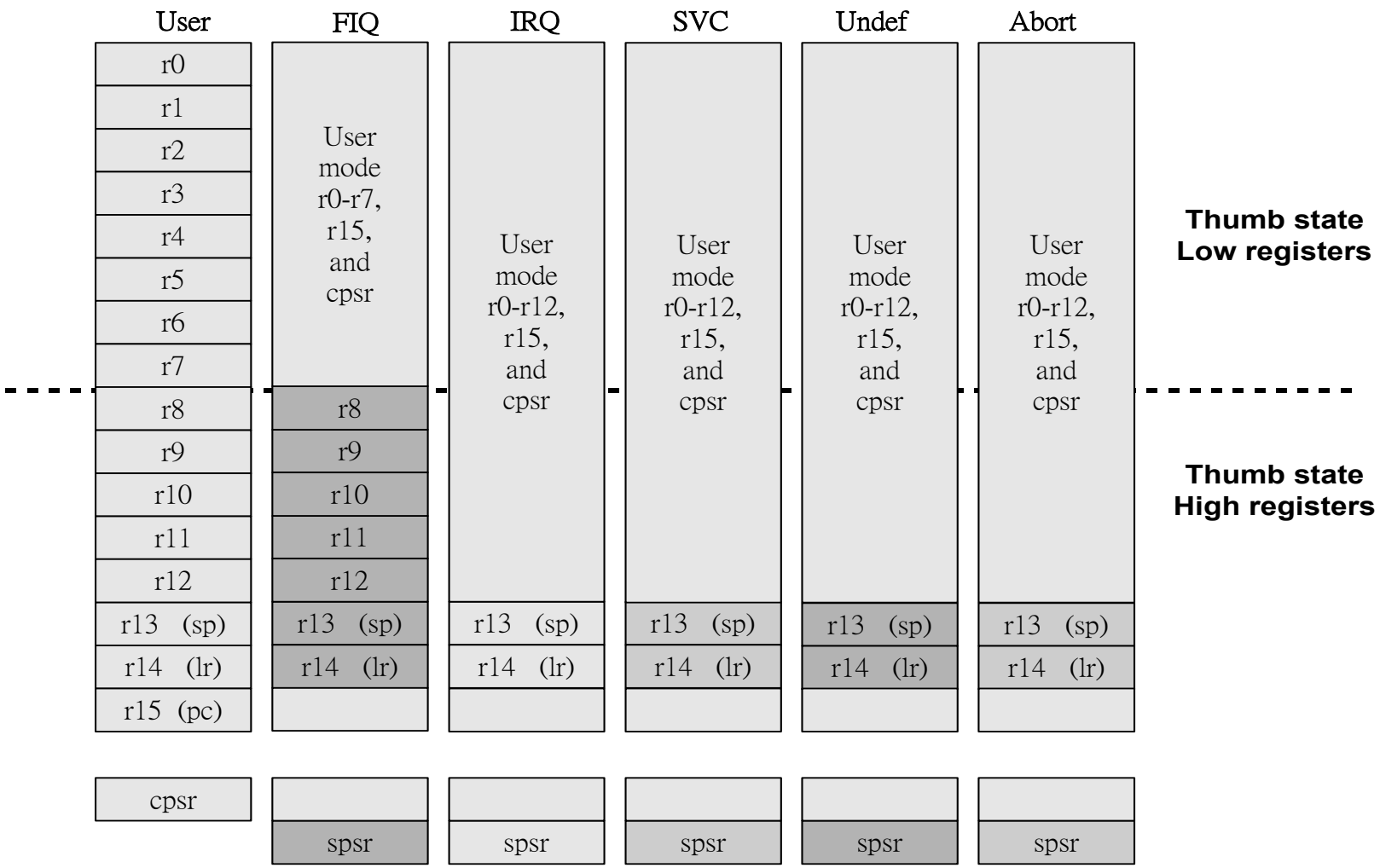
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr

Banked out Registers

FIQ	IRQ	SVC	Undef	Abort
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	spsr

Registers Organization Summary

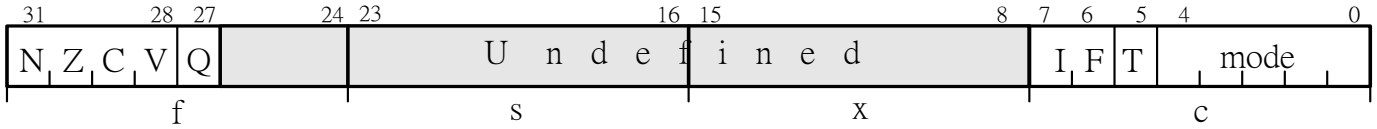


Note : System mode uses the User mode reigster set

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 are must be halfword-aligned
  - therefore the PC value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte-aligned)

Program Status Registers (CPSR)



- 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 only
  - indicates if saturation has occurred during certain operations
- 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

SPSRs

- Each privileged mode (except system mode) has associated with it a Save Program Status Register, or SPSR
- This SPSR is used to save the state of CPSR (Current Program Status Register) when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed
- Often the SPSR may be untouched from the time the privileged mode is entered to the time it is used to restore the CPSR, but if the privileged supervisor calls to itself) then the SPSR must be copied into a general register and saved

Exceptions

- Exceptions are usually used to handle unexpected events which arise during the execution of a program, such as interrupts or memory faults, also cover software interrupts, undefined instruction traps, and the system reset
- Three groups :
  1. generated as the direct effect of executing an instruction software interrupts, undefined instructions, prefetch abort (memory fault)
  2. generated as the side-effect of an instruction

     data aborts
  3. generated externally

     reset, IRQ, FIQ

Exception Entry (1/2)

- When an exception arises, ARM completes the current instruction as best it can (except that reset exception terminates the current instruction immediately) and then departs from the current instruction sequence to handle the exception which starts from a specific location (exception vector)
- Processor performs the following sequence
  - change to the operating mode corresponding to the particular exception
  - save the address of the instruction following the exception entry instruction in r14 of the new mode
  - save the old value of CPSR in the SPSR of the new mode
  - disable IRQs by setting bit of the CPSR, and if the exception is a fast interrupt, disable further faster interrupt by setting bit of the CPSR

Exception Entry (2/2)

– force the PC to begin executing at the relevant vector address

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

- Normally the vector address contains a branch to the relevant routine, though the FIQ code can start immediately
- Two banked registers in each of the privilege modes are used to hold the return address and stack pointer

Exception Return (1/3)

- Once the exception has been handled, the user task is normally resumed
- The sequence is
  - any modified user registers must be restored from the handler'stack
  - CPSR must be restored from the appropriate SPSR
  - PC must be changed back to the relevant instruction address
- The last two steps happen atomically as part of a single instruction

Exception Return (2/3)

- When the return address has been kept in the banked r14
  - to return from a SWI or undefined instruction trap

    ```
    MOVS PC,r14
    ```

  - to return from an IRQ, FIQ or prefetch abort

    ```
    SUBS PC,r14,#4
    ```

  - To return from a data abort to retry the data access

    ```
    SUBS PC,r14,#8
    ```

  - ‘S’ signifies when the destination register is the PC

Exception Return (3/3)

- When the return address has been saved onto a stack

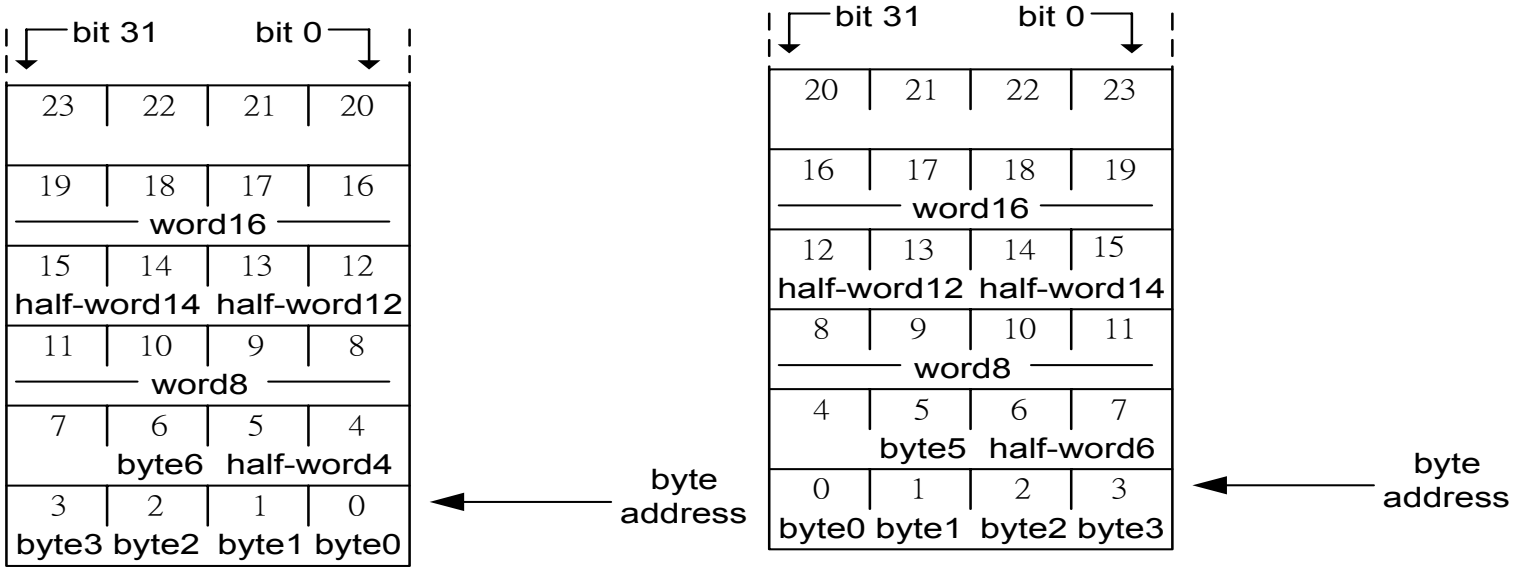
```
LDMFD r13!,{r0-r3,PC}^      ;restore and return
```

  – '^' indicates that this is a special form of the instruction the CPSR is restored at the same time that the PC is loaded from memory, which will always be the last item transferred from memory since the registers are loaded in increasing order

Exception Priorities

- Priority order
  1. reset (highest priority)
  2. data abort
  3. FIQ
  4. IRQ
  5. prefetch abort
  6. SWI, undefined instruction

Memory Organization



(a) Little-endian memory organization

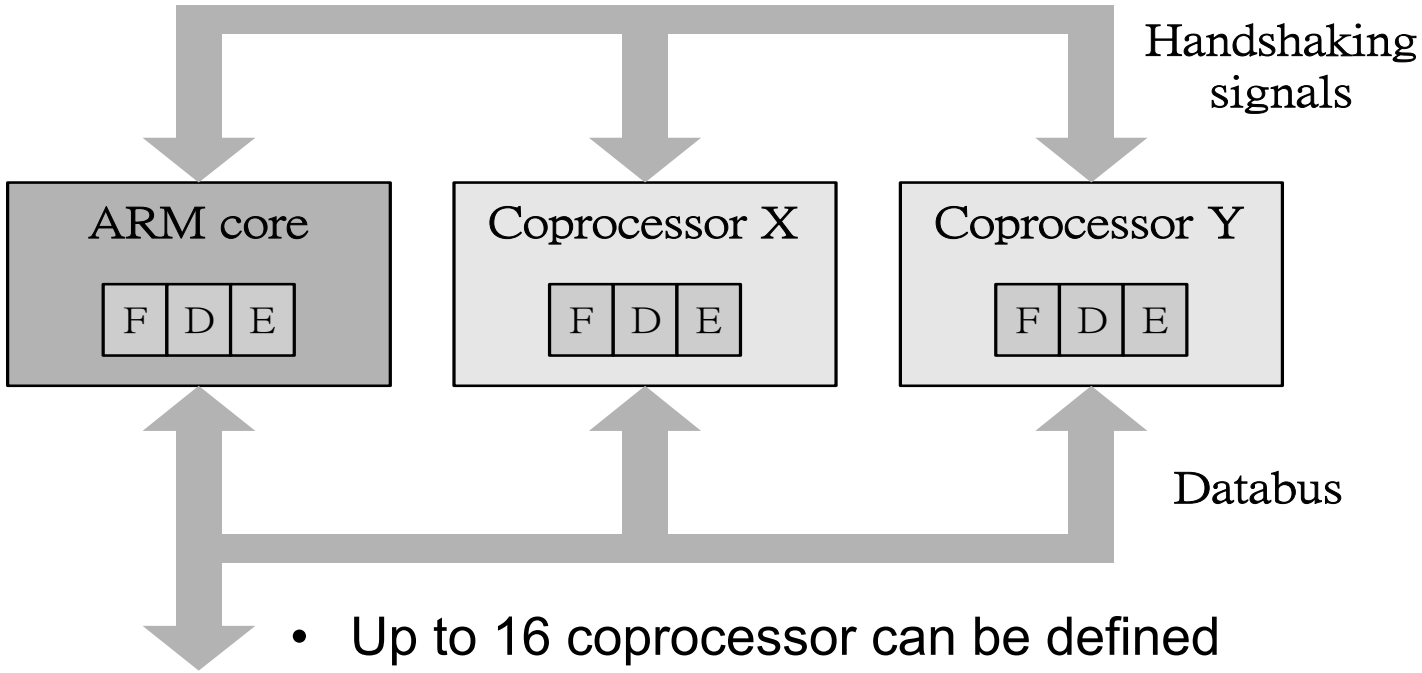
(b) Big-endian memory organization

- Word, half-word alignment (xxxxoo or xxxxxo)
- ARM can be set up to access data in either little-endian or big-endian format

Features of the ARM Instruction Set

- Load-store architecture
  - process values which are in registers
  - load, store instructions for memory data accesses
- 3-address data processing instructions
- Conditional execution of every instruction
- Load and store multiple registers
- Shift, ALU operation in a single instruction
- Open instruction set extension through the coprocessor instruction
- Very dense 16-bit compressed instruction set (Thumb)

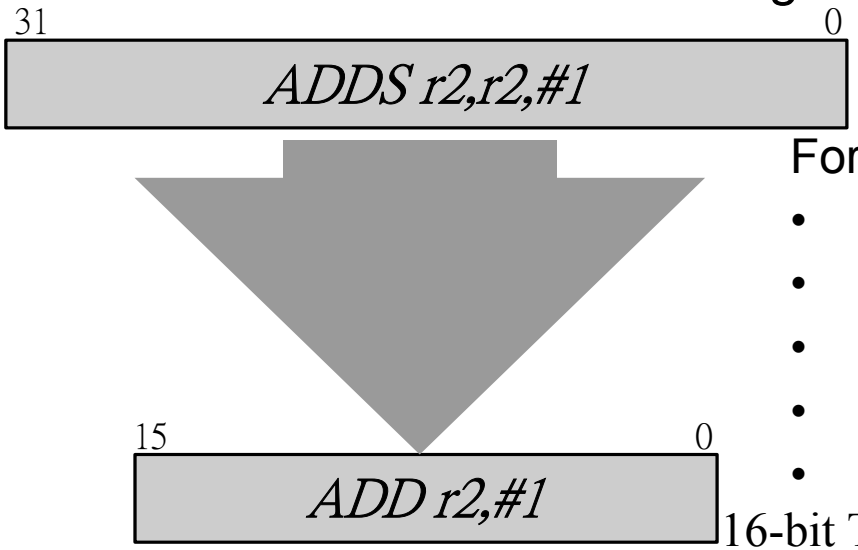
Coprocessors



- Up to 16 coprocessor can be defined
- Expands the ARM instruction set
- ARM uses them for “internal functions” so as not to enforce a particular memory map (eg cp15 is the ARM cache controller)
- Usually better for system designers to use memory mapped peripherals
  - easier to implement

Thumb

- Thumb is a 16-bit instruction set
  - optimized for code density from C code
  - improved performance from narrow memory
  - subset of the fumctionality of the ARM instruction set
- Core has two execution states – ARM and Thumb
  - switch between them using ***BX*** instruction



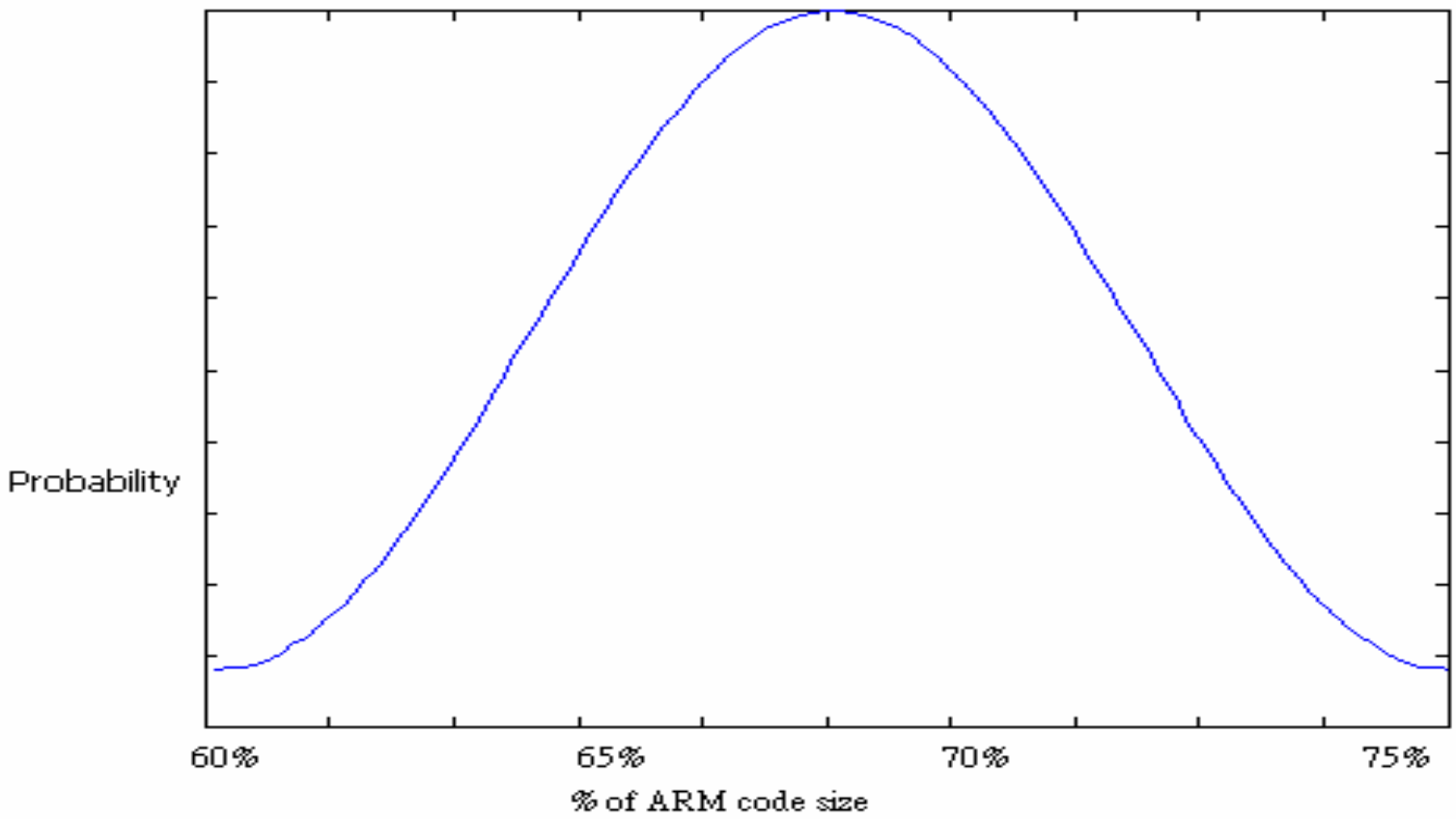
32-bit ARM instruction

For most instruction generated by compiler:

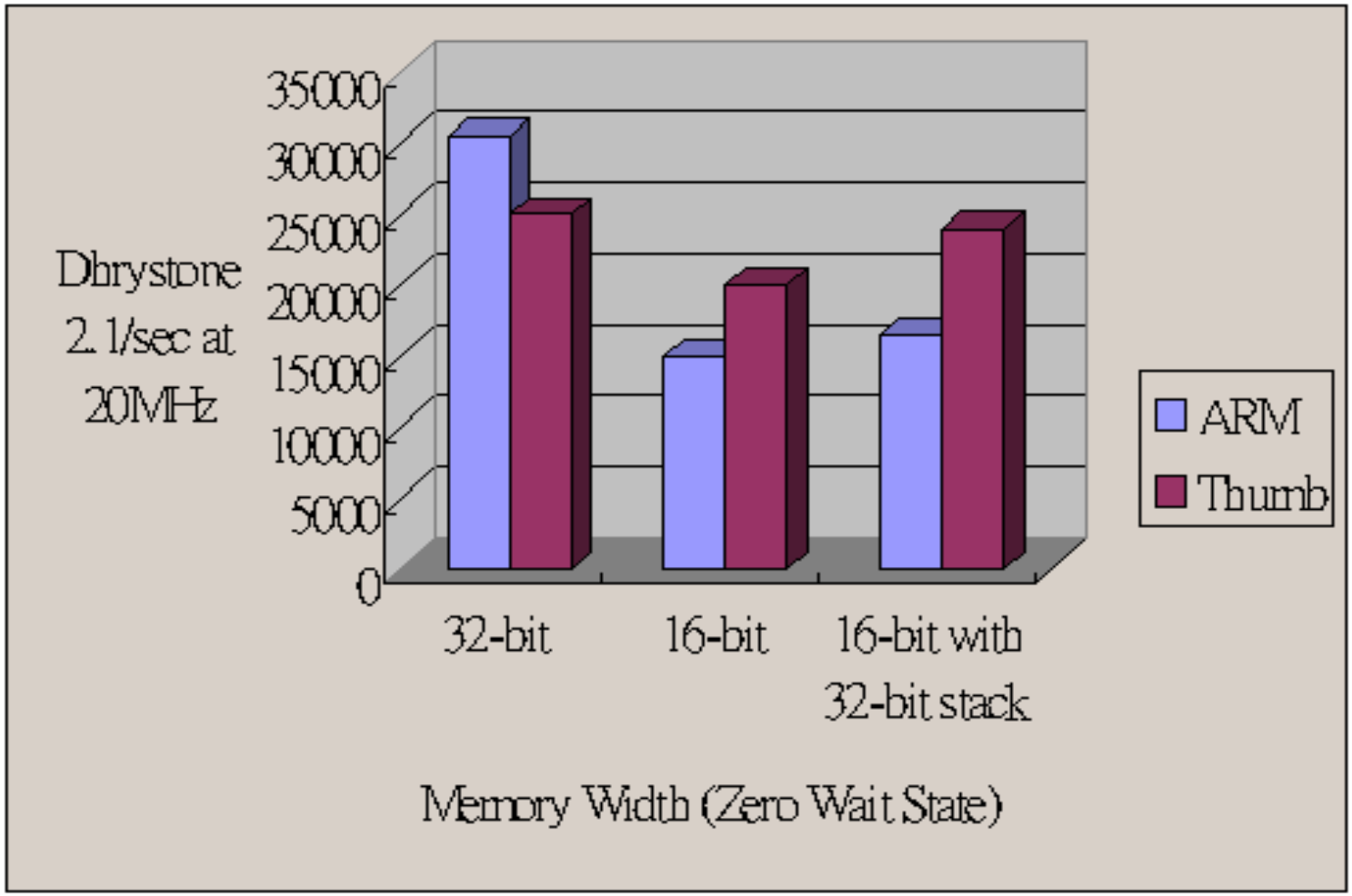
- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

16-bit Thumb instruction

Average Thumb Code Sizes



ARM and Thumb Performace



I/O System

- ARM handles input/output peripherals as memory-mapped with interrupt support
- Internal registers in I/O devices as addressable locations within ARM's memory map

  read and written using load-store instructions
- Interrupt by normal interrupt (IRQ)

  or fast interrupt (FIQ) higher priority

  input signals are level-sensitive and maskable
- May include Direct Memory Access (DMA) hardware

ARM Exceptions

- Supports interrupts, traps, supervisor calls
- When an exception occurs, the ARM:
  - copies CPSR into SPSR_<mode>
  - sets appropriate CPSR bits
    - if core currently in Thumb state then ARM state is entered
    - mode field bits
    - interrupt disable bits (if appropriate)
  - stores the return address in LR_<mode>
  - set pc to vector address
- To return, exception handler needs to:
  - restore CPSR from SPSR_<mode>
  - restore PC from LR_<mode>

  This can only be done in ARM state

0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

Vector table can be at 0xffff0000 on ARM720T and on ARM9/10 family devices

ARM Exceptions

- Exception handler use r13_<mode> which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers

ARM Processor Cores

- ARM Processor core + cache + MMU
  →ARM CPU cores
- ARM6 → ARM7 (3V operation, 50-100MHz for .25μ or .18 μ)

  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, 64-bit result

  I : embedded ICE hardware, give on-chip breakpoint and watchpoint support

ARM Processor Cores

- ARM 8 → ARM 9
  → ARM 10
- ARM 9
  - 5-stage pipeline (130 MHz or 200MHz)
  - using separate instruction and data memory ports
- ARM 10 (1998. Oct.)
  - high performance, 300 MHz
  - multimedia digital consumer applications
  - optional vector floating-point unit

ARM Architecture Versions (1/5)

- Version 1
  - the first ARM processor, developed at Acorn Computers Limited 1983-1985
  - 26-bit addressing, no multiply or coprocessor support
- Version 2
  - sold in volume in the Acorn Archimedes
  - 26-bit addressing, including 32-bit result multiply and coprocessor
- Version 2a
  - coprocessor 15 as the system control coprocessor to manage cache
  - add the atomic load store (SWP) instruction

ARM Architecture Versions (2/5)

- Version 3
  - first ARM processor designed by ARM Limited (1990)
  - ARM6 (macro cell)

    ARM60 (stand-alone processor)

    ARM600 (an integrated CPU with on-chip cache, MMU, write buffer)

    ARM610 (used in Apple Newton)
  - 32-bit addressing, separate CPSR and SPSRs
  - add the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode
- Version 3M
  - introduce the signed and unsigned multiply and multiply-accumulate instructions that generate the full 64-bit result

ARM Architecture Versions (3/5)

- Version 4
  - add the signed, unsigned half-word and signed byte load and store instructions
  - reserve some of SWI space for architecturally defined operations
  - system mode is introduced
- Version 4T
  - 16-bit Thumb compressed form of the instruction set is introduced

ARM Architecture Versions (4/5)

- Version 5T
  - introduced recently, a superset of version 4T adding the BLX, CLZand BRK instructions
- Version 5TE
  - add the signal processing instruction set extension

ARM Architecture Versions (5/5)

