

ARM 구조

- ARM v7 중심으로 -

민 홍

최근 SoC 동향

- 삼성 – Exynos

Model number	CPU instruction set	CPU	GPU
Exynos 3 Single ^[10] (previously S5PC110, Hummingbird, Exynos 3110 ^[11])	ARMv7	0.8–1.2 GHz Single-core ARM Cortex-A8	IT PowerVR SGX540 at 200 MHz (3.2 GFLOPS) ^[12]
Exynos 4 Dual 45 nm ^[8] (previously Exynos 4210 ^[13])	ARMv7	1.2–1.4 GHz Dual-core ARM Cortex-A9	ARM Mali-400 MP4 (Quad-Core) at 266 MHz (9.6 GFLOPS) ^[12]
Exynos 4 Quad ^[19] (Internally Exynos 4412) ^[20]	ARMv7	1.4–1.6 GHz Quad-core ARM Cortex-A9	ARM Mali-400 MP4 ^[21] (Quad-Core) at 440 MHz (15.84 GFLOPS) ^[12]
Exynos 5 Dual ^[26] (previously Exynos 5250 ^[27])	ARMv7	1.7 GHz Dual-core ARM Cortex-A15	ARM Mali-T604 ^[28] (Quad-core) at 533 MHz 4 x 17 x 0.533 x 2 = 72.488 GFLOPS ^[citation needed]
Exynos 5 Octa ^{[32][33]} ^[34] (Internally Exynos 5410)	ARMv7	1.6–1.8 GHz quad-core ARM Cortex-A15 and 1.2 GHz quad-core ARM Cortex-A7 (ARM big.LITTLE) ^[35]	IT PowerVR SGX544MP3 (Tri-Core) at 533 MHz (51.2 GFLOPS) ^{[36][37]}

<출처: Wikipedia>

최근 SoC 동향

- Apple – A series

Name	CPU ISA	CPU	GPU
A4	ARMv7	0.8–1.0 GHz single-core Cortex-A8	PowerVR SGX535 @ 200–250 MHz (1.6–2 GFLOPS) ^[43]
A5	ARMv7	0.8–1.0 GHz dual-core Cortex-A9	PowerVR SGX543MP2 (dual-core) @ 200 MHz * vec4 + 1 scalar: $4 \times 2 + 1 = 9 \times 8 \times 0.200 \times 9 = 14.4$ GFLOPS ^[44]
A6	ARMv7s	1.3 GHz ^[47] dual-core Swift ^[40]	PowerVR SGX543MP3 (tri-core) @ 250 MHz * vec4 + 1 scalar: $4 \times 2 + 1 = 9 \times 12 \times 0.250 \times 9 = 27$ GFLOPS ^[44]

<출처: Wikipedia>

최근 SoC 동향

- Qualcomm - Snapdragon

Snapdragon S4 [\[edit\]](#)

Model Tier	Model Number	CPU Instruction Set	GPU
Pro	APQ8064 [45]	ARMv