Core	Architecture
ARM1	v1
ARM2	v2
ARM2as, ARM3	v2a
ARM6, ARM600, ARM610	v3
ARM7, ARM700, ARM710	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
StrongARM, ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	V4T
ARM9E-S	v5TE
ARM10TDMI, ARM1020E	v5TE

32-bit Instruction Set

- ARM assembly language program
  - ARM development board or ARM emulator
- ARM instruction set
  - standard ARM instruction set
  - a compressed form of the instruction set, a subset of the full ARM instruction set is encoded into 16-bit instructions - Thumb instruction
  - some ARM cores support instruction set extensions to enhance signal processing capabilities

Instructions

- Data processing instructions
- Data transfer instructions
- Control flow instructions

ARM Instruction Set Summary (1/4)

Mnemonic	Instruction	Action
ADC	Add with carry	Rd:=Rn+Op2+Carry
ADD	Add	Rd:=Rn+Op2
AND	AND	Rd:=Rn AND Op2
B	Branch	R15:=address
BIC	Bit Clear	Rd:=Rn AND NOT Op2
BL	Branch with Link	R14:=R15 R15:=address
BX	Branch and Exchange	R15:=Rn T bit:=Rn[0]
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags:=Rn+Op2
CMP	Compare	CPSR flags:=Rn-Op2

ARM Instruction Set Summary (2/4)

Mnemonic	Instruction	Action
EOR	Exclusive OR	Rd:=Rn^Op2
LDC	Load Coprocessor from memory	(Coprocessor load)
LDM	Load multiple registers	Stack Manipulation (Pop)
LDR	Load register from memory	Rd:=(address)
MCR	Move CPU register to coprocessor register	CRn:=rRn{<op>cRm}
MLA	Multiply Accumulate	Rd:=(Rm*Rs)+Rn
MOV	Move register or constant	Rd:=Op2
MRC	Move from coprocessor register to CPU register	rRn:=cRn{<op>cRm}
MRS	Move PSR status/flags to register	Rn:=PSR
MSR	Move register to PSR status/flags	PSR:=Rm

ARM Instruction Set Summary (3/4)

Mnemonic	Instruction	Action
MUL	Multiply	Rd:=Rm*Rs
MVN	Move negative register	Rd:=~Op2
ORR	OR	Rd:=Rn OR Op2
RSB	Reverse Subtract	Rd:=Op2-Rn
RSC	Reverse Subtract with Carry	Rd:=Op2-Rn-1+Carry
SBC	Subtract with Carry	Rd:=Rn-Op2-1+Carry
STC	Store coprocessor register to memory	address:=cRn
STM	Store Multiple	Stack manipulation (Push)

ARM Instruction Set Summary (4/4)

Mnemonic	Instruction	Action
STR	Store register to memory	<address>:=Rd
SUB	Subtract	Rd:=Rn-Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd:=[Rn] [Rn]:=Rm
TEQ	Test bitwise equality	CPSR flags:=Rn EOR Op2
TST	Test bits	CPSR flags:=Rn AND Op2

ARM Instruction Set Format

31 30 29 28	27	26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	
Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2						*Data Processing / PSR Transfer*
Cond	0	0	0	0 0 0 A	S	Rd	Rn	Rs	1	0	0	1	Rm	*Multiply*
Cond	0	0	0	0 1 U A	S	RdHi	RdLo	Rn	1	0	0	1	Rm	*Multiply Long*
Cond	0	0	0	1 0 B 0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm	*Single Data Swap*
Cond	0	0	0	1 0 0 1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn	*Branch and Exchange*
Cond	0	0	0	P U 0 W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm	*Halfword Data Transfer: register offset*
Cond	0	0	0	P U 1 W	L	Rn	Rd	Offset	1	S	H	1	Offset	*Halfword Data Transfer: immediate offset*
Cond	0	1	I	P U B W	L	Rn	Rd	Offset						*Single Data Transfer*
Cond	0	1	1									1		*Undefined*
Cond	1	0	0	P U S W	L	Rn	Register List							*Block Data Transfer*
Cond	1	0	1	L	Offset									*Branch*
Cond	1	1	0	P U N W	L	Rn	CRd	CP#	Offset					*Coprocessor Data Transfer*
Cond	1	1	1	0	CP Opc	CRn	CRd	CP#	CP			0	CRm	*Coprocessor Data Operation*
Cond	1	1	1	0	CP Opc	L	CRn	Rd	CP#	CP		1	CRm	*Coprocessor Register Transfer*
Cond	1	1	1	1	Ignored by processor									*Software Interrupt*
31 30 29 28	27	26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0	

Data Processing Instructions

- Consist of
  - arithmetic (ADD, SUB, RSB)
  - logical (BIC, AND)
  - compare (CMP, TST)
  - register movement (MOV, MVN)

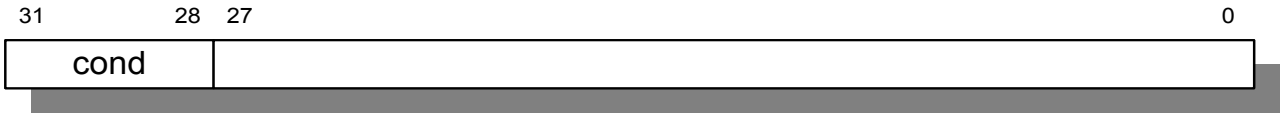
- All operands are 32-bit wide; come from registers or specified as literal in the instruction itself
- Second operand sent to ALU via barrel shifter
- 32-bit result placed in register; long multiply instruction produces 64-bit result
- 3-address instruction format

Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - all instructions contain a condition field which determines whether the CPU will execute them
  - non-executed instructions still take up 1 cycle
    - to allow other stages in the pipeline to complete
- This reduces the number of branches which would stall the pipeline
  - allows very dense in-line code
  - the time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed

Conditional Execution

Each of the 16 values causes the instruction to be executed or skipped according to the N, Z, C, V flags in the CPSR



Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Using and Updating the Condition Field

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - for example an add instruction takes the form:
    - `ADD r0,r1,r2 ;r0:=r1+r2 (ADDAL)`
  - to execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ;r0:=r1+r2 iff zero flag is set`
- By default, data processing operations do not affect the condition flags
  - with comparison instructions this is the only effect
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  - for exammple to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ;r0:=r1+r2 and set flags`

Data Processing Instructions

- Simple register operands
- Immediate operands
- Shifted register operands
- Multiply

Simple Register Operands (1/2)

- Arithmetic Operations

```
ADD r0,r1,r2  ;r0:=r1+r2
ADC r0,r1,r2  ;r0:=r1+r2+C
SUB r0,r1,r2  ;r0:=r1-r2
SBC r0,r1,r2  ;r0:=r1-r2+C-1
RSB r0,r1,r2  ;r0:=r2-r1,reverse subtraction
RSC r0,r1,r2  ;r0:=r2-r1+C-1
```

  – by default, data processing operations do not affect the condition flags

- Bit-wise Logical Operations

```
AND r0,r1,r2  ;r0:=r1 AND r2
ORR r0,r1,r2  ;r0:=r1 OR r2
EOR r0,r1,r2  ;r0:=r1 XOR r2
BIC r0,r1,r2  ;r0:=r1 AND (NOT r2), bit clear
```

Simple Register Operands (2/2)

- Register Movement Operations
  - omit 1st source operand from the format

```
MOV r0,r2      ;r0:=r2
MVN r0,r2      ;r0:=NOT r2, move 1's complement
```

- Comparison Operations
  - not produce result; omit the destination from the format
  - just set the condition code bits (N, Z, C and V) in CPSR

```
CMP r1,r2      ;set cc on r1-r2, compare
CMN r1,r2      ;set cc on r1+r2, compare negated
TST r1,r2      ;set cc on r1 AND r2, bit test
TEQ r1,r2      ;set cc on r1 XOR r2, test equal
```

Immediate Operands

- Replace the second source operand with an immediate operand, which is a literal constant, preceded by “#”

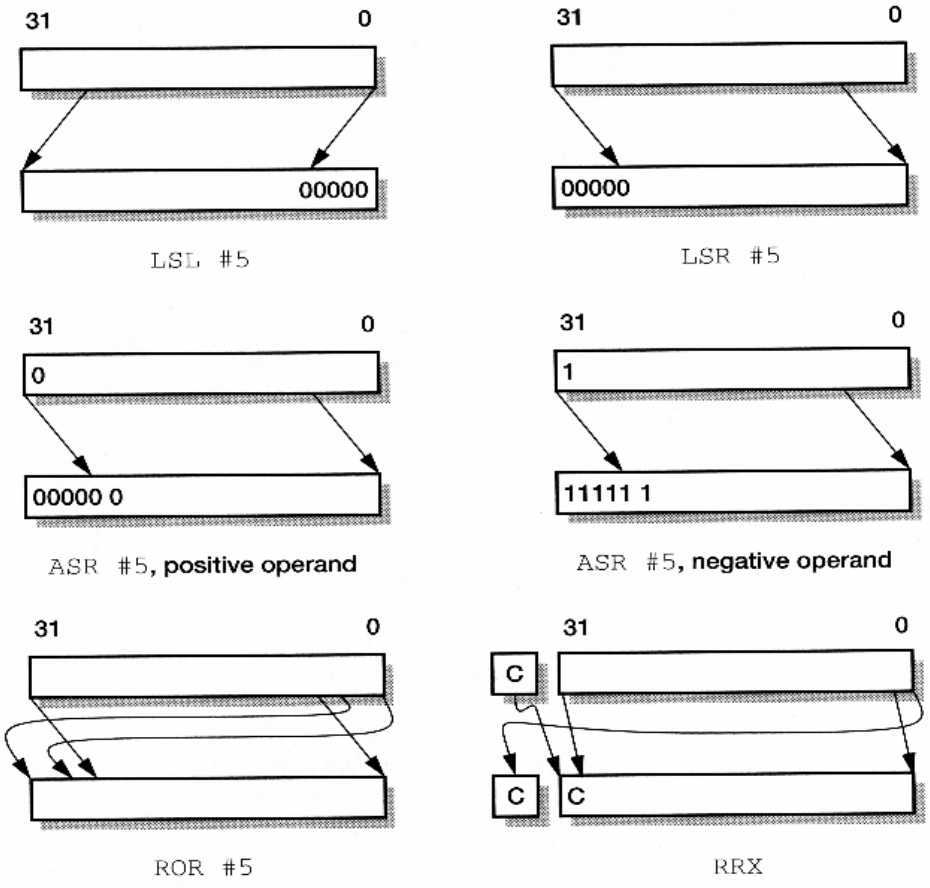
```
ADD r3,r3,#1      ;r3:=r3+1
AND r8,r7,#&FF    ;r8:=r7[7:0], &:hexadecimal
```

- Immediate = $(0\sim255)*2^{2n}$

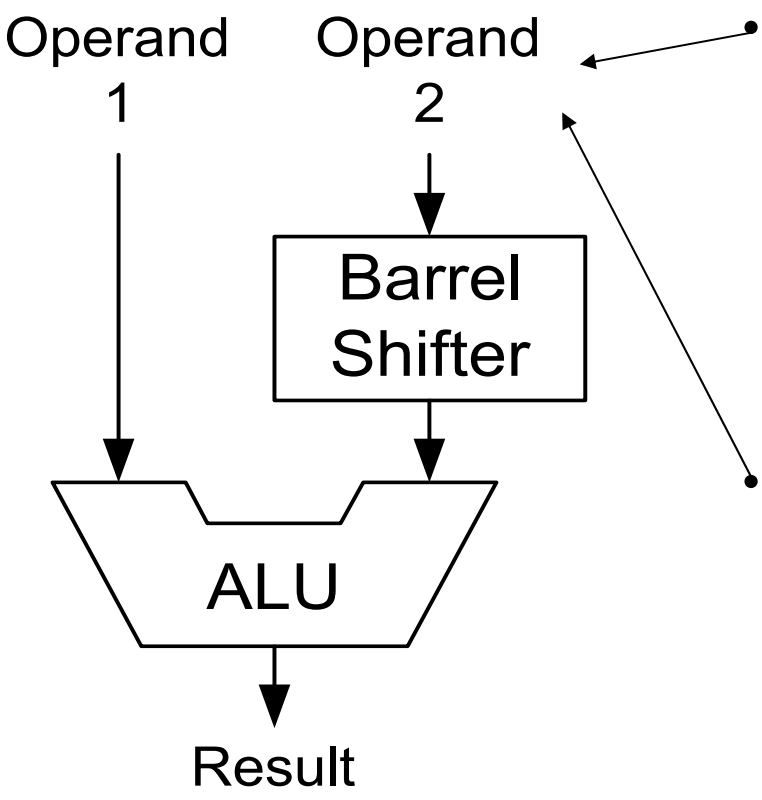
where n is 0-15 4-bit value

Shifted Register Operands

- `ADD r3,r2,r1,LSL#3`
  `;r3:=r2+8*r1`
  - a single instruction executed in a single cycle
- LSL: Logical shift left by 0 to 31 places, 0 filled at the lsb end
- LSR, ASL(Arithmetic Shift Left), ASR, ROR(Rotate Right), RRX(Rotate Right eXtended by 1 place)
- `ADD r5,r5,r3,LSL r2`
  `;r5:=r5+r3*2`^{r2}
- `MOV r12,r4,ROR r3  ;r12:=r4 rotated right by value of r3`



Using the Barrel Shifter: the 2nd Operand



- Register, optionally with shift operation applied.
  - Shift value can be either:
    - 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

Multiply

```
MUL r4,r3,r2    ;r4:=(r3*r2)[31:0]
```

- Multiply-Accumulate

```
MLA r4,r3,r2,r1        ;r4:=(r3*r2+r1)[31:0]
```

Multiplication by a Constant

- Multiplication by a constant equals to a ((power of 2) +/- 1) can be done in a single cycle
  - Using MOV, ADD or RSBs with an inline shift
- Example: r0=r1*5
- Example: r0=r1+(r1*4)

```
ADD r0,r1,r1,LSL #2
```

- Can combine several instructions to carry out other multiplies
- Example: r2=r3*119
- Example: r2=r3*17*7
- Example: r2=r3*(16+1)*(8-1)

```
ADD r2,r3,r3,LSL #4   ;r2:=r3*17
RSB r2,r2,r2,LSL #3   ;r2:=r2*7
```

Data Processing Instructions (1/3)

- `<op>{<cond>}{S} Rd,Rn,#<32-bit immediate>`
- `<op>{<cond>}{S} Rd,Rn,Rm,{<shift>}`
  - omit Rn when the instruction is monadic (MOV, MVN)
  - omit Rd when the instruction is a comparison, producing only condition code outputs (CMP, CMN, TST, TEQ)
  - <shift> specifies the shift type (LSL, LSR, ASL, ASR, ROR or RRX) and in all cases but RRX, the shift amount which may be a 5-bit immediate (# < # shift>) or a register Rs
- 3-address format
  - 2 source operands and 1 destination register
  - one source is always a register, the second may be a register, a shifted register or an immediate value

Data Processing Instructions (2/3)

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn – Op2
0011	RSB	Reverse subtract	Rd := Op2 – Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn – Op2 + C – 1
0111	RSC	Reverse subtract with carry	Rd := Op2 – Rn + C – 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn – Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Data Processing Instructions (3/3)

- Allows direct control of whether or not the condition codes are affected by S bit (condition code unchanged when S=0)

  N=1 if the result is negative; 0 otherwise

  (i.e. N=bit 31 of the result)

  Z=1 if the result is zero; 0 otherwise

  C= carry out from the ALU when ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN; carry out from the shifter

  V=1 if overflow from bit 30 to bit 31; 0 if no overflow

  (V is preserved in non-arithmetic operations)

- PC may be used as a source operand (address of the instruction plus 8) except when a register-specified shift amount is used
- PC may be specified as the destination register, the instruction is a form of branch (return from a subroutine)

Examples

- `ADD r5,r1,r3`
- `ADD Rs,PC,#offset  ;PC is ADD address+8`
- Decrement r2 and check for zero

```
            SUBS r2,r2#1   ;dec r2 and set cc
            BEQ LABEL
            …
```

- Multiply r0 by 5

```
            ADD r0,r0,r0,LSL #2
```

- A subroutine to multiply r0 by 10

```
            MOV r0,#3
            BL      TIMES10
            ……
TIMES10     MOV r0,r0,LSL #1        ;*2
            ADD r0,r0,r0,LSL #2     ;*5
            MOV PC,r14              ;return
```

- Add a 64-bit integer in r1, r0 to one in r3, r2

```
            ADDs r2,r2,r0
            ADC r3,r3,r1
```

Multiply Instructions (1/2)

- 32-bit Product (Least Significant)

```
MUL{<cond>}{S} Rd,Rm,Rs
MLA{<cond>}{S} Rd,Rm,Rs,Rn
```

- 64-bit Product

```
<mul>{<cond>}{S} RdHi,RdLo,Rm,Rs
<mul> is (UMULL,UMLAC,SMULL,SMLAL)
```

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

Multiply Instructions (2/2)

- Accumulation is denoted by “+=“
- Example: form a scalar product of two vectors

```
        MOV r11,#20          ;initialize loop counter
        MOV r10,#0           ;initialize total
Loop    LDR r0,[r8],#4       ;get first component
        LDR r1,[r9],#4       ;get second component
        MLA r10,r0,r1,r10
        SUBS r11,r11,#1
        BNE Loop
```

Count Leading Zeros (CLZ-V5T only)

- CLZ{<cond>} Rd,Rm
  - set Rd to the number of the bit position of the most significant 1 in Rm; If Rm is zero, Rd=32
  - useful for renormalizing numbers
- Example

```
MOV r0,#&100
CLZ r1,r0        ;r1:=23
```

- Example

```
CLZ r1,r2
MOVS r2,r2,LSL r1
```

Data Transfer Instructions

- Three basic forms to move data between ARM registers and memory
  - single register load and store instruction
    - a byte, a 16-bit half word, a 32-bit word
  - multiple register load and store instruction
    - to save or restore workspace registers for procedure entry and exit
    - to copy blocks of data
  - single register swap instruction
    - a value in a register to be exchanged with a value in memory
    - to implement semaphores to ensure mutual exclusion on accesses

Single Register Data Transfer

- Word transfer

  `LDR / STR`

- Byte transfer

  `LDRB / STRB`

- Halfword transfer

  `LDRH / STRH`

- Load singed byte or halfword-load value and sign extended to 32 bits

  `LDRSB / LDRSH`

- All of these can be conditionally executed by inserting the appropriate condition code after STR/LDR

  `LDREQB`

Addressing

- Register-indirect addressing
- Base-plus-offset addressing
  - base register
    - r0-r15
  - offset, add or subtract an unsigned number
    - immediate
    - register (not PC)
    - scaled register (only available for word and unsigned byte instructions)
- Stack addressing
- Block-copy addressing

Register-indirect Addressing

- Use a value in one register (base register) as a memory address

```
LDR r0,[r1]  ;r0:=mem32[r1]
STR r0,[r1]  ;mem32[r1]:=r0
```

- Other forms
  - adding immediate or register offsets to the base address

Initializing an Address Pointer

- A small offset to the program counter, r15
  - ARM assembler has a “pseudo” instruction, `ADR`

- As an example, a program which must copy data from TABLE1 to TABLE2, both of which are near to the code

```
COPY        ADR r1,TABLE1 ;r1 points to TABLE1
            ADR r2,TABLE2 ;r2 points to TABLE2
            …
TABLE1
            …             ;<source>
TABLE2
            …             ;<destination>
```

Single Register Load and Store

- A base register, an offset which may be another register or an immediate value

```
Copy        ADR r1,TABLE1
            ADR r2,TABLE2
Loop        LDR r0,[r1]
            STR r0,[r2]
            ADD r1,r1,#4
            ADD r2,r2,#4
            ???
            …
TABLE1
            …
TABLE2
            …
```

Base-plus-offset Addressing (1/3)

- Pre-indexing

  `LDR r0,[r1,#4]` ;r0:=mem_{32}[r1+4]

  – offset up to 4K, added or subtracted, (#-4)

- Post-indexing

  `LDR r0,[r1],#4` ;r0:=mem_{32}[r1], r1:=r1+4

  – equivalent to a simple register-indirect load, but faster, less code space

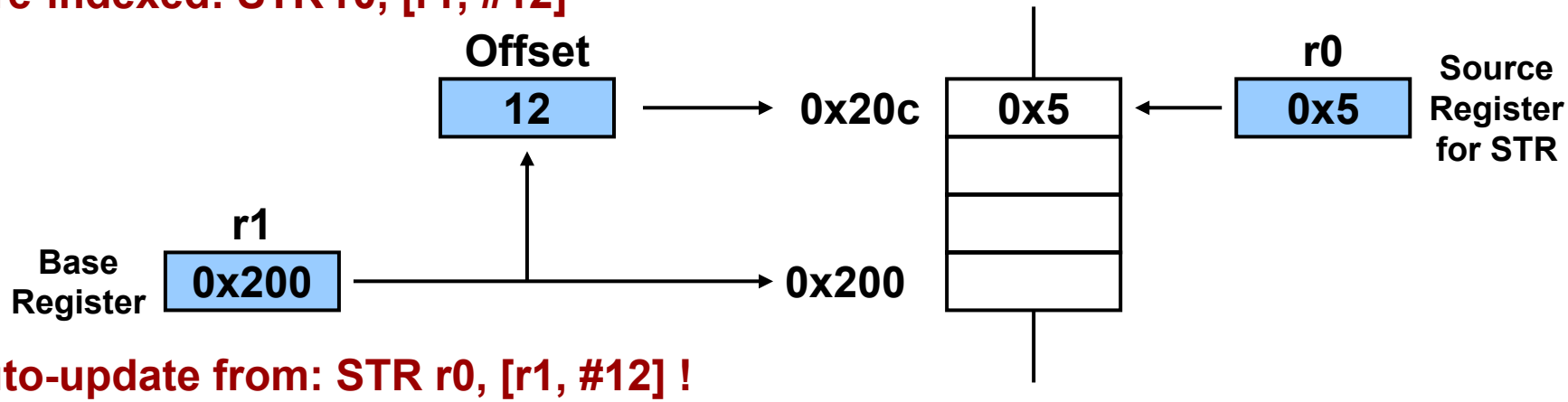
- Auto-indexing

  `LDR r0,[r1,#4]!` ;r0:=mem32[r1+4], r1:=r1+4

  – no extra time, auto-indexing performed while the data is being fetched from memory

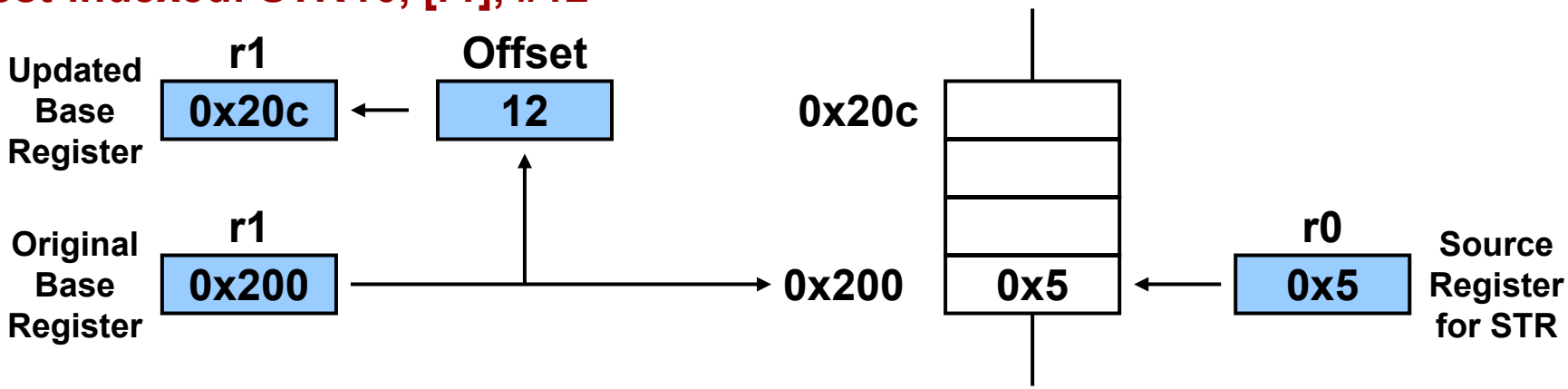
Base-plus-offset Addressing (2/3)

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



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

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



Base-plus-offset Addressing (3/3)

-
```
Copy        ADR r1,TABLE1
            ADR r2,TABLE2
Loop        LDR r0,[r1],#4
            STR r0,[r2],#4
            ???
            …
TABLE1
            …
TABLE2
            …
```

- A single unsigned byte load

  `LDRB r0,[r1]  ;r0:=mem`$_8$`[r1]`

  – also support signed bytes, 16-bit half-word

Loading Constants (1/2)

- No single ARM instruction can load a 32-bit immediate constant directly into a register
  - all ARM instructions are 32-bit long
  - ARM instructions do not use the instruction stream as data
- The data processing instruction format has 12 bits available for operand 2
  - if used directly, this would only give a range of 4096
- Instead it is used to store 8-bit constants, give a range of 0~255
- These 8 bits can then be rotated right through an even number of positions (i.e. RORs by 0,2,4,…,30)
- This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory

Loading Constants (2/2)

- This gives us:
  - 0~255 [0-0xff]
  - 256,260,264,…,1020 [0x100-0x3fc,step4,0x40-0xff ror 30]
  - 1024,1240,…,4080 [0x400-0xff0,step16,0x40-0xff ror 28]
  - 4096,4160,…,16320 [0x1000-0x3fc0,step64,0x40-0xff ror 26]

- To load a constant, simply move the required value into a register - the assembler will convert to the rotate form for us
  - `MOV r0,#4096 ;MOV r0,#0x1000 (0x40 ror 26)`

- The bitwise complements can also be formed using MVN:
  - `MOV r0,#0xFFFFFFFF ;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,#Imm10]
```

- As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.

Multiple Register Data Transfer (1/2)

- The load and store multiple instructions (LDM/STM) allow between 1 and 16 registers to be transferred to or from memory
  - order of register transfer cannot be specified, order in the list is insignificant
  - lowest register number is always transferred to/from lowest memory location accessed
- The transferred registers can be either
  - any subset of the current bank of registers (default)
  - any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a “^”)
- Base register used to determine where memory access should occur
  - 4 different addressing modes
  - base register can be optionally updated following the transfer (using “!”)
- These instructions are very efficient for
  - moving blocks of data around memory
  - saving and restoring context - stack

Multiple Register Data Transfer (2/2)

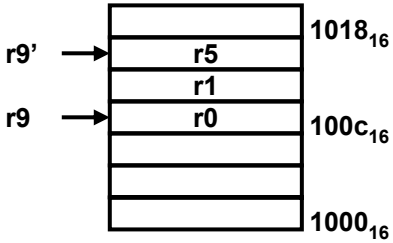
- Allow any subset (or all, r0 to r15) of the 16 registers to be transferred with a single instruction

```
LDMIA r1,{r0,r2,r5}
    ;r0:=mem32[r1]
    ;r2:=mem32[r1+4]
    ;r5:=mem32[r1+8]
```

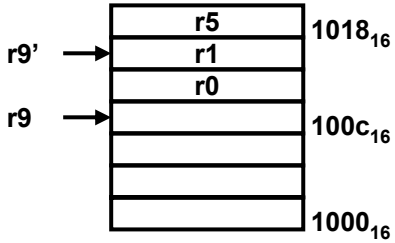
Stack Processing

- A stack is usually implemented as a linear data structure which grows up (an ascending stack) or down (a descending stack) memory
- A stack pointer holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a full stack), or by pointing to the vacant slot where the next data item will be placed (an empty stack)
- ARM multiple register transfer instructions support all four forms of stacks
  - **full ascending**: grows up; base register points to the highest address containing a valid item
  - **empty ascending**: grows up; base register points to the first empty location above the stack
  - **full descending**: grows down; base register points to the lowest address containing a valid data
  - **empty descending**: grows down, base register points to the first empty location below the stack

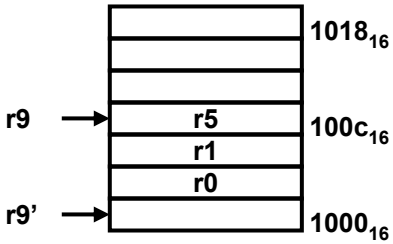
Block Copy Addressing (1/2)



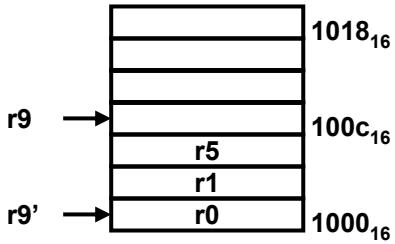
STMIA r9!, {r0, r1, r5}



STMIB r9!, {r0, r1, r5}



STMDA r9!, {r0, r1, r5}



STMDB r9!, {r0, r1, r5}

- Addressing modes

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Block Copy Addressing (2/2)

- Copy 8 words from the location r0 points to to the location r1 points to

```
LDMIA r0!,{r2-r9}
STMIA r1,{r2-r9}
```

  – r0 increased by 32, r1 unchanged

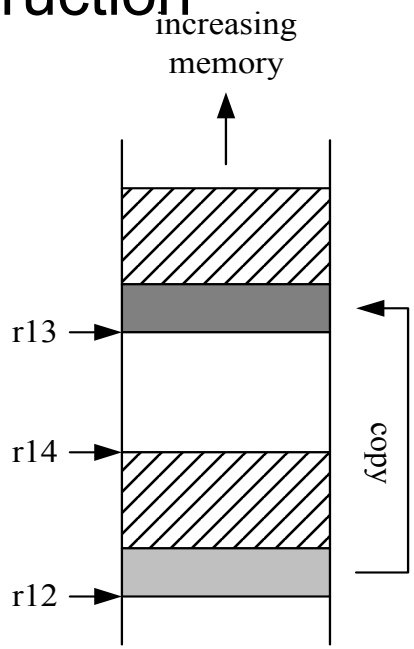
- If r2 to r9 contained useful values, preserve them by pushing them onto a stack

```
STMFD r13!,{r2-r9}
LDMIA r0!,{r2-r9}
STMIA r1,{r2-r9}
LDMFD r13,{r2-r9}
```

  – FD postfix: full descending stack addressing mode

Memory Block Copy

- The direction that the base pointer moves through memory is given by the postfix to the STM/LDM instruction
  - STMIA/LDMIA: Increment After
  - STMIB/LDMIB: Increment Before
  - STMDA/LDMDA: Decrement After
  - STMDB/LDMDB: Decrement Before



- For Example

```
;r12 points to start of source data
;r14 points to end of source data
;r13 points to start of destination data
Loop   LDMIA r12!,{r0-r11} ;load 48 bytes
       STMIA r13!,{r0-r11} ;and store them
       CMP   r12,r14       ;check for the end
       BNE   Loop          ;and loop until done
```

  - this loop transfers 48 bytes in 31 cycles
  - over 50Mbytes/sec at 33MHz

Single Word and Unsigned Byte Data Transfer Instructions

- Pre-indexed form

```
LDR|STR {<cond>}{B} Rd,[Rn,<offset>]{!}
```

- Post-index form

```
LDR|STR {<cond>}{B}{T} Rd,[Rn],<offset>
```

- PC-relative form

```
LDR|STR {<cond>}{B} Rd,LABEL
```

  – LDR ‘load register’; STR ‘store register’

    ‘B’ unsigned byte transfer, default is word;

    <offset> may be # +/-<12-bit immediate> or +/- Rm{,shift}

    ! auto-indexing

    T flag selects the user view of the memory translation and protection system

Example

- Store a byte in r0 to a peripheral

```
            LDR r1,UARTADD
            STRB r0,[r1]        ;store data to UART
            …
UARTADD     & &10000000
```

Half-word and Signed Byte Data Transfer Instructions

- Pre-indexed form

```
LDR|STR{<cond>}H|SH|SB Rd,[Rn,<offset>]{!}
```

- Post-indexed form

```
LDR|STR{<cond>}H|SH|SB Rd,[Rn],<offset>
```

  – <offset> is # +/-<8-bit immediate> or +/-Rm
  – H|SH|SB selects the data type - unsigned half-word, signed half-word and signed byte. Otherwise is for word and unsigned byte transfer

Example

- Expand an array of signed half-words into an array of words

```
        ADR   r1,ARRAY1    ;half-word array start
        ADR   r2,ARRAY2    ;word array start
        ADR   r3,ENDARR1   ;ARRAY1 end+2
Loop    LDRSH r0,[r1],#2   ;get signed half-word
        STR   r0,[r2],#4   ;save word
        CMP   r1,r3        ;check for end of array
        BLT   Loop         ;if not finished, loop
```

Multiple Register Transfer Instructions

- `LDM|STM {<cond>}<add mode> Rn{!}, <registers>`
  - <add mode> specifies one of the addressing modes; '!': auto-indexing; <registers> a list of registers, e.g. {r0,r3-r7,pc}
- In non-user mode, the CPSR may be restored by

  `LDM{<cond>}<add mode> Rn{!},<registers+PC>^`
- In non-user mode, the user registers may be saved or restored by

  `LDM|STM{<cond>}<add mode> Rn,<registers-PC>^`
  - The register list must not contain PC and write-back is not allowed

Example

- Save 3 work registers and the return address upon entering a subroutine (assume r13 has been initialized for use as a stack pointer)

```
STMFD r13!,{r0-r2,r14}
```

- Restore the work registers and return

```
LDMFD r13!,{r0-r2,pc}
```

Swap Memory and Register Instructions

- `SWP{<cond>}{B} Rd,Rm,[Rn]`
- `Rd <- [Rn], [Rn] <- Rm`

- Combine a load and a store of a word or an unsigned byte in a single instruction
- Example

```
ADR r0,SEMAPHORE
SWPB r1,r1,[r0]        ;exchange byte
```

Status Register to General Register Transfer Instructions

- `MRS{<cond>} Rd,CPSR|SPSR`

- The CPSR or the current mode SPSR is copied into the destination register. All 32 bits are copied.

- Example

```
MRS r0,CPSR
MRS r3,SPSR
```

General Register to Status Register Transfer Instructions

- `MSR{<cond>} CPSR_<field>|SPSR_<field>,#<32-bit immediate>`
  `MSR{<cond>} CPSR_<field>|SPSR_<field>,Rm`
  - <field> is one of
    - c - the control field PSR[7:0]
    - x - the extension field PSR[15:8]
    - s - the status field PSR[23:16]
    - f - the flag field PSR[31:24]
- Example
  - set N, X, C, V flags

```
MSR CPSR_f,#&f0000000
```

  - set just C, preserving N, Z, V

```
MRS r0,CPSR
ORR r0,r0,#&20000000        ;set bit29 of r0
MSR CPSR_f,r0
```

Control Flow Instructions

- Branch instructions
- Conditional branches
- Conditional execution
- Branch and link instructions
- Subroutine return instructions
- Supervisor calls
- Jump tables

Branch Instructions

```
            B  LABEL
            ...
LABEL       ...
```

– LABEL comes after or before the branch instruction

Conditional Branches

- The branch has a condition associated with it and it is only executed if the condition codes have the correct value - taken or not taken

```
        MOV r0,#0      ;initialize counter
Loop    …
        ADD r0,r0,#1   ;increment loop counter
        CMP r0,#10     ;compare with limit
        BNE Loop       ;repeat if not equal
                       ;else fall through
```

Conditional Branch

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional Execution

- An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions

```
            CMP r0,#5                                  CMP r0,#5
            BEQ Bypass      ;if(r0!=5)      ⇒          ADDNE r1,r1,r0
            ADD r1,r1,r0    ;{r1=r1+r0-r2}             SUBNE r1,r1,r2
            SUB r1,r1,r2
Bypass      …
```

- Whenever the conditional sequence is 3 instructions or fewer it is better (smaller and faster) to exploit conditional execution than to use a branch

```
if((a==b)&&(c==d)) e++;          CMP r0,r1
                          ⇒      CMPEQ r2,r3
                                 ADDEQ r4,r4,#1
```

Branch and Link Instructions

- Perform a branch, save the address following the branch in the link register, r14

```
            BL SUBR         ;branch to SUBR
            …               ;return here
SUBR        …               ;subroutine entry point
            MOV PC,r14      ;return
```

- For nested subroutine, push r14 and some work registers required to be saved onto a stack in memory

```
            BL SUB1
            …
SUB1        STMFD r13!,{r0-r2,r14} ;save work and link regs
            BL SUB2
            …
SUB2        …
```

Subroutine Return Instructions

```
SUB         …
            MOV PC,r14      ;copy r14 into r15 to return
```

- Where the return address has been pushed onto a stack

```
SUB1  STMFD r13!,{r0-r2,r14} ;save work regs and link
      BL SUB2
      …
      LDMFD r13!,{r0-e12,PC} ;restore work regs & return
```

Supervisor Calls

- The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly (e.g. input or output)
- SWI instruction
  - software interrupt or supervisor call

```
SWI SWI_WriteC ;output r0[7:0]
SWI SWI_Exit   ;return to monitor program
```

Jump Tables

- To call one of a set of subroutines, the choice depending on a value computed by the program

```
        BL JUMPTAB                BL JUMPTAB
        …                         …
JUMPTAB       CMP r0,#0           JUMPTAB       ADR r1,SUBTAB;r1->SUBTAB
        BEQ SUB0                  CMP r0,#SUBMAX       ;check for overrun
        CMP r0,#1                 LDRLS PC,[r1,r0,LSL#2];if OK,table jump
        BEQ SUB1                  B ERROR
        CMP r0,#2           SUBTAB DCD SUB0
        BEQ SUB2                  DCD SUB1
        …                         DCD SUB2
                                  …
```

- The ‘DCD’ directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right, which in these cases is just the address of the label.

Branch and Branch with Link (B,BL)

- `B{L}{<cond>} <target address>`
  - <target address> is normally a label in the assembler code.

24-bit offset, sign-extended, shift left 2 places
+ PC (address of branch instruction + 8)

---

target address

Examples

- Unconditional jump

```
            B        LABEL
            …
LABEL       …
```

- Loop ten times

```
            MOV r0,#10
Loop…
            SUBS r0,#1
            BNE Loop
            …
```

- Call a subroutine

```
            BL SUB
            …
SUB         …
            MOV PC,r14
```

- Conditional subroutine call

```
            CMP r0,#5
            BLLT SUB1        ; if r0<5, call SUB1
            BLGE SUB2        ; else call SUB2
```

Branch, Branch with Link and eXchange

- `B{L}X{<cond>} Rm`
  - the branch target is specified in a register, Rm
  - bit[0] of Rm is copied into the T bit in CPSR; bit[31:1] is moved into PC
  - if Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit
  - if Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[1]
- `BLX <target address>`
  - call Thumb subroutine from ARM
  - the H bit (bit 24) is also added into bit 1 of the resulting address, allowing an odd half-word address to be selected for the target instruction which will always be a Thumb instruction

Example

- A call to a Thumb subroutine

```
        CODE32
        …
        BLX TSUB        ;call Thumb subroutine
        …
        CODE16          ;start of Thumb code
TSUB    …
        BX r14          ;return to ARM code
```

Software Interrupt (SWI)

- `SWI{<cond>} <24-bit immediate>`
  - used for calls to the operating system and is often called a "supervisor call"
  - it puts the processor into supervisor mode and begins executing instruction from address 0x08
    - Save the address of the instruction after SWI in r14_svc
    - Save the CPSR in SPSR_svc
    - Enter supervisor mode and disable IRQs by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1
    - Set PC to 08_{16} and begin executing the instruction there
  - the 24-bit immediate does not influence the operation of the instruction but may be interpreted by the system code

Examples

- Output the character ‘A’

```
        MOV     r0,#'A'
        SWI     SWI_WriteC
```

- Finish executing the user program and return to the monitor

```
        SWI     SWI_Exit
```

- A subroutine to output a text string

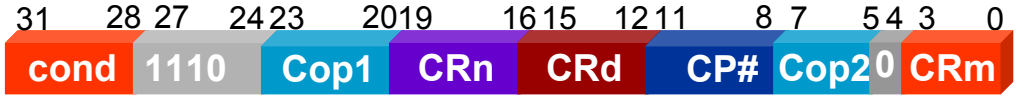
```
        BL STROUT
        =  "Hello World",&0a,&0d,0
        …
STROUT  LDRB r0,[r14],#1      ;get character
        CMP r0,#0             ;check for end marker
        SWINE SWI_WriteC      ;if not end, print
        BNE STROUT            ;   …       , loop
        ADD r14,#3            ;align to next word
        BIC r14,#3
        MOV PC,r14            ;return
```

Coprocessor Instructions

- Extend the instruction set through the addition of coprocessors
  - **System Coprocessor**: control on-chip function such as cache and memory management unit
  - **Floating-point Coprocessor**
  - **Application-Specific Coprocessor**
- Coprocessors have their own private register sets and their state is controlled by instructions that mirror the instructions that control ARM registers

Coprocessor Data Operations

- `CDP{<cond>}<CP#>,<Cop1>,CRd,CRn,CRm{,<Cop2>}`



- Use to control internal operations on data in coprocessor registers
- CP# identifies the coprocessor number
- Cop1, Cop2 operation
- Examples

```
CDP     P2,3,C0,C1,C2
CDPEQ   P3,6,C1,C5,C7,4
```

Coprocessor Data Transfers

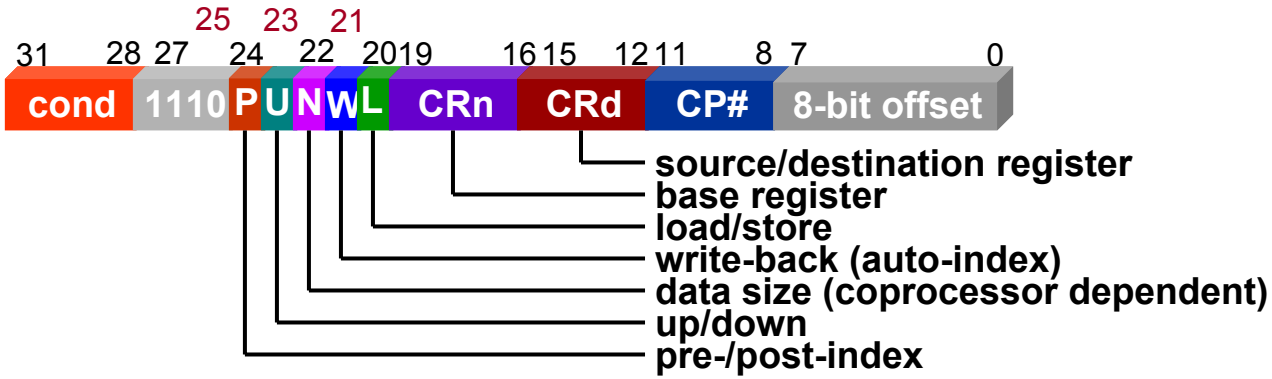
- Pre-indexed form

```
LDC|STC{<cond>}{L}<CP#>,CRd,[Rn,<offset>]{!}
```

- Post-indexed form

```
LDC|STC{<cond>}{L}<CP#>,CRd,[Rn],<offset>
```

  – L flag, if present, selects the long data type
  – <offset> is # +/-<8-bit immediate>



  – the number of words transferred is controlled by the coprocessor
  – address calculated within ARM; number of words transferred controlled by the coprocessor

- Examples

```
LDC P6,c0,[r1]
STCEQL P5,c1,[r0],#4
```

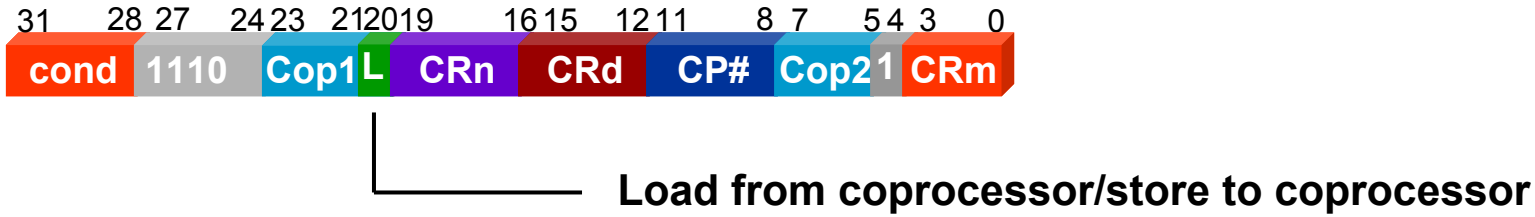
Coprocessor Register Transfers

- Move to ARM register from coprocessor

```
MRC{<cond>} <CP#>,<Cop1>,Rd,CRn,CRm,{,<Cop2>}
```

- Move to coprocessor from ARM register

```
MCR{<cond>} <CP#>,<Cop1>,Rd,CRn,CRm,{,<Cop2>}
```



- Examples

```
MCR P14,3,r0,c1,c2
MRCCS P2,4,r3,c3,c4,6
```

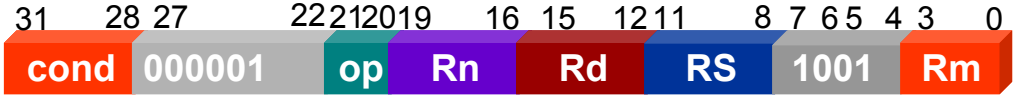
Breakpoint Instructions (BKPT-v5T only)

- `BKPT <16-bit immediate>`

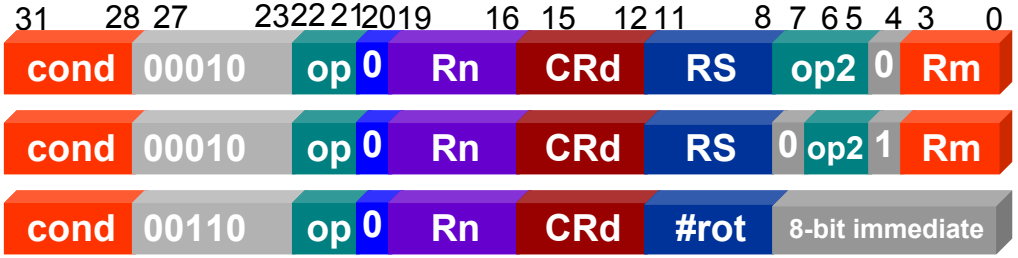
- Used for software debugging purposes; they cause the processor to break from normal instruction execution and enter appropriate debugging procedures
- BKPT is unconditional
- Handled by an exception handler installed on the prefetch abort vector

Unused Instruction Space

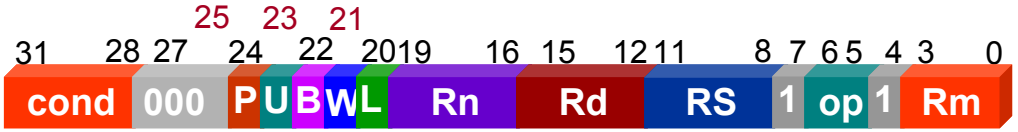
- Unused Arithmetic Instructions



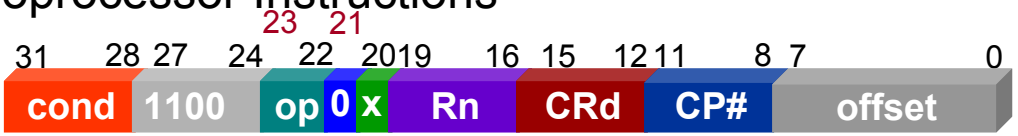
- Unused Control Instructions



- Unused Load/Store Instructions



- Unused Coprocessor Instructions



- Undefined Instruction Space

16-bit Instruction Set

Thumb Instruction Set (1/3)

Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
ADC	Add with carry	○		○
ADD	Add	○	○	○
AND	AND	○		○
ASR	Arithmetic Shift Right	○		○
B	Branch	○		
Bxx	Conditional Branch	○		
BIC	Bit Clear	○		○
BL	Branch with Link			
BX	Branch and Exchange	○	○	
CMN	Compare Negative	○		○
CMP	Compare	○	○	○
EOR	EOR	○		○
LDMIA	Load Multiple	○		
LDR	Load Word	○		

Thumb Instruction Set (2/3)

Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
LDRB	Load Byte	○		
LDRH	Load Halfword	○		
LSL	Logical Shift Left	○		○
LDSB	Load Signed Byte	○		
LDSH	Load Signed Halfword	○		
LSR	Logical Shift Right	○		○
MOV	Move Register	○	○	○
MUL	Multiply	○		○
MVN	Move Negative Register	○		○
NEG	Negate	○		○
ORR	OR	○		○
POP	Pop Registers	○		
PUSH	Push Registers	○		
ROR	Rotate Right	○		○

Thumb Instruction Set (3/3)

Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
SBC	Subtract with Carry	○		○
STMIA	Store Multiple	○		
STR	Store Word	○		
STRB	Store Byte	○		
STRH	Store Halfword	○		
SWI	Software Interrupt			
SUB	Subtract	○		○
TST	Test Bits	○		○

Thumb Instruction Format

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			*Move shifted register*
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			*Add/subtract*
3	0	0	1	Op		Rd			Offset8								*Move/compare/add /subtract immediate*
4	0	1	0	0	0	0	Op				Rs			Rd			*ALU operations*
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			*Hi register operations /branch exchange*
6	0	1	0	0	1	Rd			Word8								*PC-relative load*
7	0	1	0	1	L	B	0	Ro			Rb			Rd			*Load/store with register offset*
8	0	1	0	1	H	S	1	Ro			Rb			Rd			*Load/store sign-extended byte/halfword*
9	0	1	1	B	L	Offset5					Rb			Rd			*Load/store with immediate offset*
10	1	0	0	0	L	Offset5					Rb			Rd			*Load/store halfword*
11	1	0	0	1	L	Rd			Word8								*SP-relative load/store*
12	1	0	1	0	SP	Rd			Word8								*Load address*
13	1	0	1	1	0	0	0	0	S	SWord7							*Add offset to stack pointer*
14	1	0	1	1	L	1	0	R	Rlist								*Push/pop registers*
15	1	1	0	0	L	Rb			Rlist								*Multiple load/store*
16	1	1	0	1	Cond				Soffset8								*Conditional branch*
17	1	1	0	1	1	1	1	1	Value8								*Software Interrupt*
18	1	1	1	0	0	Offset11											*Unconditional branch*
19	1	1	1	1	H	Offset											*Long branch with link*
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

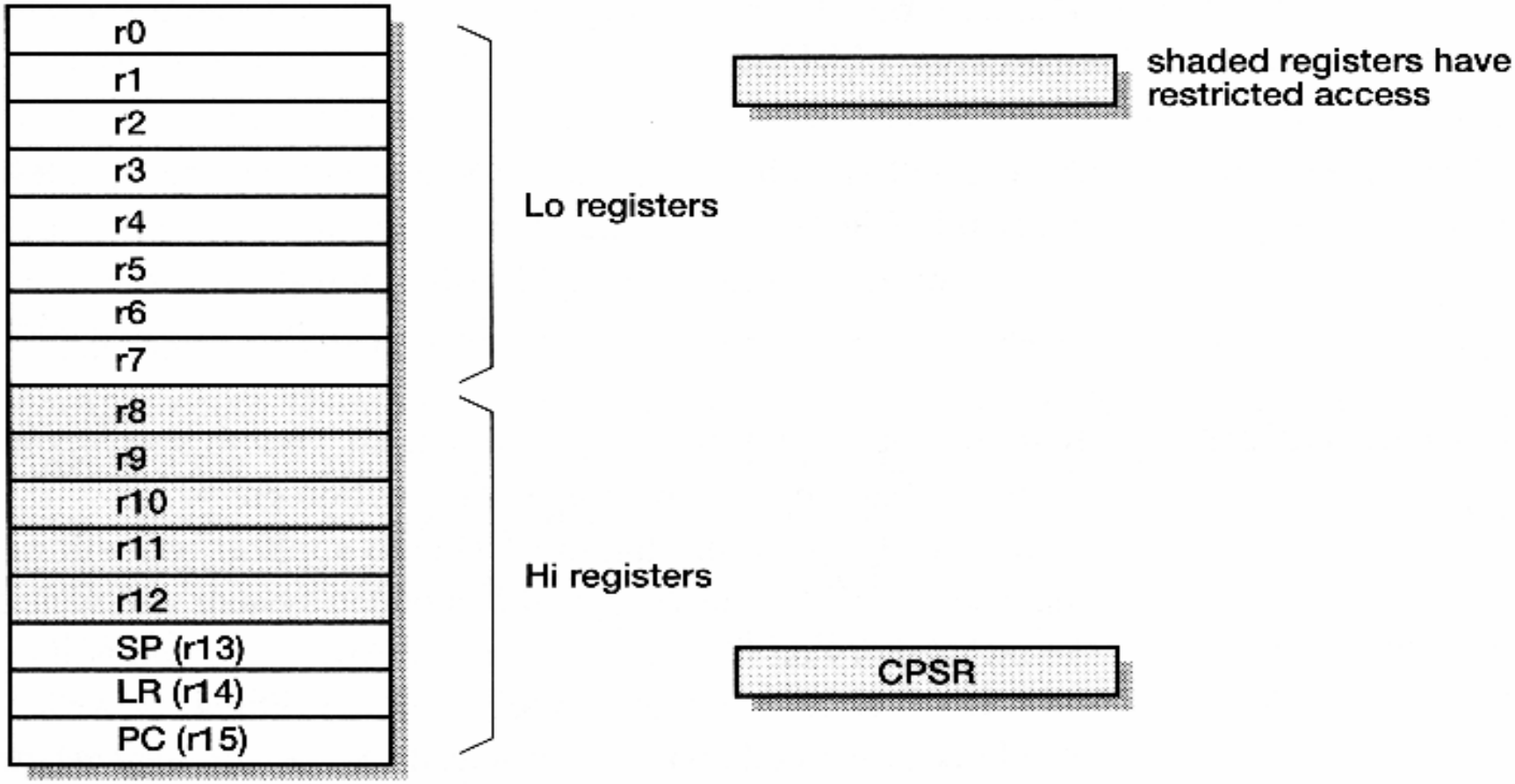
Thumb-ARM Difference

- Thumb instruction set is a subset of the ARM instruction set and the instructions operate on a restricted view of the ARM registers
- Most Thumb instructions are executed unconditionally (All ARM instructions are executed conditionally)
- Many Thumb data processing instructions use a 2-address format, i.e. the destination register is the same as one of the source registers (ARM data processing instructions, with the exception of the 64-bit multiplies, use a 3-address format)
- Thumb instruction formats are less regular than ARM instruction formats => dense encoding

Register Access in Thumb

- Not all registers are directly accessible in Thumb
- Low register **r0~r7**: fully accessible
- High register **r8~r12**: only accessible with MOV, ADD, CMP; only CMP sets the condition code flags
- **SP**(stack pointer), **LR**(link register) & **PC**(program counter): limited accessibility, certain instructions have implicit access to these
- **CPSR**: only indirect access
- **SPSR**: no access

Thumb Accessible Registers



Branches

- Thumb defines three PC-relative branch instructions, each of which have different offset ranges
  - Offset depends upon the number of available bits
- Conditional Branches
  - `B<cond> label`
  - 8-bit offset: range of -128 to 127 instructions (+/-256 bytes)
  - Only conditional Thumb instructions
- Unconditional Branches
  - `B label`
  - 11-bit offset: range of -1024 to 1023 instructions (+/- 2Kbytes)
- Long Branches with Link
  - `BL subroutine`
  - Implemented as a pair of instructions
  - 22-bit offset: range of -2097152 to 2097151 instructions (+/- 4Mbytes)

Data Processing Instructions

- Subset of the ARM data processing instructions
- Separate shift instructions (e.g. LSL, ASR, LSR, ROR)

```
LSL Rd,Rs,#Imm5    ;Rd:=Rs <shift> #Imm5
ASR Rd,Rs          ;Rd:=Rd <shift> Rs
```

- Two operands for data processing instructions
  - act on low registers

```
BIC Rd,Rs          ;Rd:=Rd AND NOT Rs
ADD Rd,#Imm8             ;Rd:=Rd+#Imm8
```

  - also three operand forms of add, subtract and shifts

```
ADD Rd,Rs,#Imm3    ;Rd:=Rs+#Imm3
```

- Condition code always set by low register operations

Load or Store Register

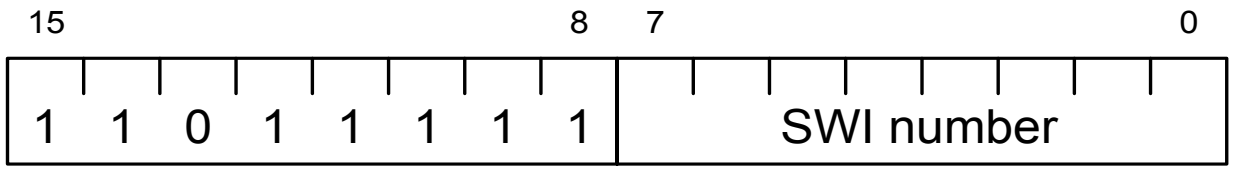
- Two pre-indexed addressing modes
  - base register+offset register
  - base register+5-bit offset, where offset scaled by
    - 4 for word accesses (range of 0-124 bytes / 0-31 words)
      - `STR Rd,[Rb,#Imm7]`
    - 2 for halfword accesses (range of 0-62 bytes / 0-31 halfwords)
      - `LDRH Rd,[Rb,#Imm6]`
    - 1 for byte accesses (range of 0-31 bytes)
      - `LDRB Rd,[Rb,#Imm5]`
- Special forms:
  - load with PC as base with 1Kbyte immediate offset (word aligned)
    - used for loading a value from a literal pool
  - load and store with SP as base with 1Kbyte immediate offset (word aligned)
    - used for accessing local variables on the stack

Block Data Transfers

- Memory copy, incrementing base pointer after transfer
  - `STMIA Rb!, {Low Reg list}`
  - `LDMIA Rb!, {Low Reg list}`
- Full descending stack operations
  - `PUSH {Low Reg list}`
  - `PUSH {Low Reg list, LR}`
  - `POP {Low Reg list}`
  - `POP {Low Reg list, PC}`
- The optional addition of the LR/PC provides support for subroutine entry/exit.

Miscellaneous

- Thumb SWI instruction format
  - same effect as ARM, but SWI number limited to 0~255
  - syntax:
    - `SWI <SWI number>`



- Indirect access to CPSR and no access to SPSR, so no MRS or MSR instructions
- No coprocessor instruction space

Thumb Instruction Entry and Exit

- T bit, bit 5 of CPSR
  - if T=1, the processor interprets the instruction stream as 16-bit Thumb instruction
  - if T=0, the processor interprets it as standard ARM instructions
- Thumb Entry
  - ARM cores startup, after reset, executing ARM instructions
  - executing a Branch and Exchange instruction (BX)
    - set the T bit if the bottom bit of the specified register was set
    - switch the PC to the address given in the remainder of the register
- Thumb Exit
  - executing a Thumb BX instruction

The Need for Interworking

- The code density of Thumb and its performance from narrow memory make it ideal for the bulk of C code in many systems. However there is still a need to change between ARM and Thumb state within most applications:
  - ARM code provides better performance from wide memory
    - therefore ideal for speed-critical parts of an application
  - some functions can only be performed with ARM instructions, e.g.
    - access to CPSR (to enable/disable interrupts & to change mode)
    - access to coprocessors
  - exception Handling
    - ARM state is automatically entered for exception handling, but system specification may require usage of Thumb code for main handler
  - simple standalone Thumb programs will also need an ARM assembler header to change state and call the Thumb routine

Interworking Instructions

- Interworking is achieved using the Branch Exchange instructions
  - in Thumb state

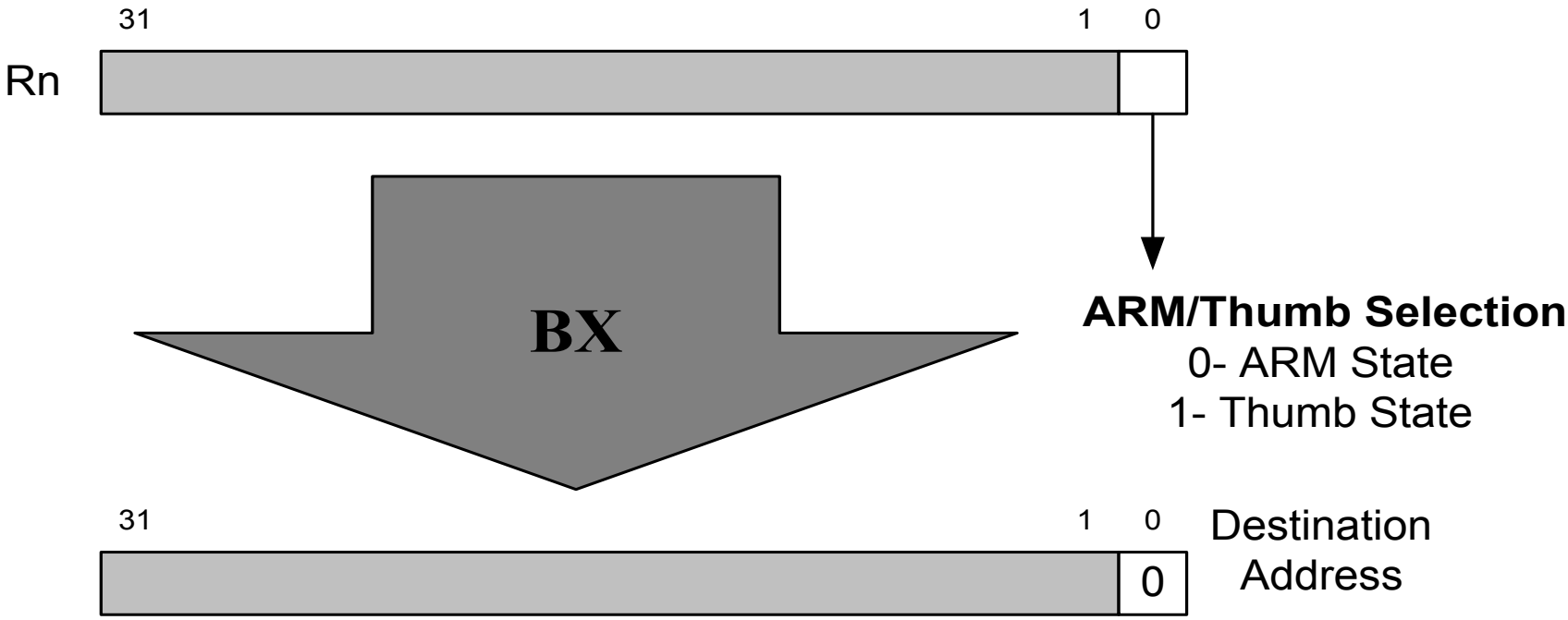
    ```
    BX Rn
    ```

  - in ARM state (on Thumb-aware cores only)

    ```
    BX<condition> Rn
    ```

  where Rn can be any registers (r0 to r15)
- This performs a branch to an absolute address in 4GB address space by copying Rn to the program counter
- Bit 0 of Rn specifies the state to change to

Switching between States

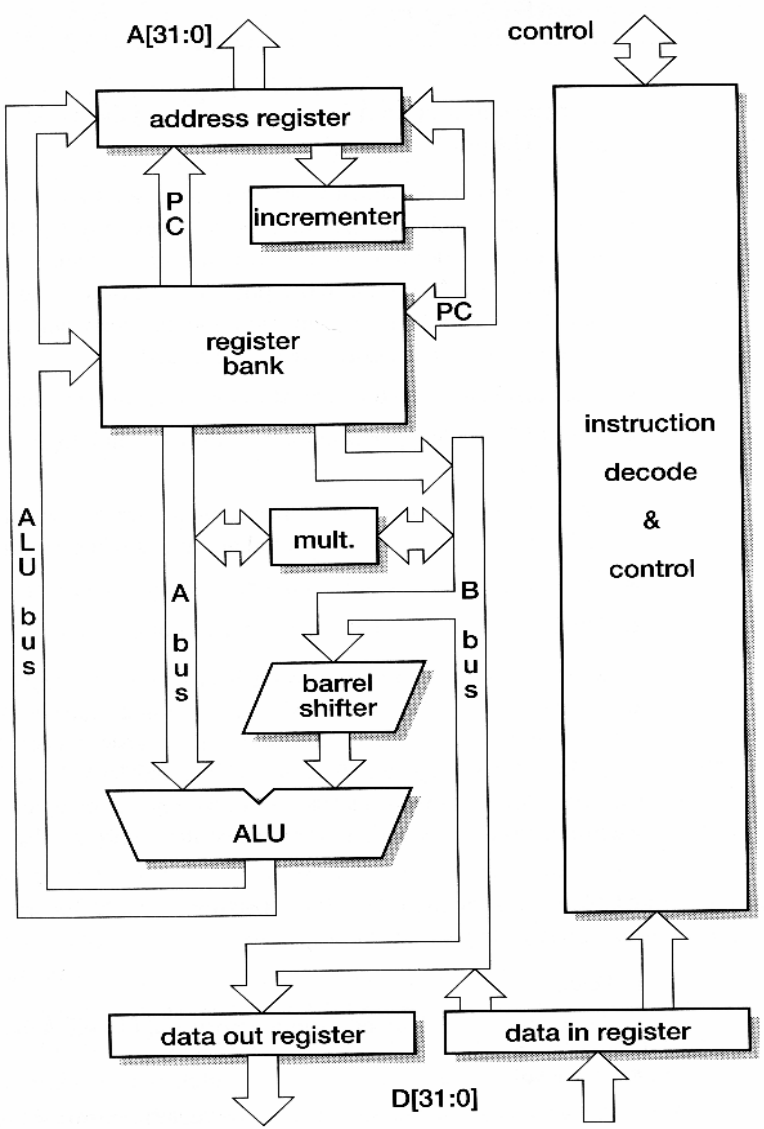


Example

```
;start off in ARM state
                CODE32
                ADR r0,Into_Thumb+1     ;generate branch target
                                ;address & set bit 0,
                                        ;hence arrive Thumb state
                BX r0                   ;branch exchange to Thumb
                …
                CODE16                  ;assemble subsequent as
                                ;Thumb
Into_Thumb      …
                ADR r5,Back_to_ARM      ;generate branch target to
                                        ;word-aligned address,
                                        ;hence bit 0 is cleared.
                BX r5                   ;branch exchange to ARM
                …
                CODE32                  ;assemble subsequent as
                                        ;ARM
Back_to_ARM     …
```

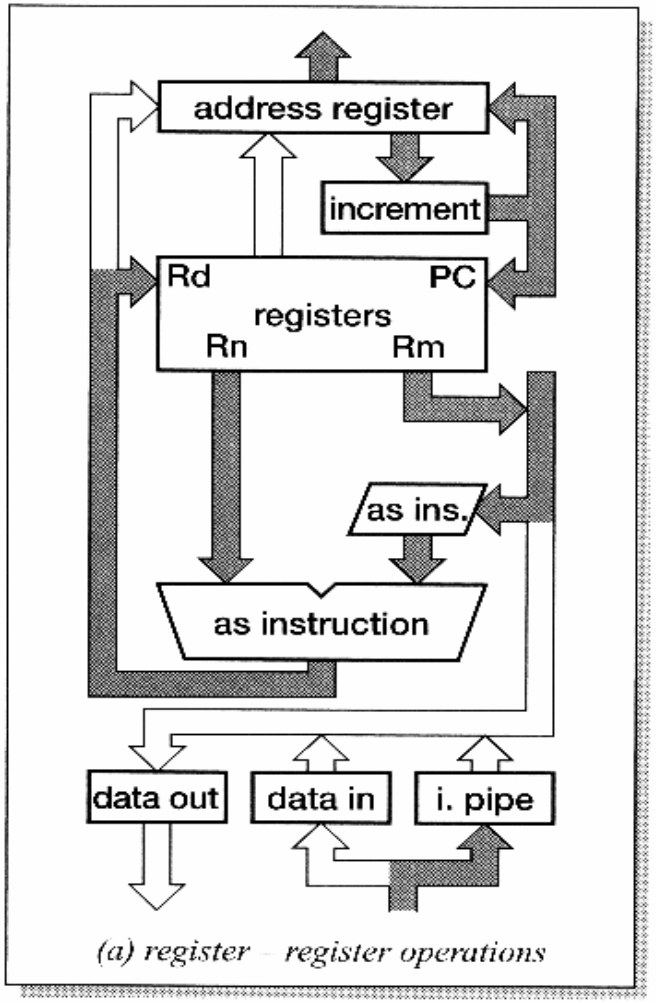
ARM Processor Core

3-Stage Pipeline ARM Organization

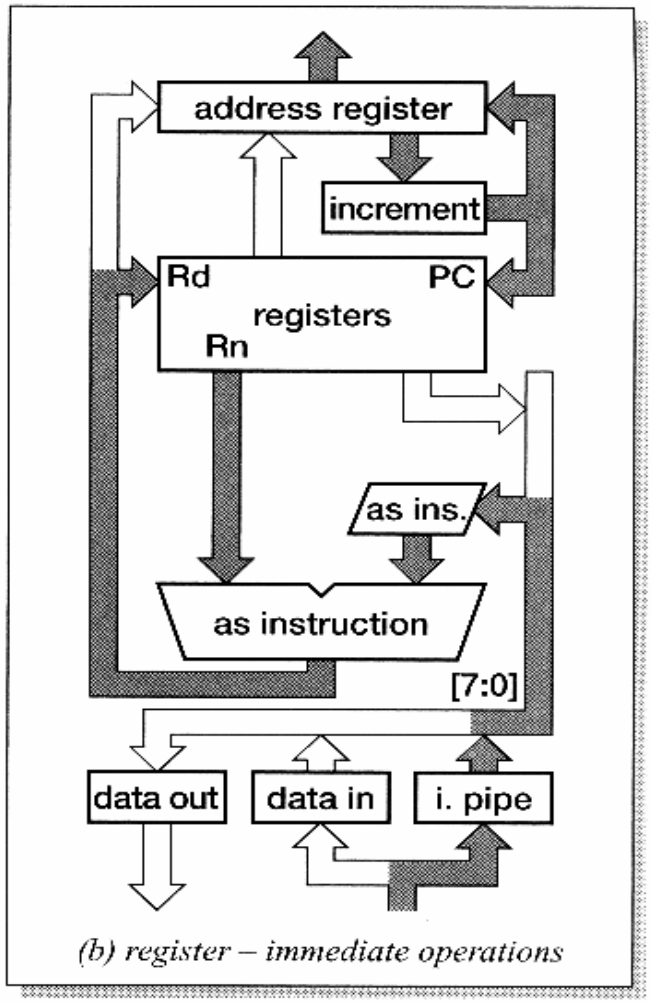


- Register Bank
  - 2 read ports, 1write ports, access any register
  - 1 additional read port, 1 additional write port for r15 (PC)
- Barrel Shifter
  - Shift or rotate the operand by any number of bits
- ALU
- Address register and incrementer
- Data Registers
  - Hold data passing to and from memory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

Data Processing Instructions



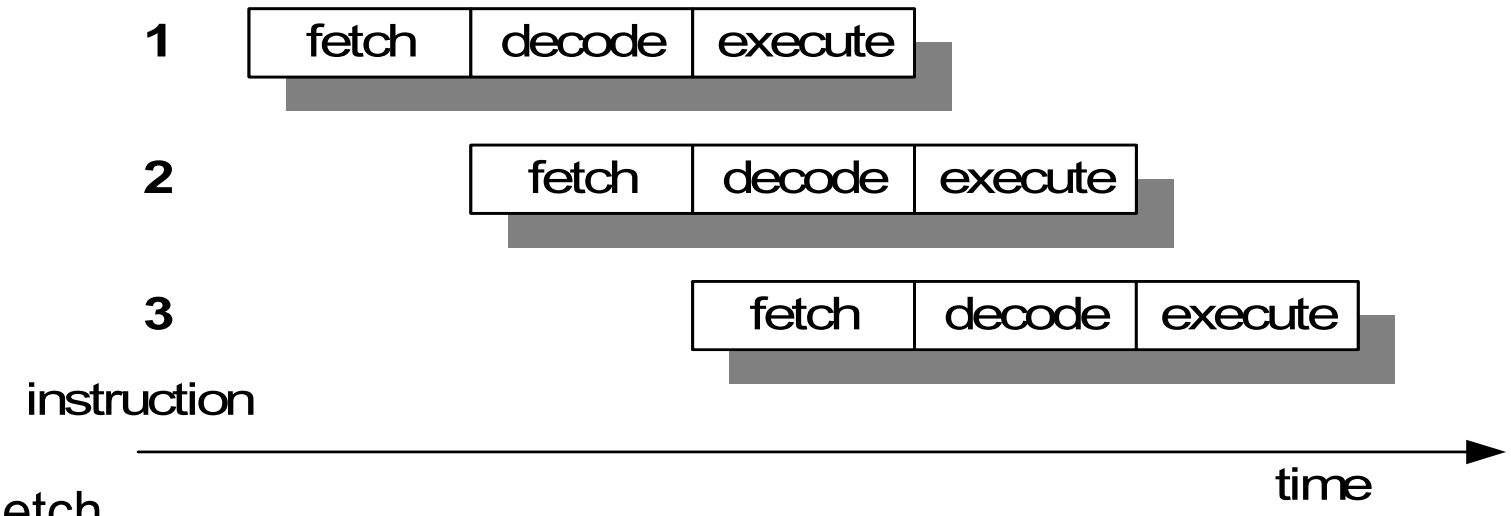
(a) register – register operations



(b) register – immediate operations

- All Operations take place in a single clock cycle

3-Stage Pipeline (1/2)

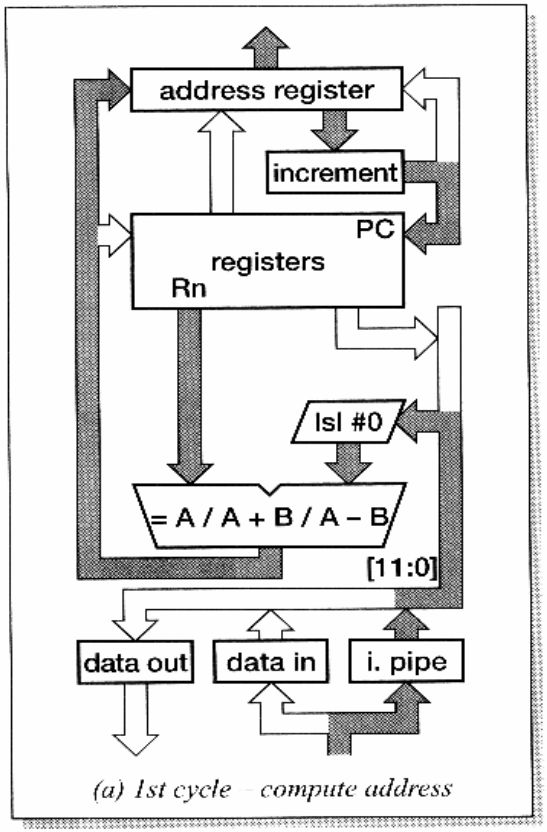


- Fetch
  - the instruction is fetched from memory and placed in the instruction pipeline
- Decode
  - the instruction is decoded and the datapath control signals prepared for the next cycle
- Execute
  - the register bank is read, an operand shifted, the ALU result generated and written back into a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

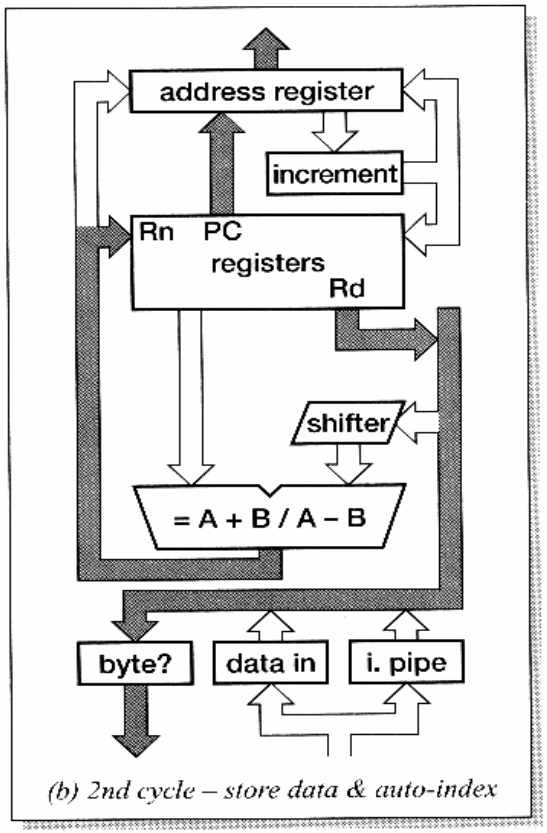
3-Stage Pipeline (2/2)

- 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
- When accessing r15 (PC), r15=address of current instruction + 8

Data Transfer Instructions



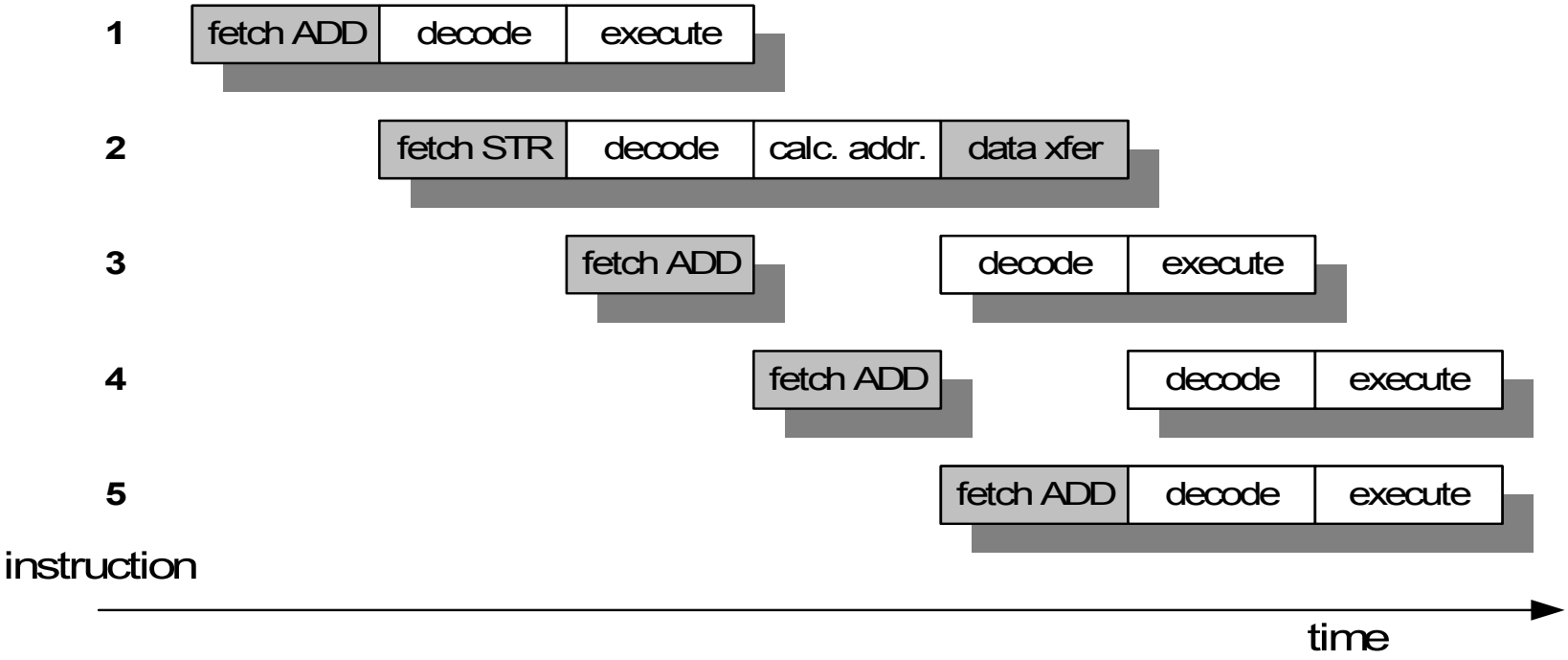
(a) 1st cycle – compute address



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

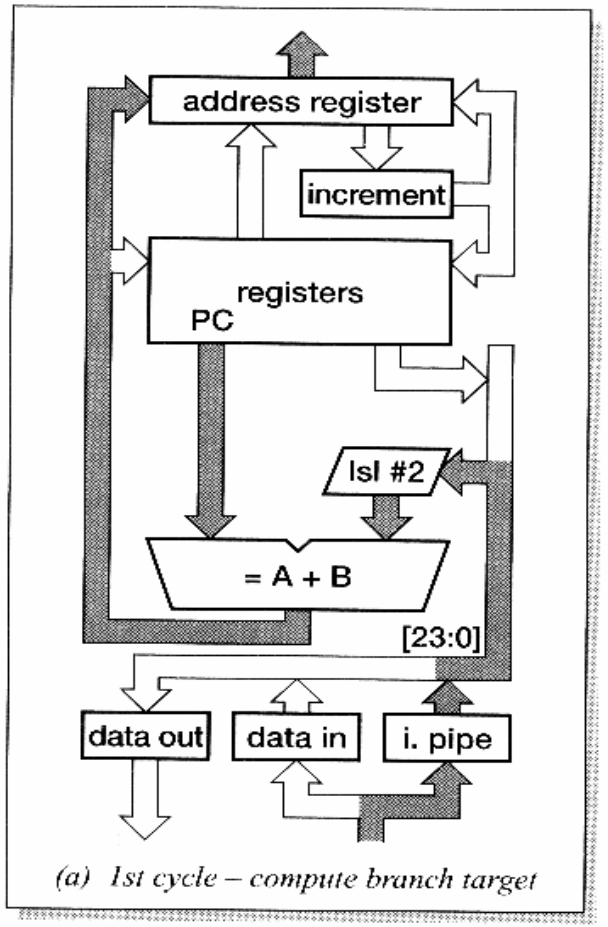
- Computes a memory address similar to a data processing instruction
- Load instruction follow a similar pattern except that the data from memory only gets as far as the 'data in' register on the 2nd cycle and a third cycle is needed to transfer the data from there to the destination register

Multi-cycle Instruction

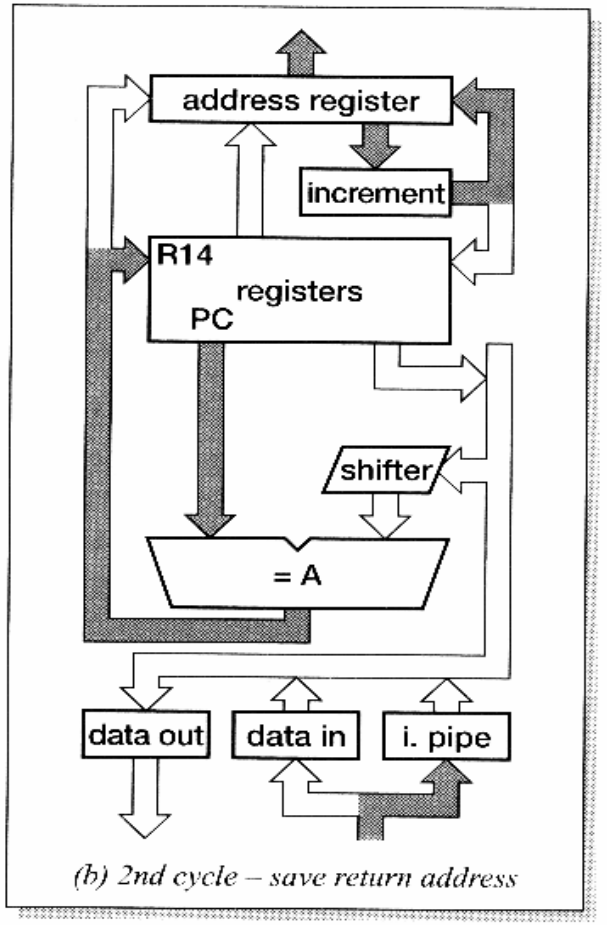


- Memory access (fetch, data transfer) in every cycle
- Datapath used in every cycle (execute, address calculation, data transfer)
- Decode logic generates the control signals for the data path use in next cycle (decode, address calculation)

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 make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch

Branch Pipeline Example

Cycle				1	2	3	4	5
address	opeation							
0x8000	BL	fetch	decode	execute	linkret	adjust		
0x8004	X		fetch	decode				
0x8008	XX			fetch				
0x8FEC	ADD				fetch	decode	execute	
0x8FF0	SUB					fetch	decode	execute
0x8FF4	MOV						fetch	decode
								fetch

Breaking the pipeline

Note that the core is executing in ARM state

Interrupt Pipeline Example

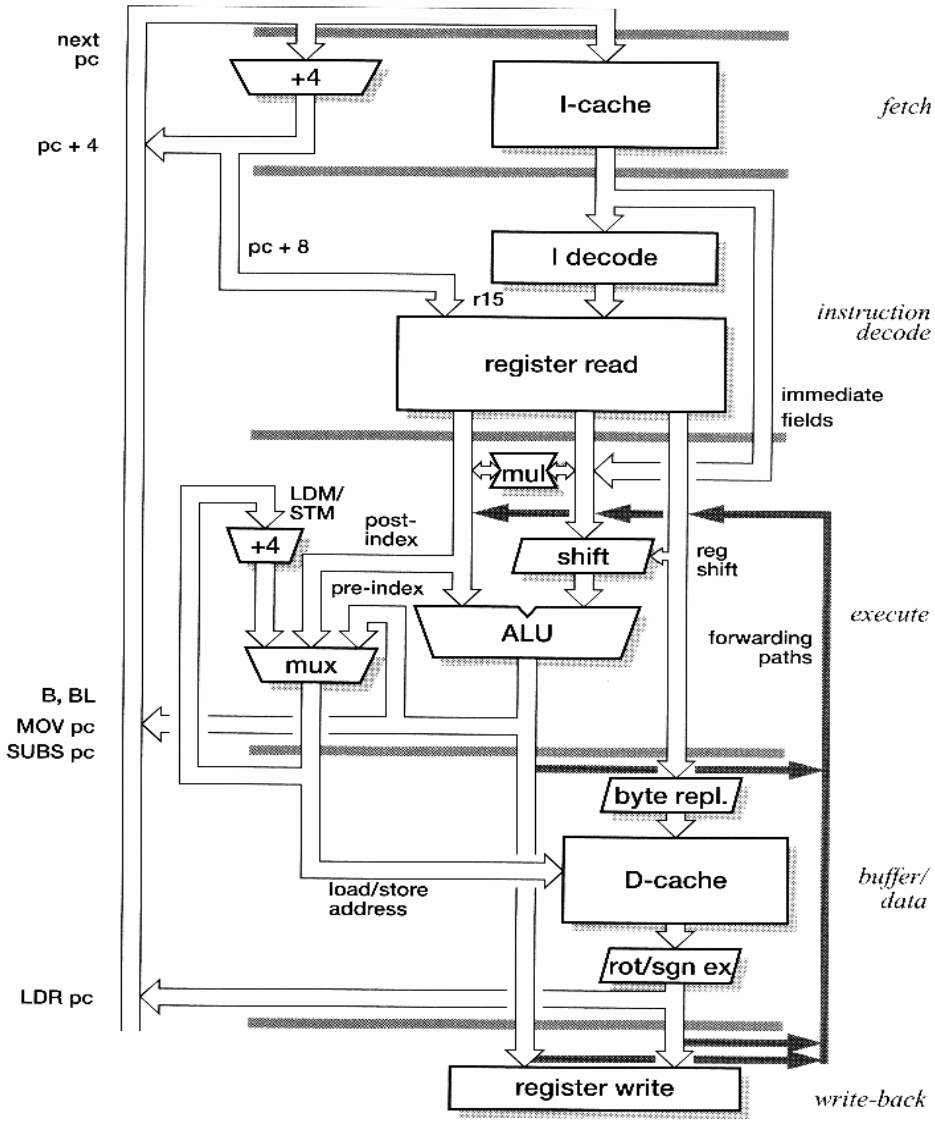
Cycle				IRQ 1	2	3	4	5	6	7	8
address	opeation										
0x8000	ADD	fetch	decode	execute							
0x8004	SUB		fetch	decode IRQ	execute IRQ	linkret	adjust				
0x8008	MOV			fetch							
0x800C	X				fetch						
0x001C	B(to 0xAF00)					fetch	decode	execute			
0x0018	XX						fetch	decode			
0x0020	XXX							fetch			
0xAF00	STMFD								fetch	decode	execute
0xAF04	MOV									fetch	decode
0xAF08	LDR										fetch

IRQ interrupt minimum latency = 7 cycles

5-Stage Pipelined ARM Organization

- Tprog=Ninst*CPI*cycle_time
  - Ninst, compiler dependent
  - CPI, hazard => pipeline stalls
  - cycle_time, frequency
- Separate instruction and data memories => 5 stage pipeline
- Used in ARM9TDMI

ARM9TDMI 5-stage Pipeline Organization



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

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

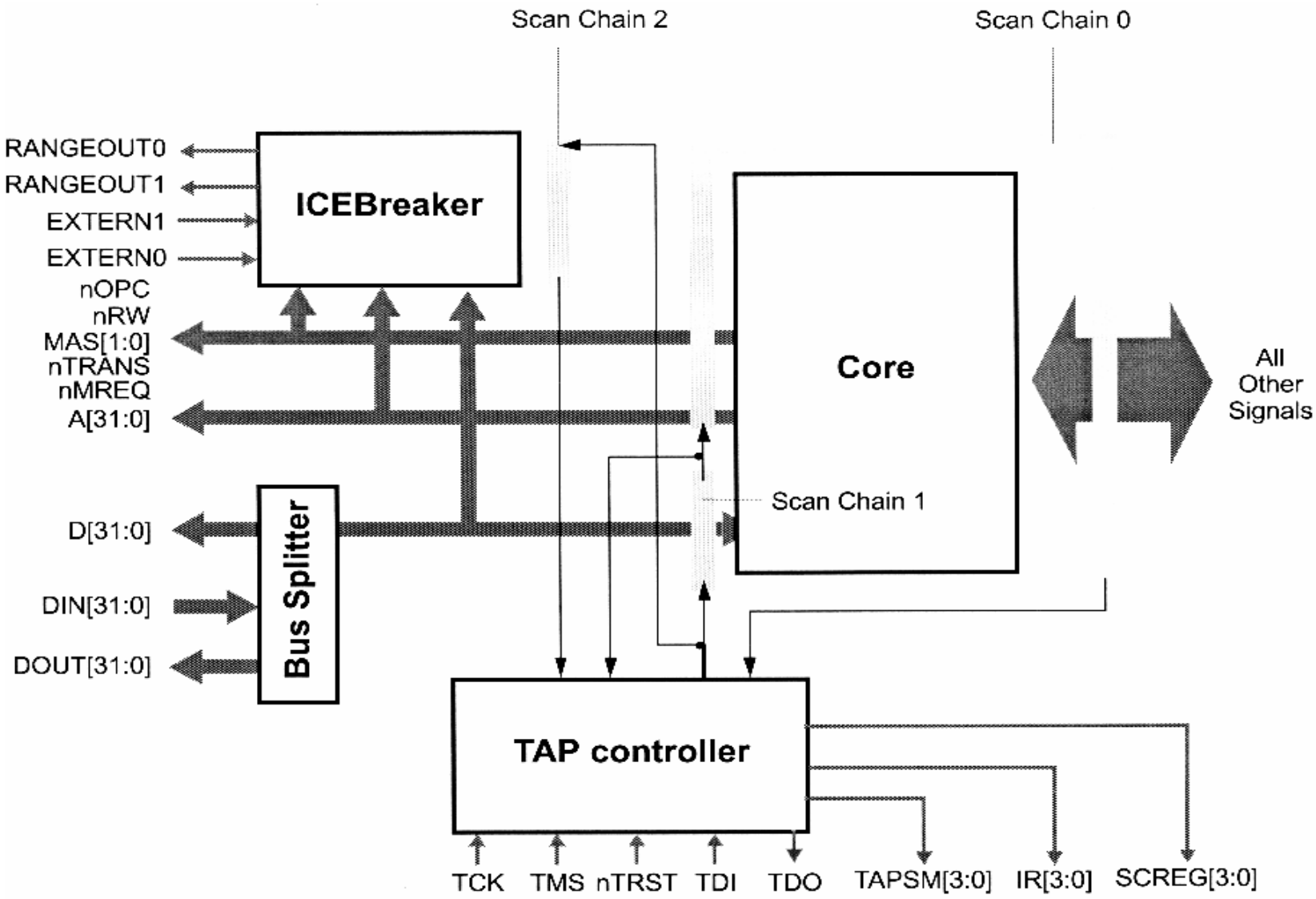
```
LDR rN,[…]
ADD r2,r1,rN      ;use rN immediately
```

  – one stall
  – compiler rescheduling

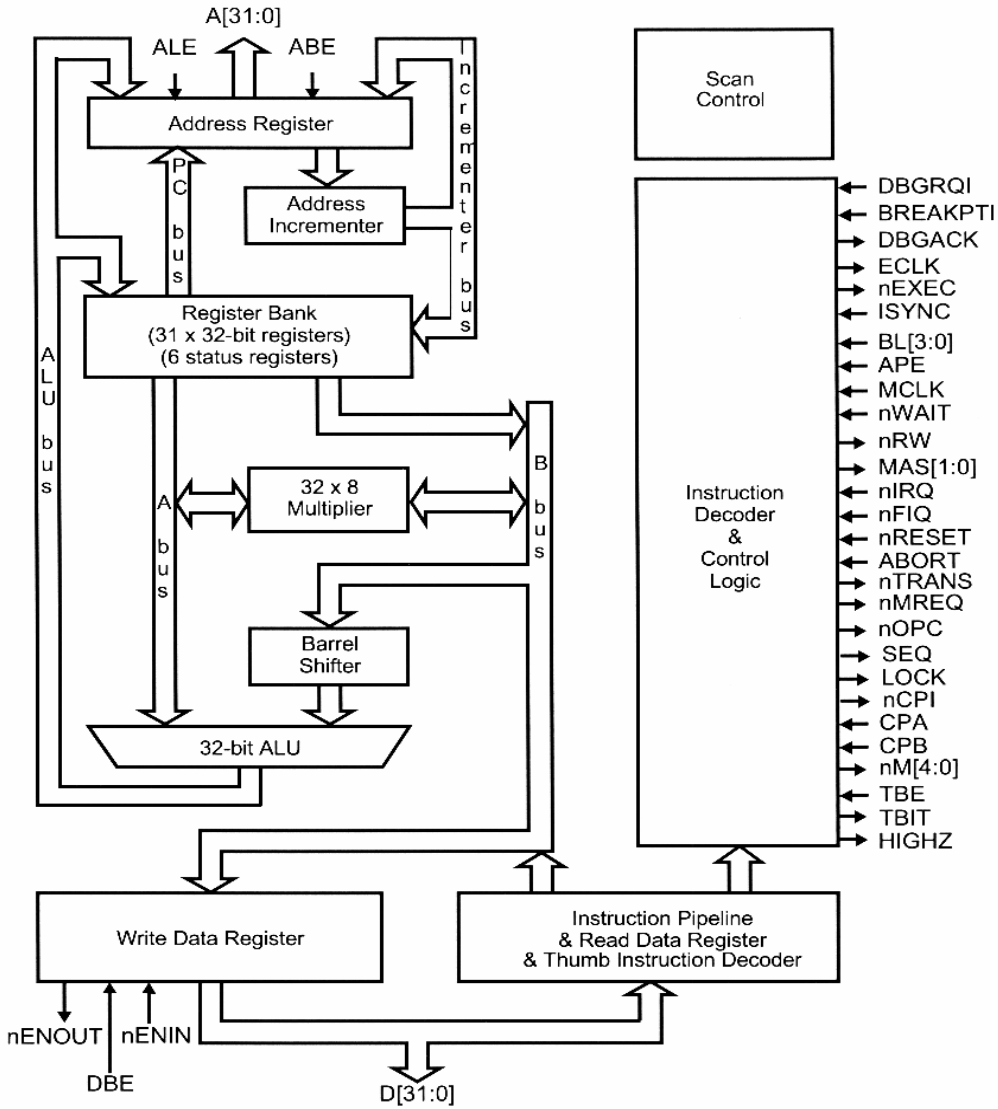
ARM7TDMI Processor Core

- 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**: Embedded ICE hardware
- Von Neumann architecture
- 3-stage pipeline, CPI ~1.9

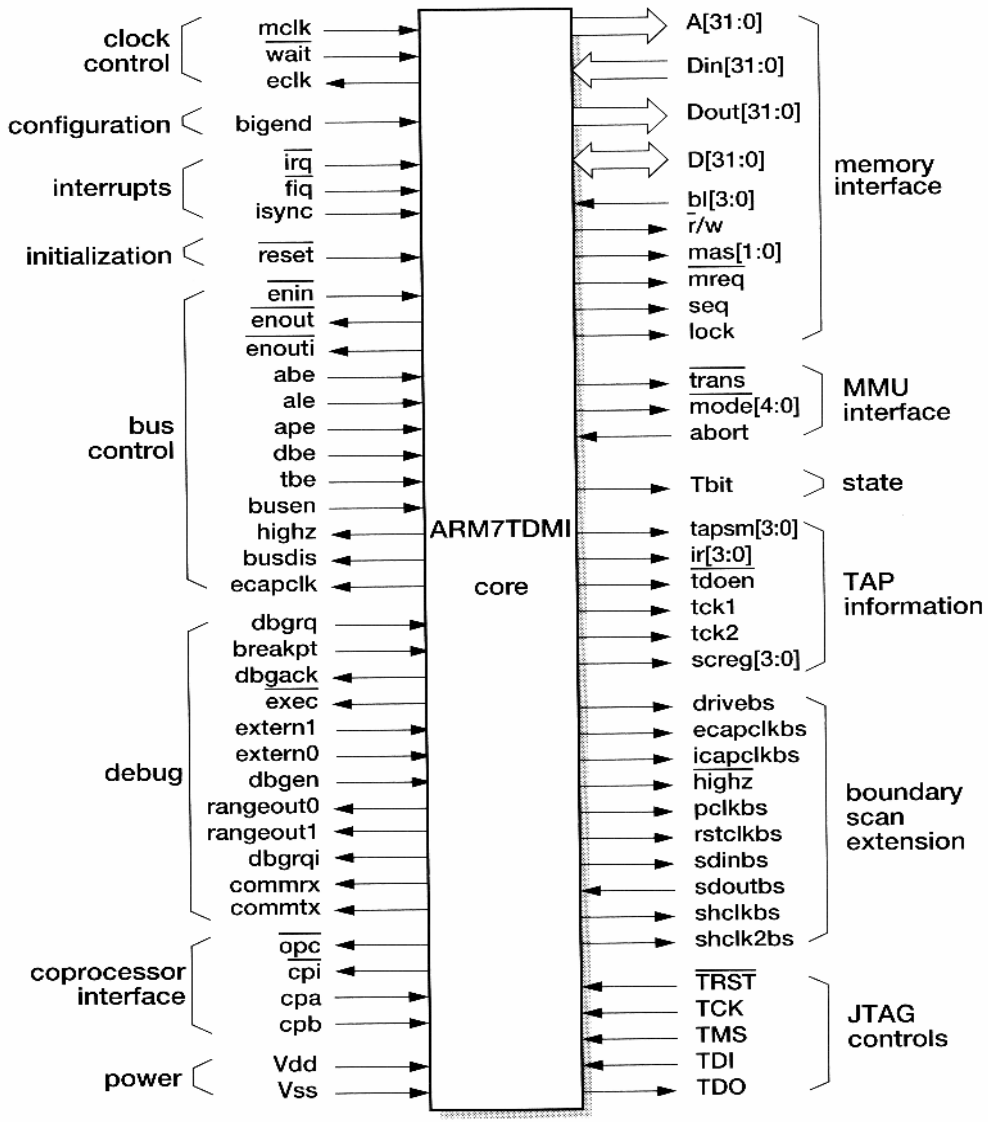
ARM7TDMI Block Diagram



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 a processor cycle which requires a memory access
  - seq indicates that the memory address will be sequential to that used in the previous cycle

$\overline{mreq}$	*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

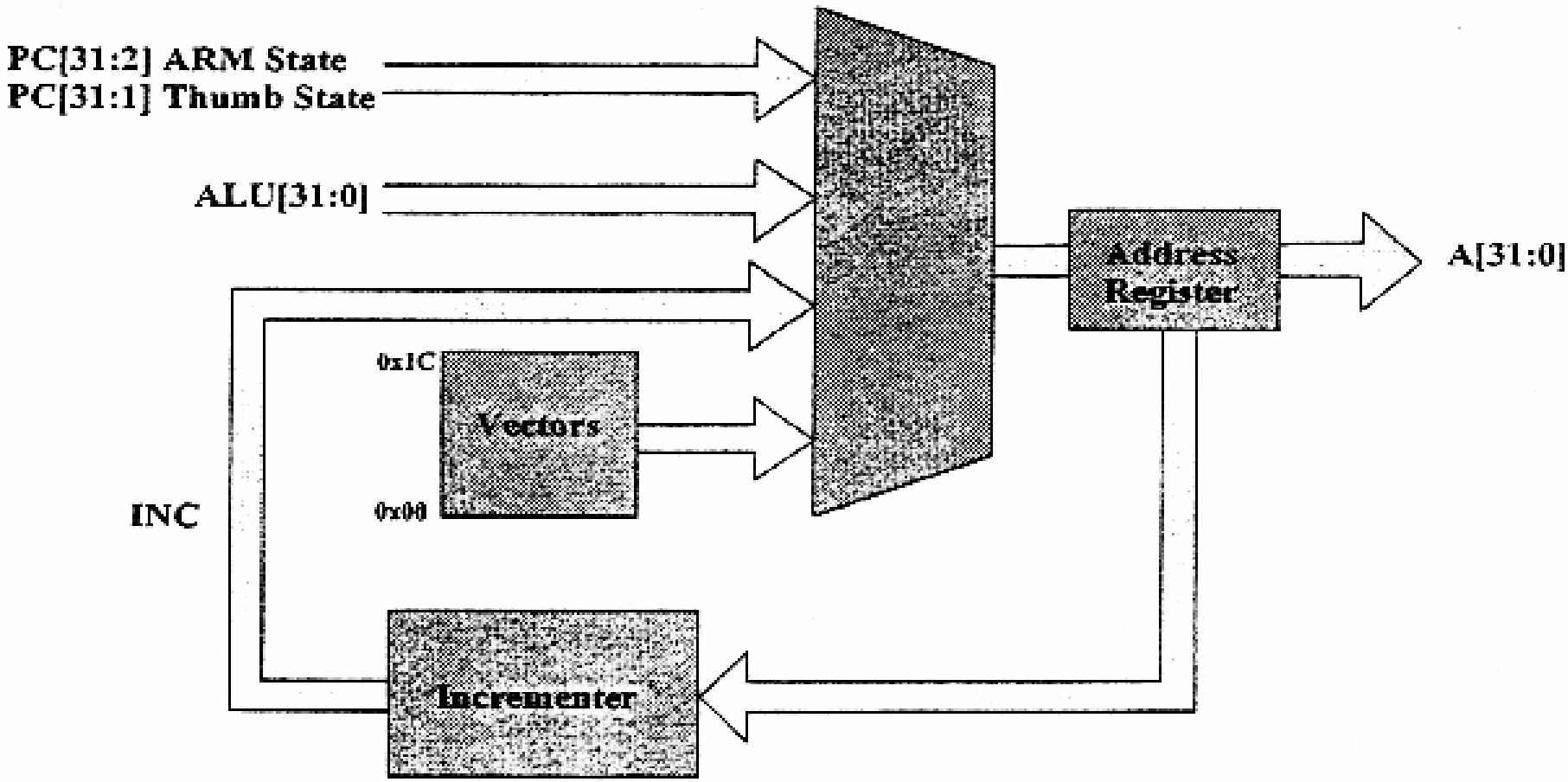
- MMU interface
  - \trans (translation control), 0:user mode, 1:privileged mode
  - \mode[4:0], bottom 5 bits of the CPSR (inverted)
  - abort, disallow access
- State
  - T bit, whether the processor is currently executing ARM or Thumb instructions
- Configuration
  - bigend, big-endian or little-endian

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

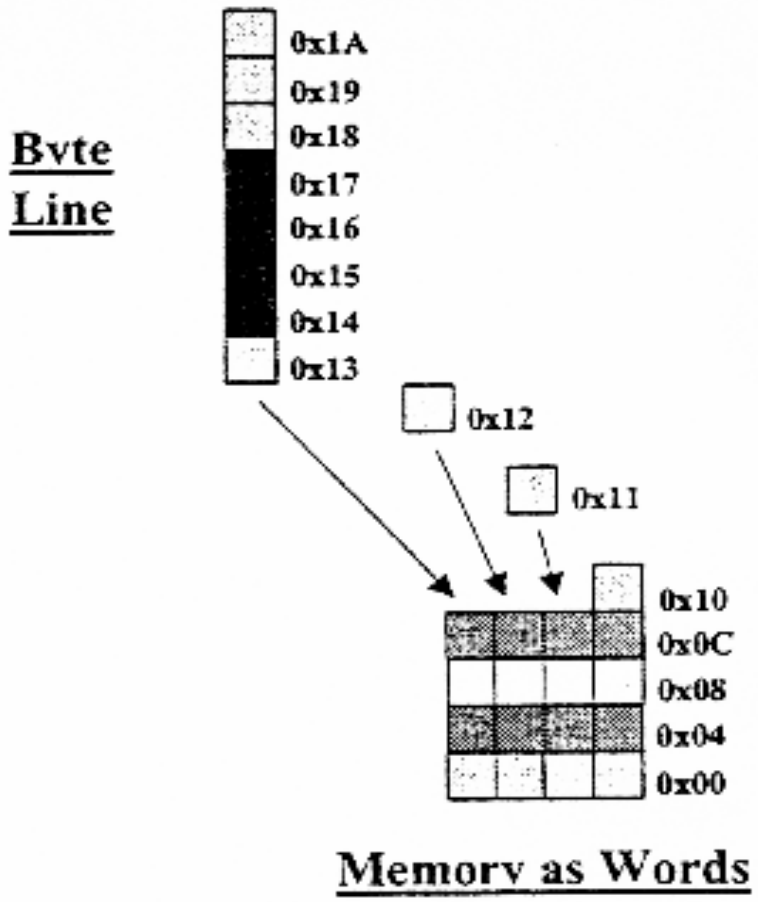
Process	0.35 µm	**Transistors**	74,209	**MIPS**	60
Metal layers	3	**Core area**	2.1 mm^2	**Power**	87 mW
Vdd	3.3 V	**Clock**	0–66 MHz	**MIPS/W**	690

External Address Generation



Memory Access

* **The ARM7 is a Von Neumann, load/store architecture. i.e.**
  - One 32 bit data bus for both instructions and data.
  - Only the load/store instructions (and SWP) access memory.
* **Memory is addressed as a 32 bit address space.**
* **Data types 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 = 0, SEQ = 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 = 0, SEQ = 0
  - The ARM core requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.
* **Internal (I cycle)**
  - nMREQ =1, SEQ = 0
  - The ARM core does not require a transfer, as it is performing an internal function, and no useful prefetching can be performed at the same time.
* **Coprocessor register transfer (C cycle)**
  - nMREQ = 1, SEQ = 1
  - The ARM core wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

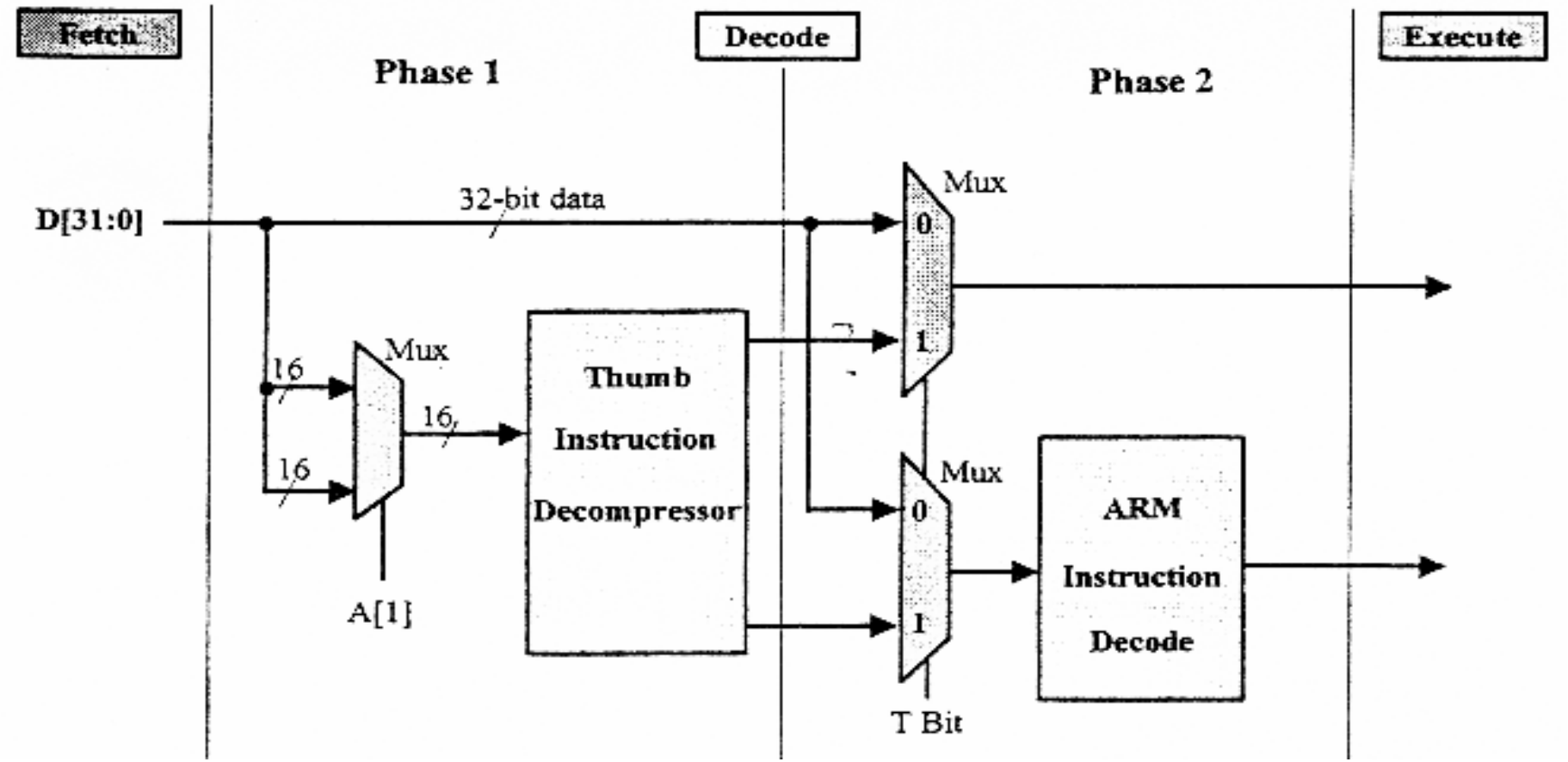
Instruction Execution Cycles (1/2)

Instruction	Qualifier	Cycle count
Any unexecuted	Condition codes fail	+S
Data processing	Single-cycle	+S
Data processing	Register-specified shift	+I +S
Data processing	R15 destination	+N +2S
Data processing	R15, register-specified shift	+I +N +2S
MUL		+(m)I +S
MLA		+I +(m)I +S
MULL		+(m)I +I +S
MLAL		+I +(m)I +I +S
B, BL		+N +2S
LDR	Non-R15 destination	+N +I +S

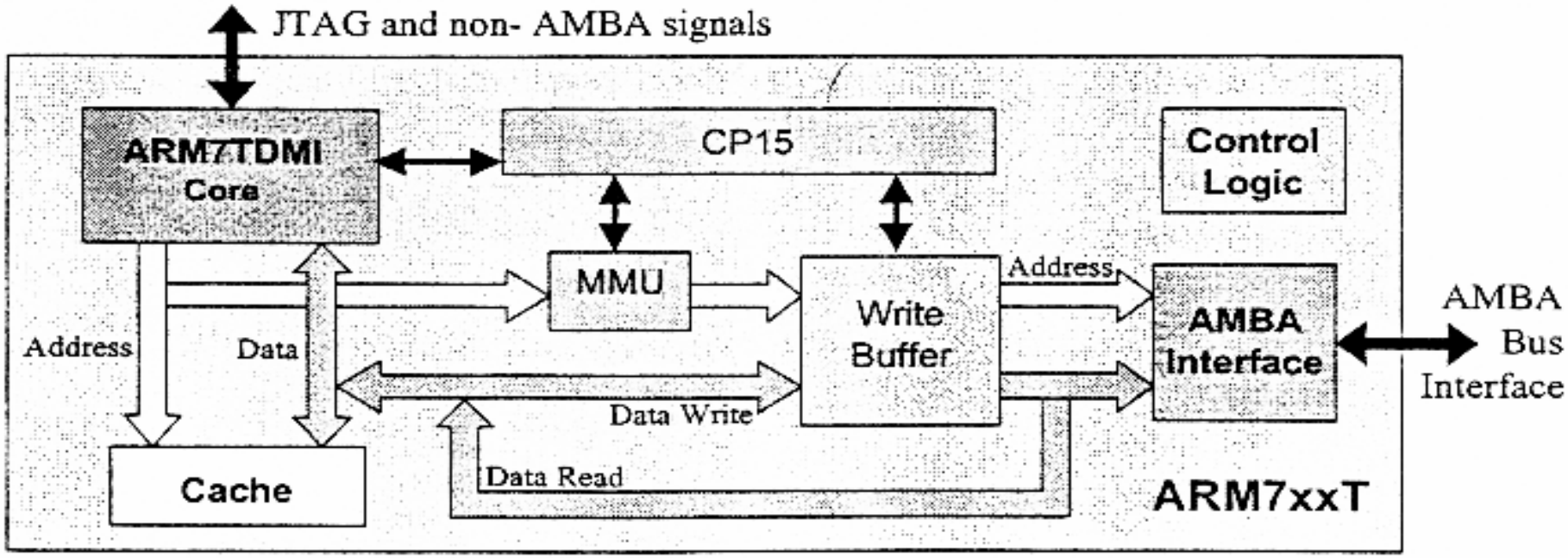
Instruction Execution Cycles (2/2)

Instruction	Qualifier	Cycle count
LDR	R15 destination	+N +I +N +2S
STR		+N +N
SWP		+N +N +I +S
LDM	Non-R15 destination	+N +(n-1)S +I +S
LDM	R15 destination	+N +(n-1)S +I +N +2S
STM		+N +(n-1)S +I +N
MSR, MRS		+S
SWI, trap		+N +2S
CDP		+(b)I +S
MCR		+(b)I +C +N
MRC		+(b)I +C +I +S
LDC, STC		+(b)I +N +(n-1)S +N

Effect of T bit



Cached ARM7TDMI Macrocells



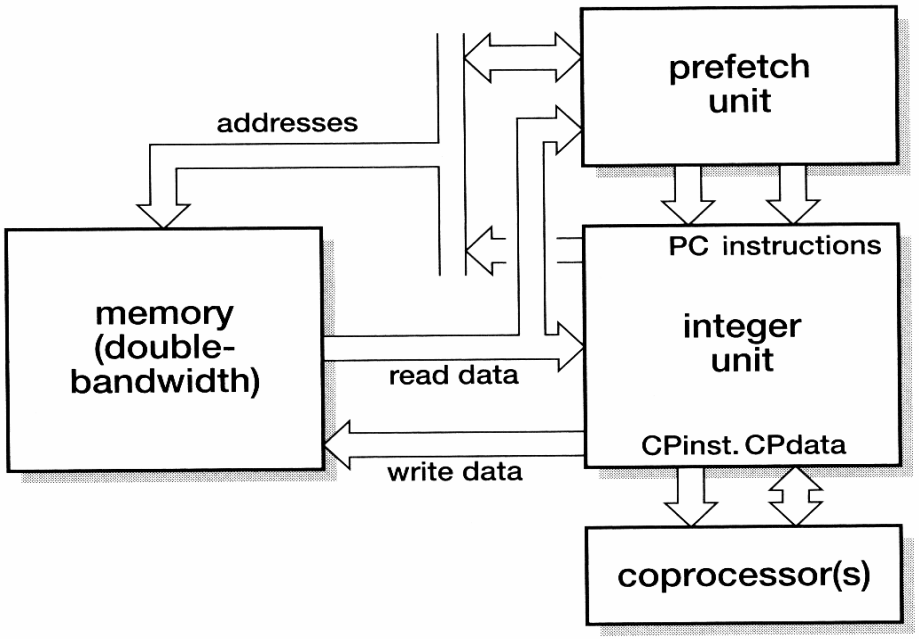
* **ARM710T**
  - 8K Unified write through cache
  - Full Memory Management Unit supporting virtual memory and memory protection
  - Write Buffer

* **ARM720T**
  - As ARM710T but with WinCE support

* **ARM740T**
  - 8K Unified write through cache
  - Memory Protection Unit
  - Write Buffer

ARM 8

- 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
- ARM8 → ARM9TDMI
  → ARM10TDMI
- 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

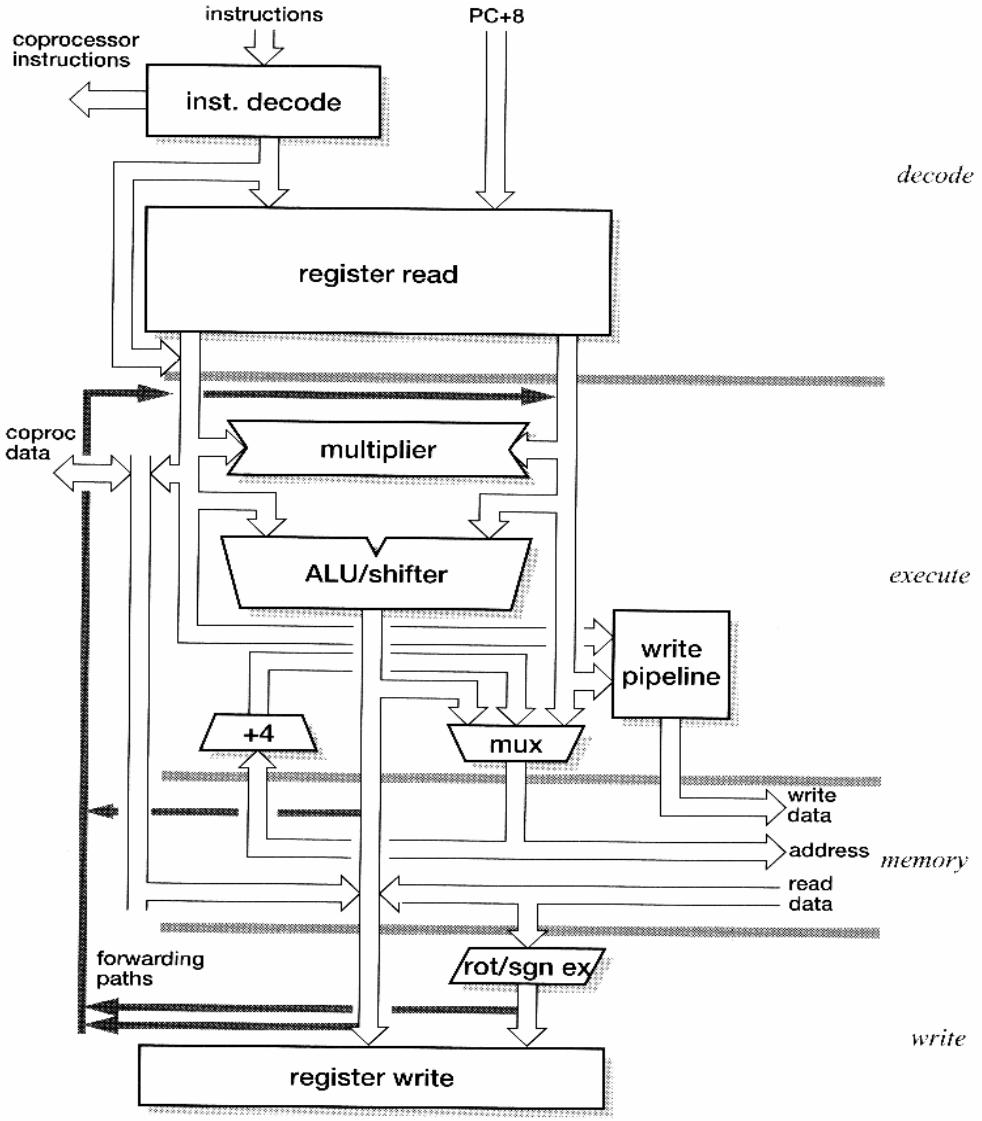
(2) Instruction decode and register read

(3) Execute (shift and ALU)

(4) Data memory access

(5) Write back results

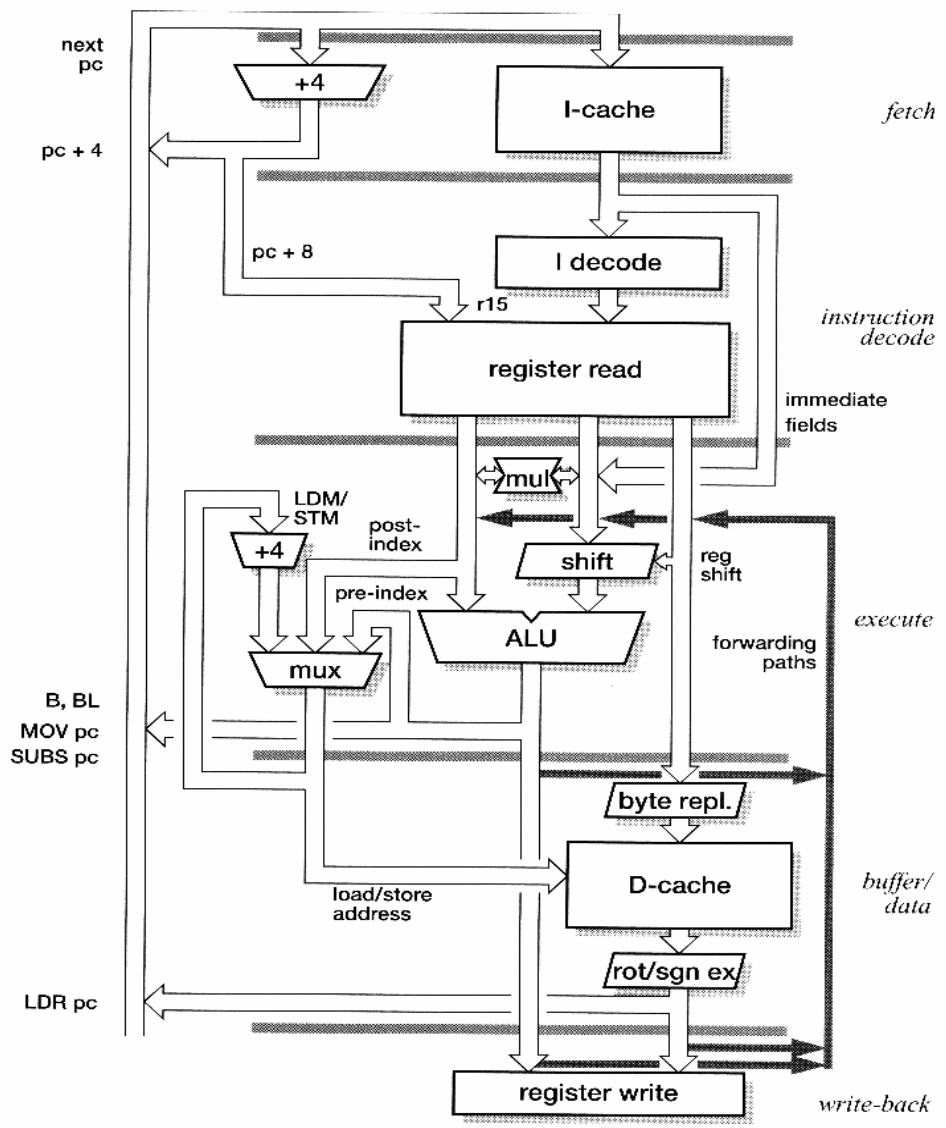
Integer Unit Organization



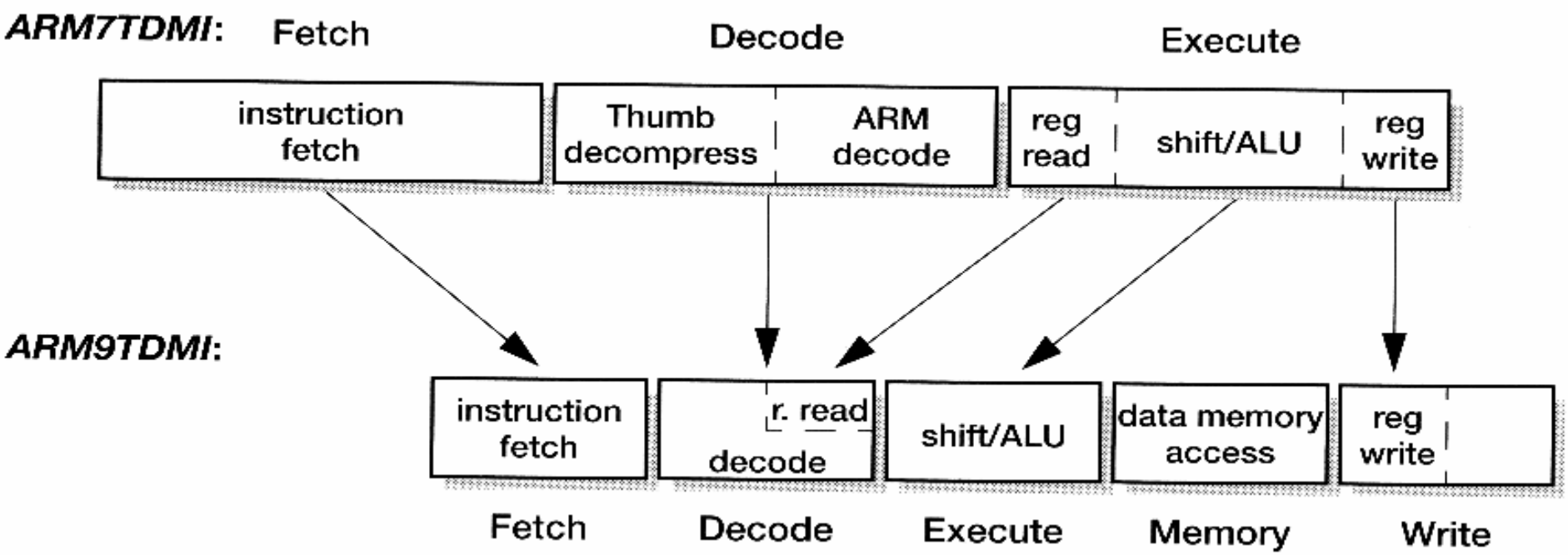
ARM9TDMI

- 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

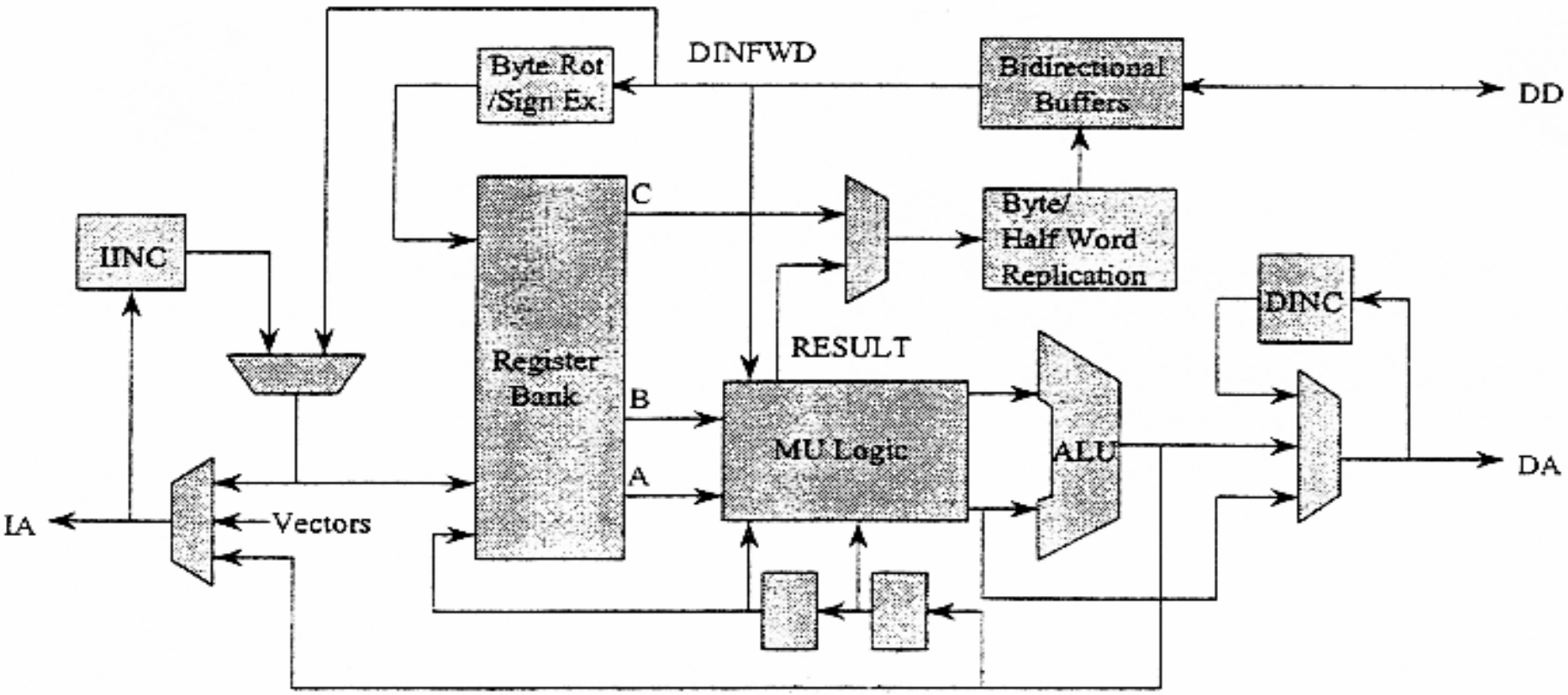


ARM9TDMI Pipeline Operations (1/2)

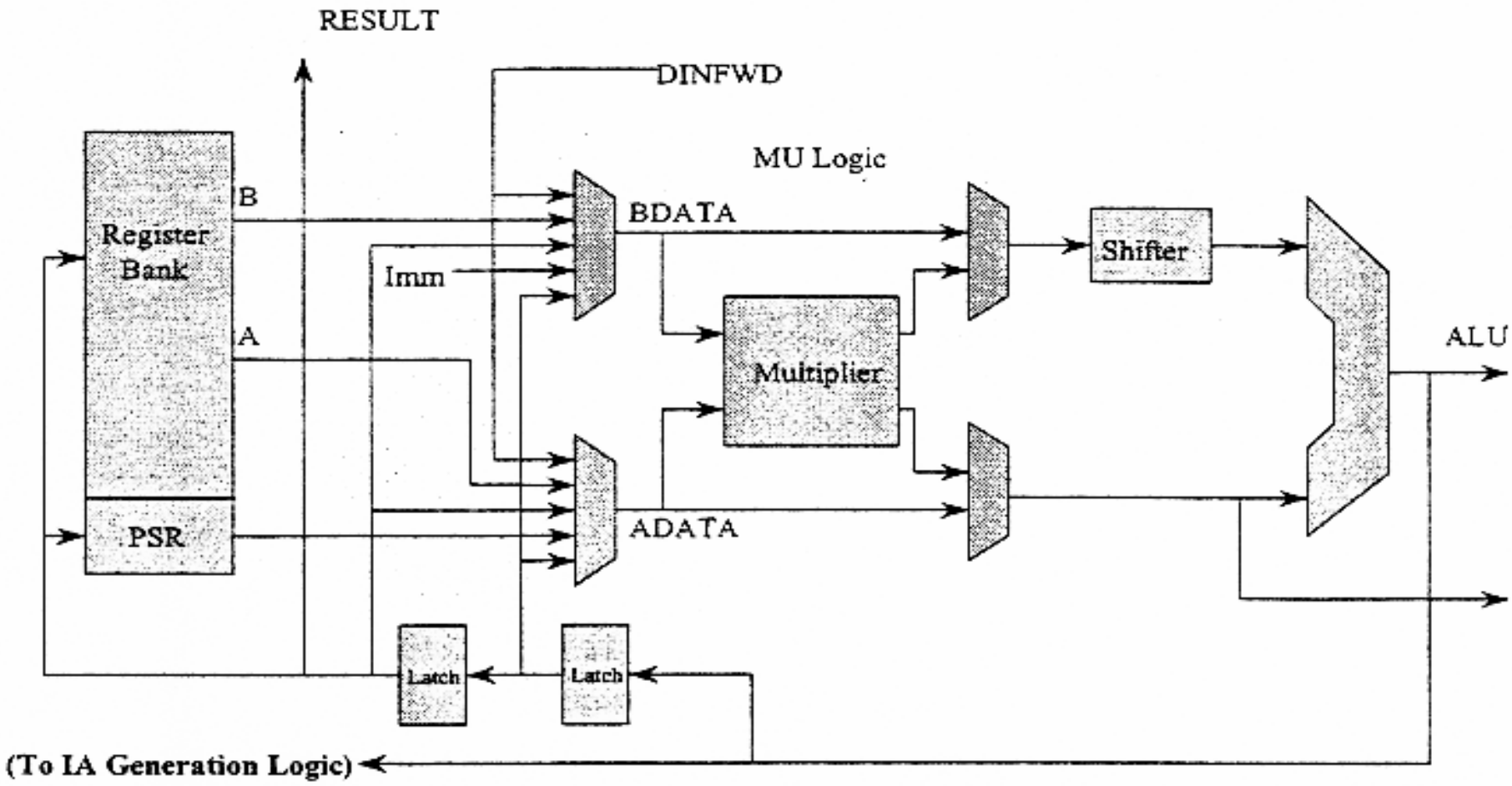


- Not sufficient slack time to translate Thumb instructions into ARM instructions and then decode, instead the hardware decode both ARM and Thumb instructions directly

ARM9TDMI Datapath (1/2)



ARM9TDMI Datapath (2/2)



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 - Execute* *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 causes 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 - Execute* *I - Interlock* *M - Memory*
W - Writeback

* In this example it takes 6 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
Operation												
LDMIA	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	

F - Fetch D - Decode E - Execute I - Interlock M - Memory
MW - 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

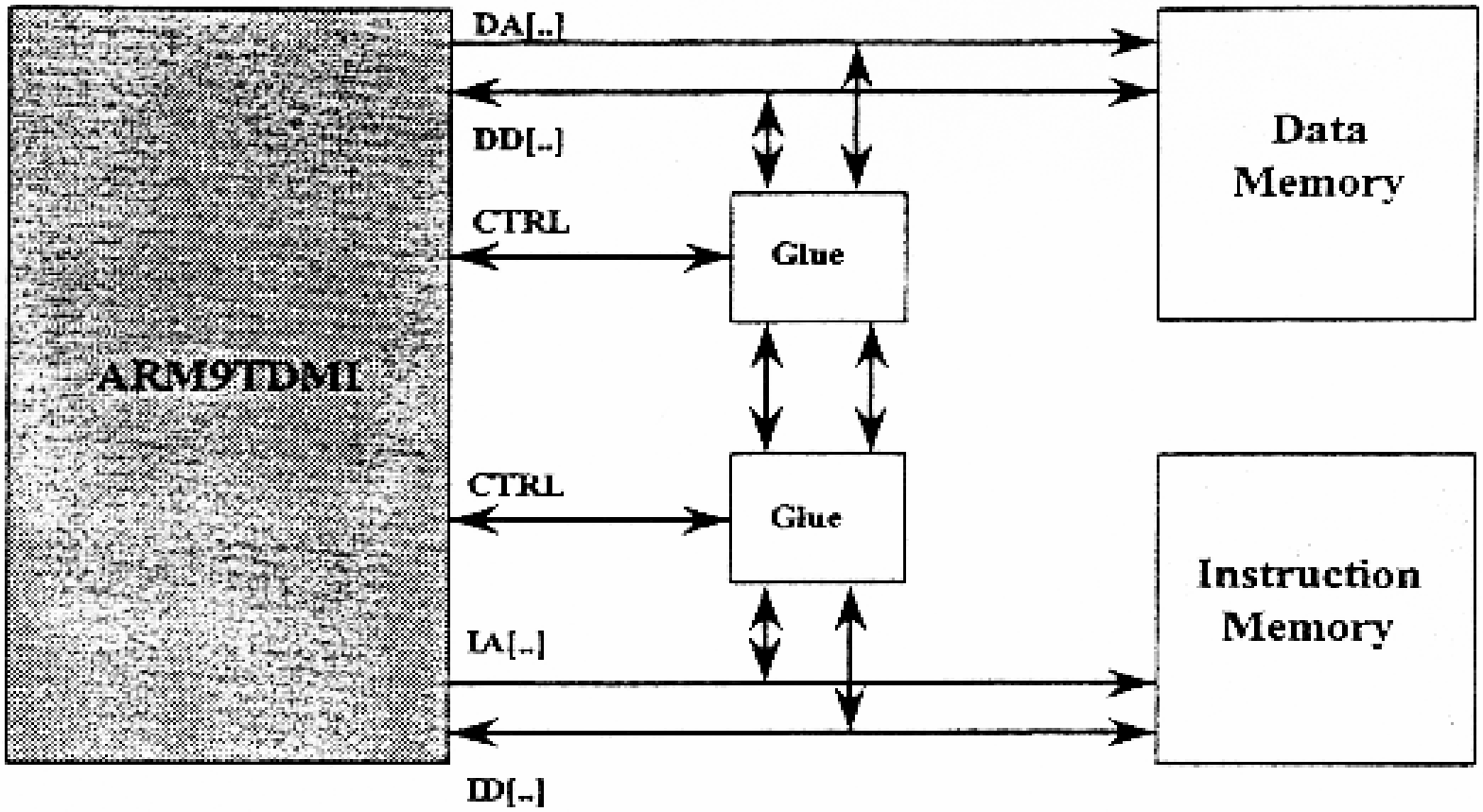
LDM Interlock (2/2)

Cycle				1	2	3	4	5	6	7	8	9	
Operation													
LDMIA	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 - Execute I - Interlock M - Memory
MW - Simultaneous Memory and Writeback W - Writeback

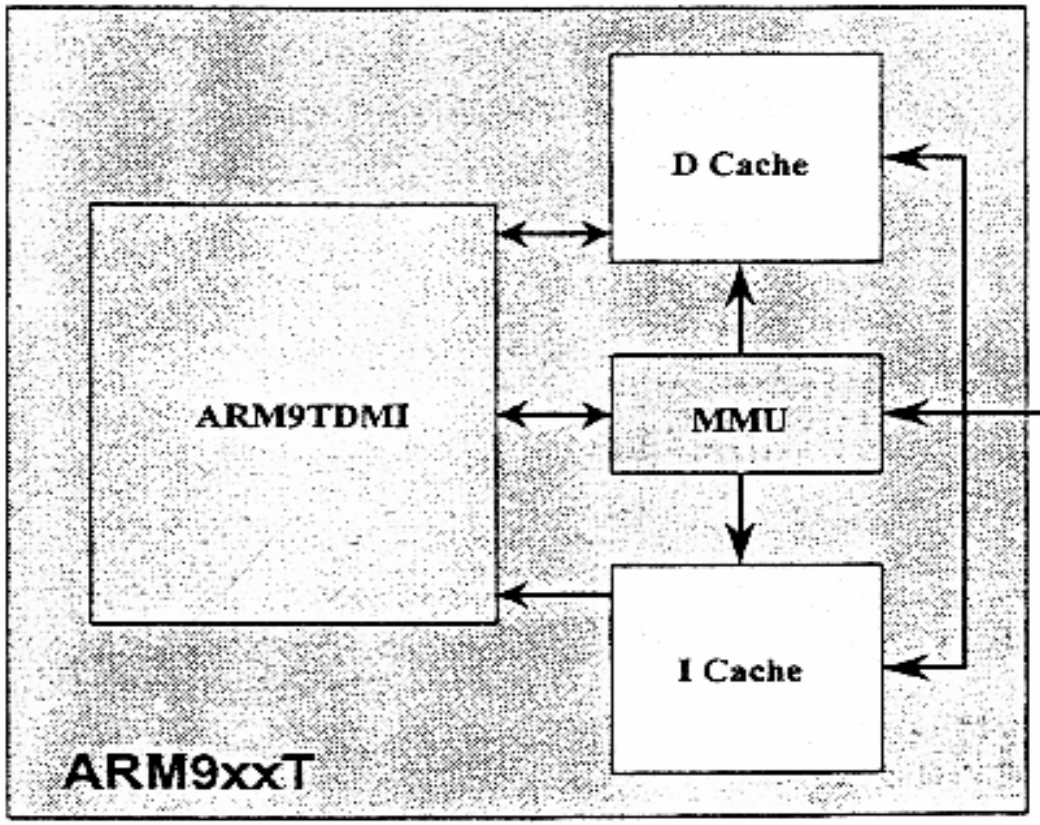
* **In this example it takes 9 clock cycles to execute 5 instructions, of 1.8**
* **The sub incurs a further cycle of interlock due to it using the highest specified register in the LDM instruction**
  - This would occur for any of the LDM variants, e.g. IA, DB, FD, etc.

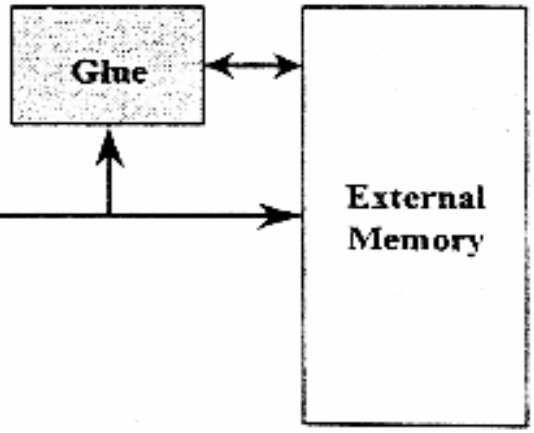
Example ARM9TDMI System



Note:
The data interface must have read access to the instruction memory for literal pool data. For debug purposes it is recommended that the data interface have read and write access to instruction memory

Cached ARM9TDMI Macrocell





* ARM920T
  - 2x 16K caches
  - Full Memory Management Unit supporting virtual addressing and memory protection
  - Write Buffer

* ARM940T
  - 2x 4K caches
  - Memory Protection Unit
  - Write Buffer

ARM9TDMI Pipeline Operations (2/2)

- Coprocessor support
  - coprocessors: floating-point, digital signal processing, special-purpose hardware accelerator
- On-chip debug
  - additional features compared to ARM7TDMI
    - hardware single stepping
    - breakpoint can be set on exceptions
- ARM9TDMI characteristics

Process	0.25 μm	**Transistors**	111,000	**MIPS**	220
Metal layers	3	**Core area**	2.1 mm^2	**Power**	150 mW
Vdd	2.5 V	**Clock**	0–200 MHz	**MIPS/W**	1500

ARM9E-S Family Overview

- ARM9E is based on an ARM9TDMI with the following extensions
  - single cycle 32*16 multiplier implementation
  - EmbeddedICE Logic RT
  - improved ARM/Thumb interworking
  - new 32*16 and 16*16 multiply instructions
  - new count leading zeros instruction
  - new saturated maths 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
- ARM966E-S
  - similar to ARM946-S, but with no cache

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

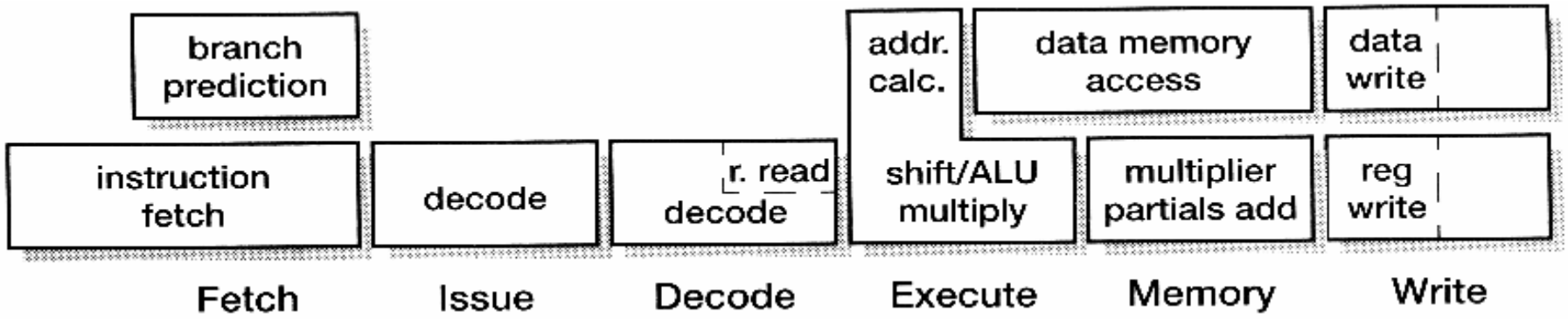
ARM10TDMI (1/2)

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

ARM10TDMI ← ARM9TDMI ← ARM7TDMI

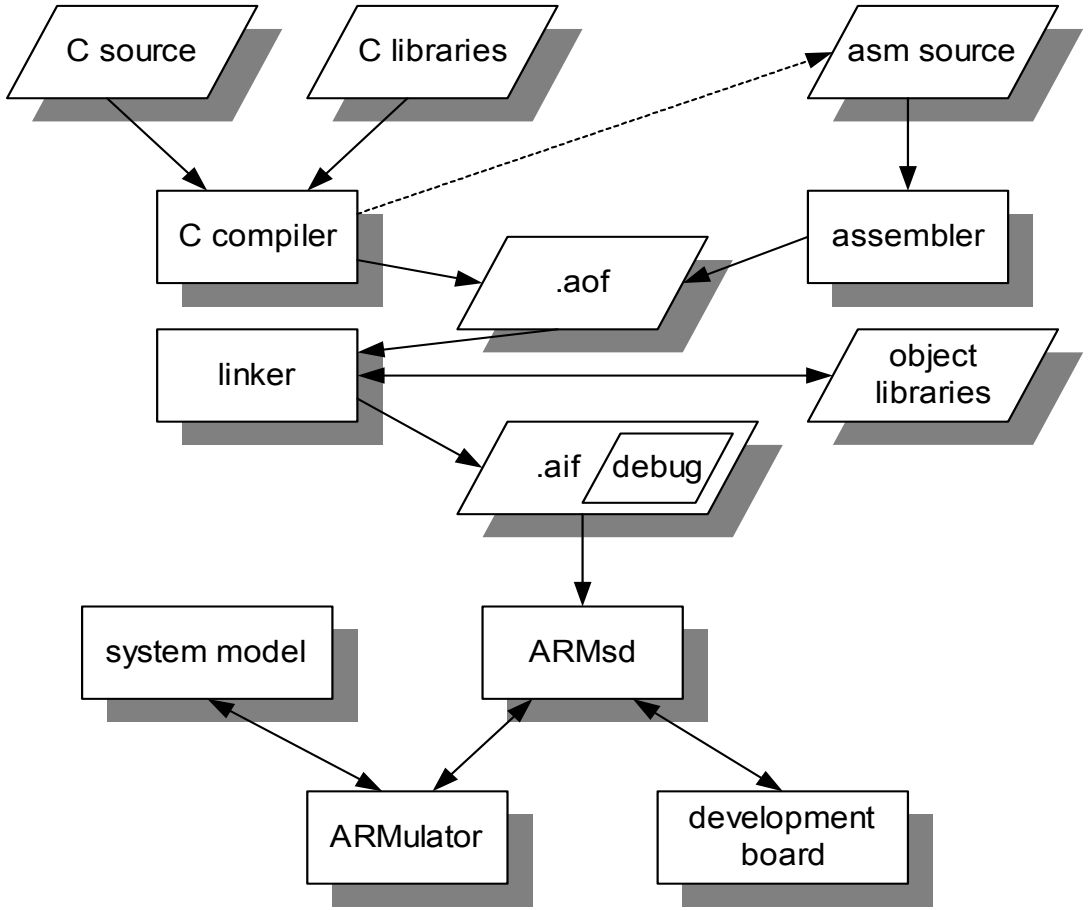
*2 *2

- 300MHz, 0.25uM CMOS
- Increase clock rate



Software Development

- 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

  armcpp ARM C++ compiler

  tcc Thumb C compiler

  tcpp Thumb C++ compiler

  armasm ARM and Thumb assembler

  armlink ARM linker

  - combine the contents of one or more object files with selected parts of one or more object libraries to produce an executable program
  - ARM linker creates ELF executable images

  armsd ARM and Thumb symbolic debugger

  - can single-step through C or assembly language sources, set break-points and watch-points, and examine program variables or memory

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 (ARMulator)
- AxD (ADW, ADU)
  - ARM debugger for Windows and Unix with graphics user interface
  - debug C, C++, and assembly language source

  Code Warrior 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

  Angel 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 emulation (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

- 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 accurate
  - cycle accurate
  - timing accurate

  => instruction count or number of cycles can be measured for a program

  => performance analysis
- Timing accurate model is used for cache, memory management unit analysis, and so on

ARM Development Board

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

Writing Assembly Language Programs

```
        AREA    HelloW,CODE,READONLY   ;declare code area
SWI_WriteC      EQU     &0             ;output character in r0
SWI_Exit        EQU     &11            ;finish program
        ENTRY                          ;code entry point
START   ADR     r1,TEXT                ;r1-> "Hello World"
LOOP    LDRB    r0,[r1],#1             ;get next byte
        CMP     r0,#0                  ;check for text end
        SWINE   SWI_WriteC             ;if not end print …
        BNE     LOOP                   ;… and loop back
        SWI     SWI_Exit               ;end of execution
TEXT    =       "Hello World",&0a,&0d,0
        END                            ;end of program source
```

- The following tools are needed
  - a text editor to type the program into
  - an assembler to translate the program into ARM binary code
  - an ARM system or emulator to execute the binary code
  - a debugger to see what is happening inside the code

Program Design

- Start with understanding the requirements, translate the requirements into an unambiguous specifications
- Define a program structure, the data structure and the algorithms that are used to perform the required operations on the data
- The algorithms may be expressed in pseudo-code
- Individual modules should be coded, tested and documented
- Nearly all programming is based on high-level languages, however it may be necessary to develop small software components in assembly language to get the best performance

System Architecture (1/2)

- ARM processor, memory system, buses, and the ARM reference peripheral specification
- The reference peripheral specification defines a basic set of components, providing a framework within which an operating system can run but leaving full scope for application-specific system
- Components include
  - a memory map
  - an interrupt control
  - a counter timer
  - a reset controller with defined boot behavior, power-on reset detection, a “wait for interrupt” pause mode
- The system must define
  - the base address of the interrupt controller (ICBase)
  - the base address of the counter-timer (CTBase)
  - the base address of the reset and pause controller (RPCBase)

  All the address of the registers are relative to one of the base addresses

System Architecture (2/2)

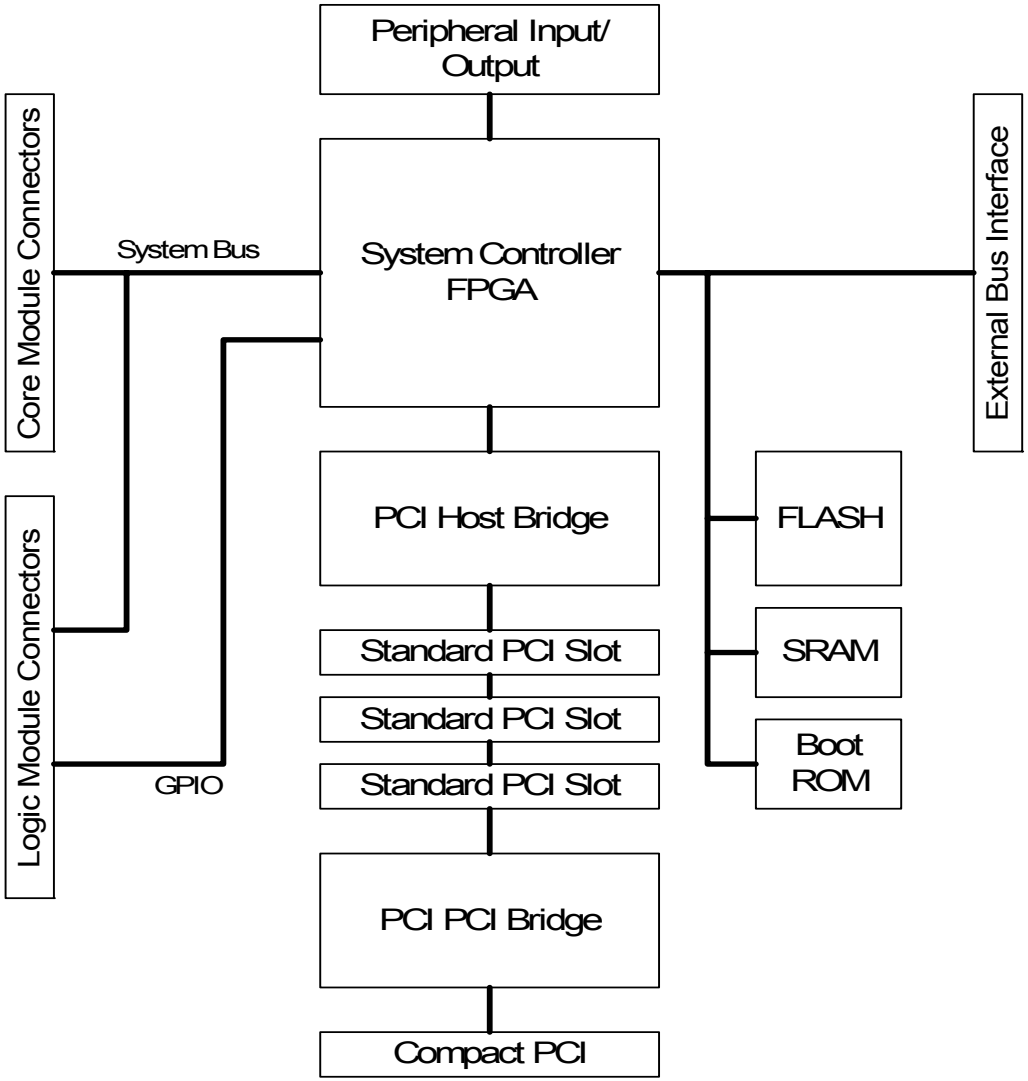
- Interrupt controller provides a way of enabling, disabling (by mask) and examining the status of up to 32 level-sensitive IRQ sources and one FIQ source
- Two 16-bit counter-timers, controlled by registers. The counters operate from the system clock with selectable pre-scaling
- Reset and pause controller includes some registers
  - the readable registers give identification and reset status information
  - the writable registers can set or clear the reset status, clear the reset map and put the system into pause mode where it uses minimal power until an interrupt wakes it up again
- Add application-specific peripherals

Hardware System Prototype

- Verifying the function correctness of hardware blocks, software modules(on-developing) and speed performance is acceptable
- Simulating the system using software tools => slower, can't verify the full system
- Hardware Prototyping
  - building a hardware platform by pre-existing or on-developing components for system verification and software development
  - "ARM Integrator" or "Rapid Silicon Prototyping"

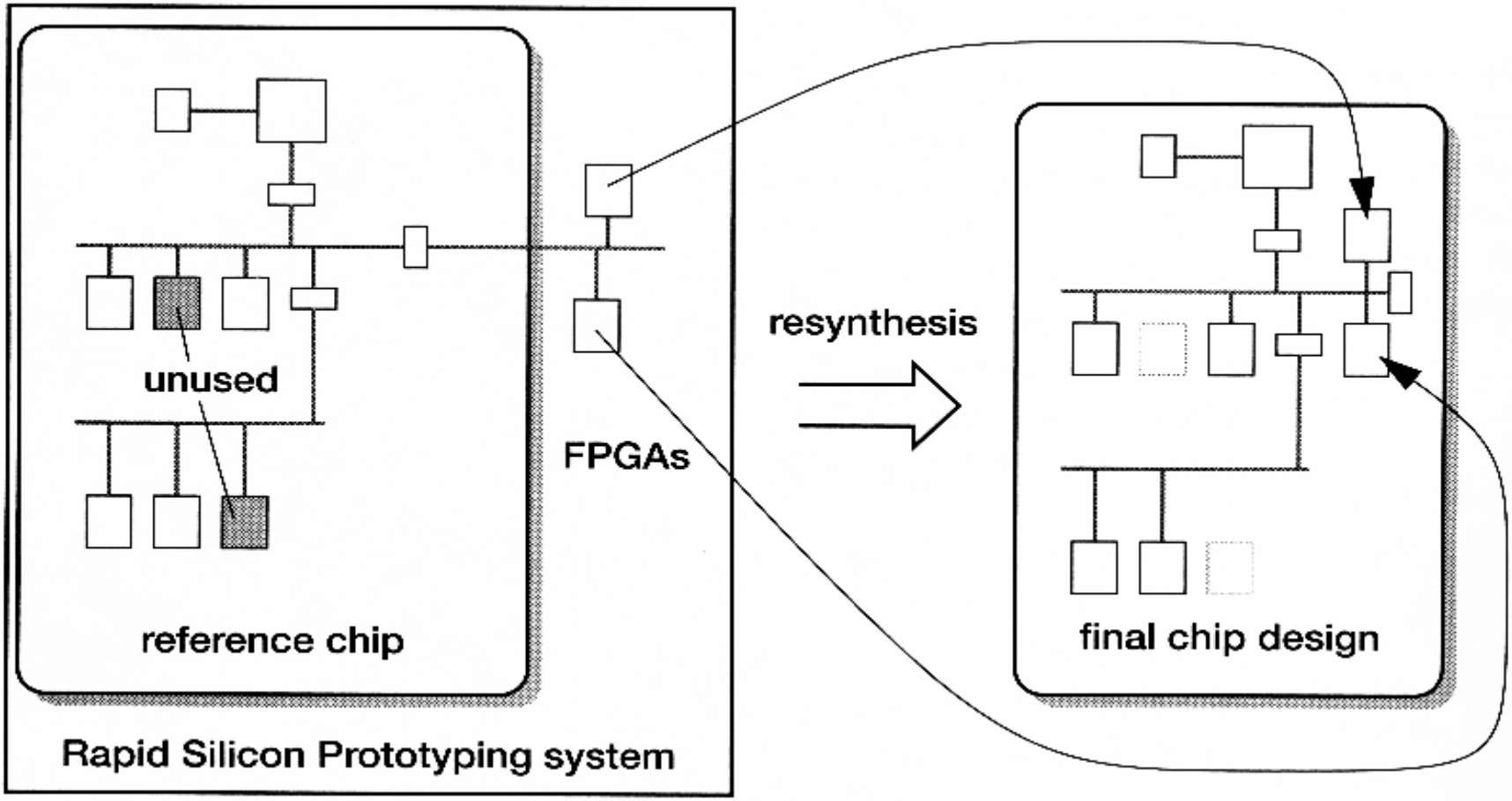
ARM Integrator

- A motherboard with some extensions to support the development of applications
- Provide core modules, logic modules (Xilinx Virtex FPGA), OS, input/output resources, bus arbitration, interrupt handling



Rapid Silicon Prototyping (VLSI Tech. Inc.)

- Specially developed reference chips + off-chip extensions



ARMulator (1/2)

- ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors (It is an instruction set simulator)
- ARMulator is suited to software development and benchmarking ARM-targeted software. It models the instruction set and counts cycles.
- 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

ARMulator (2/2)

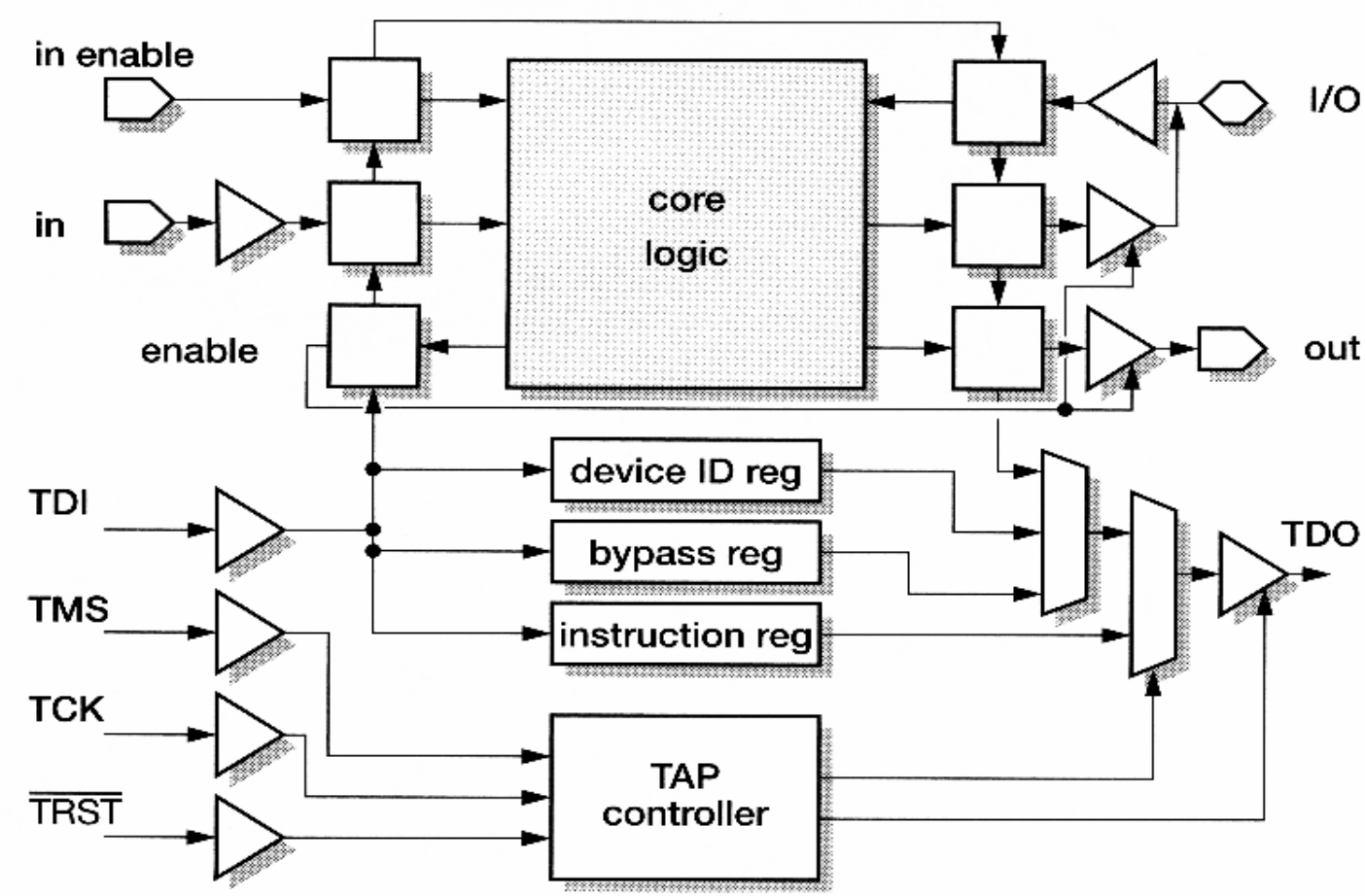
- It includes
  - processor core models which can emulate any ARM core
  - a memory interface which allows the characteristics of the target memory system to be modeled
  - a coprocessor interface that supports custom coprocessor models
  - an OS interface that allows individual system calls to be handled
- The processor core model incorporates the remote debug interface, so the processor and the system state are visible from the ARM symbolic debugger
- ARMulator => a cycle accurate model of a system including a cache, MMU, physical memory, peripheral devices, OS, software
- Once the design is OK,

  hardware -> design or synthesis by CAD

  software -> still use ARMulator model, but instruction accurate

JTAG Boundary Scan (1/2)

- IEEE 1149, Standard Test Access Port and Boundary Scan Architecture or called JTAG boundary scan (by Joint Test Action Group)



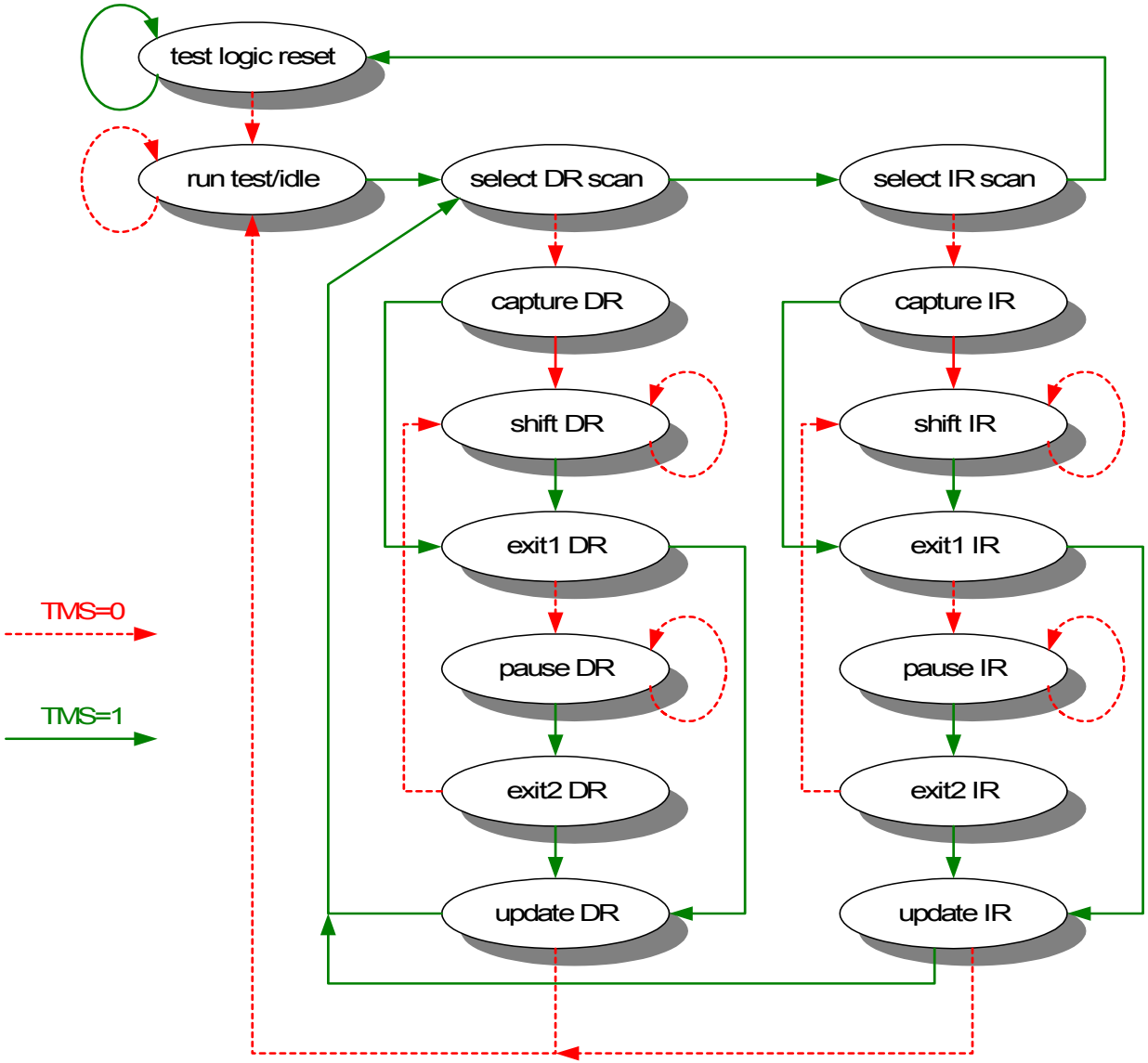
JTAG Boundary Scan (2/2)

- Test signals
  - \TRST: a test reset input
  - TCK: test clock which controls the timing of the test interface independently from any system clock
  - TMS: test mode select which controls the operation of the test interface state machine
  - TDI: test data input line
  - TDO: test data output line

- TAP controller (Test Access Port)

  A state machine whose state transitions are controlled by TMS

TAP Controller (1/2)

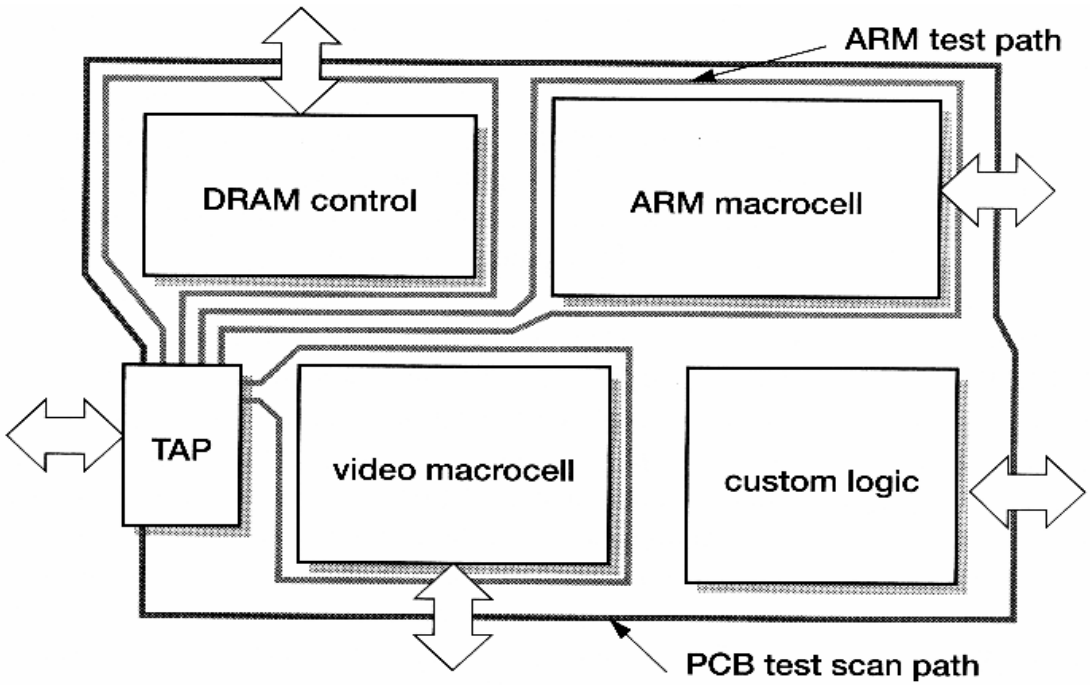


TAP Controller (2/2)

- Test instruction selects various data registers
  - device ID register, bypass register, boundary scan register
- Some public instructions
  - **BYPASS**: connect TDI to TDO with 1-clock delay
  - **EXTEST**: test the board-level connectivity, boundary scan register is connected
    - **capture DR**: captured by the boundary scan register
    - **shift DR**: shift out via TDO
    - **update DR**: new data applied to the boundary scan register via TDI
  - **TNTEST**: test the core logic
  - **INCODE**: ID register is connect

Macrocell Testing

- System chip is composed of the pre-designed macrocells with application-specific custom logic
- Various approaches to test the macrocells
  - test mode provided which multiplexes the signals in turn onto the chip
  - on-chip bus may support direct test access to macrocell pins
  - each macrocell may have a boundary scan path using JTAG extensions



ARM Debug Architecture (1/2)

- Two basic approaches to debug
  - from the outside, use a logic analyzer
  - from the inside, tools supporting single stepping, breakpoint setting
- **Breakpoint**: replacing an instruction with a call to the debugger

  Watchpoint: a memory address which halts execution if it is accessed as a data transfer address

  Debug Request: through ICEBreaker programming or by DBGRQ pin asynchronously

ARM Debug Architecture (2/2)

- In debug state, the core's internal state and the system's external state may be examined. Once examination is complete, the core and system state may be restored and program execution is resumed.
- The internal state is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the core's pipeline without using the external data bus.
- When in debug state, a store-multiple (STM) could be inserted into the instruction pipeline and this would dump the contents of ARM's registers.

Debugger (1/2)

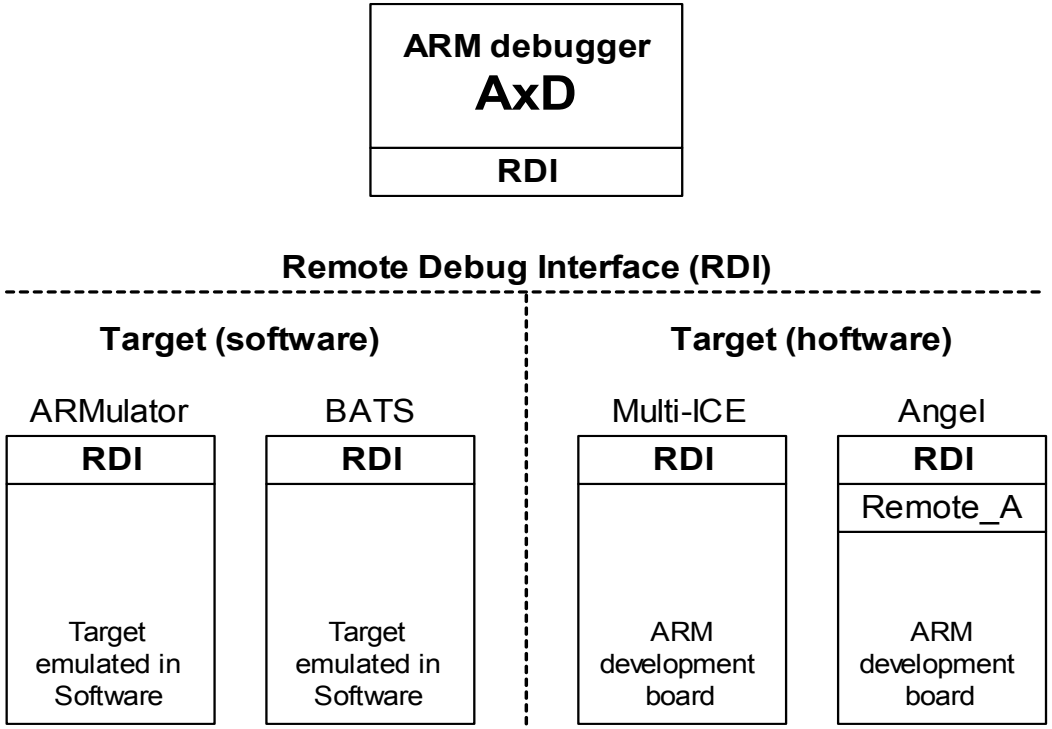
- A debugger is software that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target
- Different forms of the debug target
  - early stage of product development, software
  - prototype, on a PCB including one or more processors
  - final product
- The debugger issues instructions that can
  - load software into memory on the target
  - start and stop execution of that software
  - display the contents of memory, registers, and variables
  - allow you to change stored values
- A debug agent performs the actions requested by the debugger, such as
  - setting breakpoints
  - reading from / writing to memory

Debugger (2/2)

Examples of debug agents

- Multi-ICE
- Embedded ICE
- ARMulator
- BATS
- Angle

- Remote Debug Interface (RDI) is an open ARM standard procedural interface between a debugger and the debug agent



In Circuit Emulator (ICE)

- The processor in the target system is removed and replaced by a connection to an emulator
- The emulator may be based around the same processor chip, or a variant with more pins, but it will also incorporate buffers to copy the bus activity to a “trace buffer” and various hardware resources which can watch for particular events, such as execution passing through a breakpoint

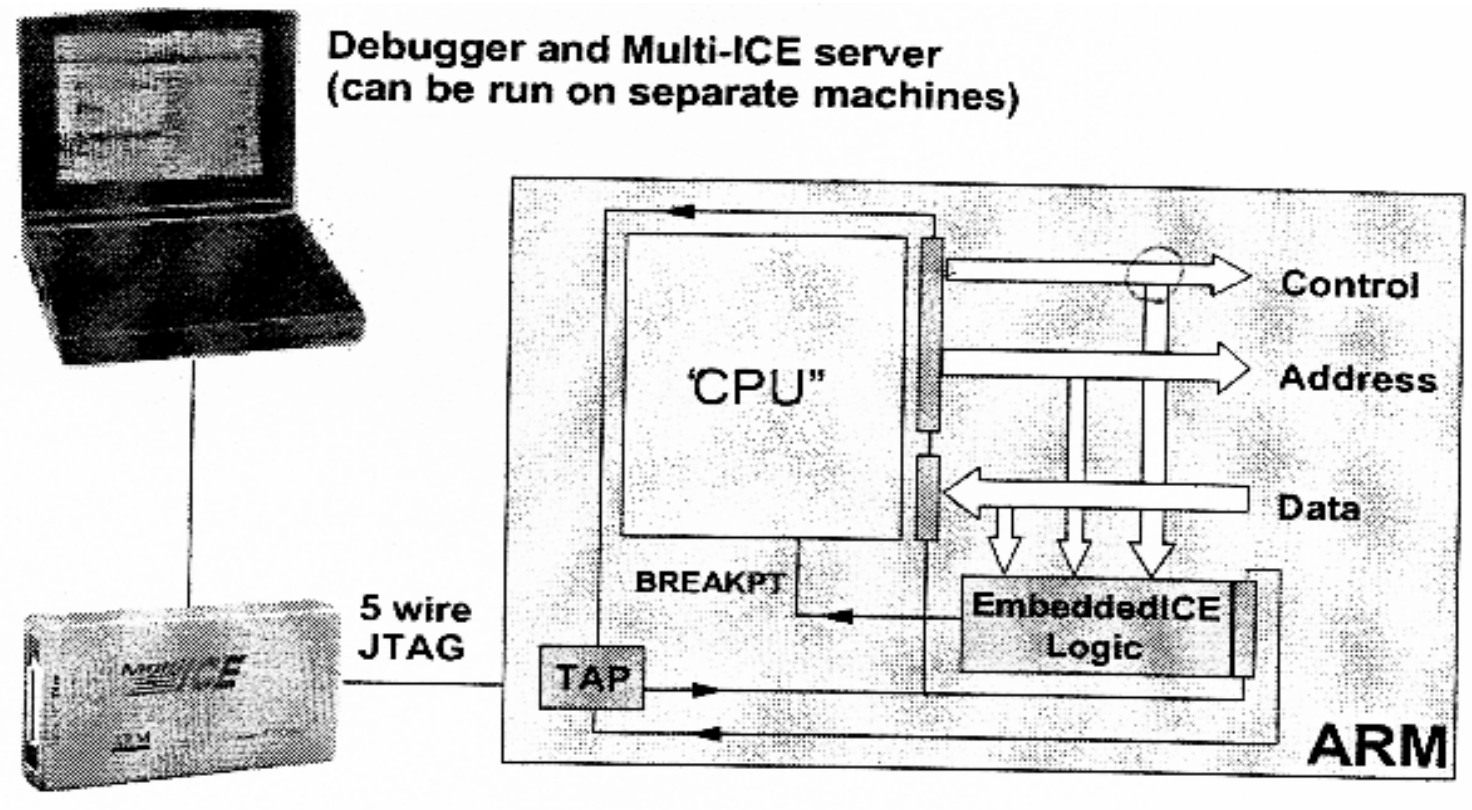
Multi-ICE and Embedded ICE

- Multi-ICE and Embedded ICE are JTAG-based debugging systems for ARM processors
- They provide the interface between a debugger and an ARM core embedded within an ASIC
- It provides
  - real time address-dependent and data-dependent breakpoints
  - single stepping
  - full access to, and control of the ARM core
  - full access to the ASIC system
  - full memory access (read and write)
  - full I/O system access (read and write)

Basic Debug Requirements

- Control of program execution
  - set watchpoints on interesting data accesses
  - set breakpoints on interesting instructions
  - single step through code
- Examine and change processor state
  - read and write register values
- Examine and change system state
  - access to system memory
    - download initial code

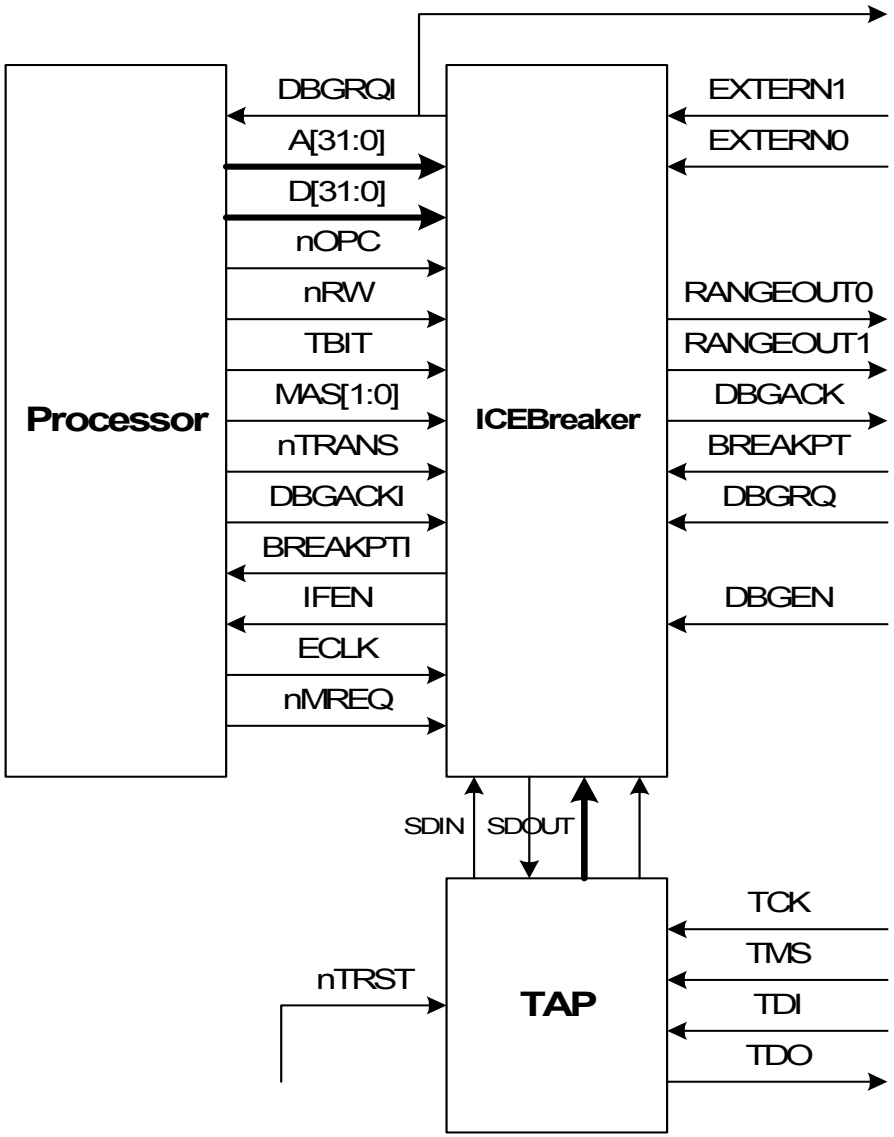
Debugging with Multi-ICE



- The system being debugged may be the final system
- Third party protocol converters are also available at
  `http://www.arm.com/DevSupp/ICE_Analyz/`

ICEBreaker (EmbeddedICE macrocell)

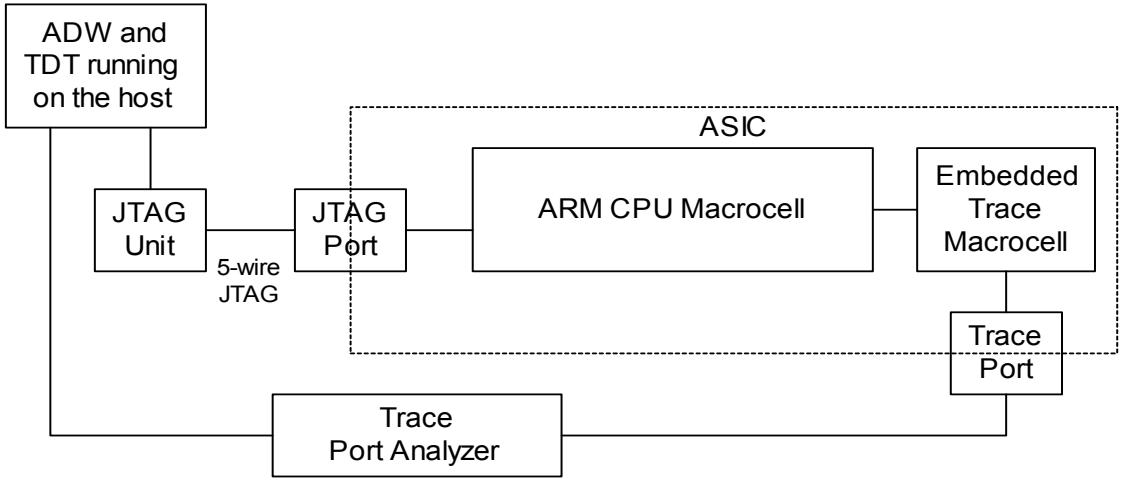
- ICEBreaker is programmed in a serial fashion using the TAP controller
- It consists of 2 real-time watch-point units, together with a control and status register
- Either watch-point unit can be configured to be a watch-point or a breakpoint



Real-Time Trace (1/2)

- Debugging uses the breakpoint and single-step to run application code to a given point, and then stop the processor to examine or change memory or register contents, and then step or restart the code
- Some bugs occur while the system is running at full clock speed => need non-instrusive trace of instruction flow and data accesses
- Using Trace Debug Tool (TDT), you can set up the trace filter facility to collect trace data only during the interrupt routine, and use a trigger to stop tracing

Real-Time Trace (2/2)



- Embedded trace macrocell
  - monitor the ARM core buses, passed compressed information through the trace port to Trace Port Analyzer (TPA)
  - the on-chip cell contains the trigger and filter logic
- Trace port analyzer
  - an external device that stores the information from the trace port
- Trace debug tool
  - set up the trigger and filter logic, retrieve the data from the analyzer and reconstruct a historical view of processor activity

ARM10TDMI (2/2)

- Reduce CPI
  - branch prediction
  - non-blocking load and store execution
  - 64-bit data memory => transfer 2 registers in each cycle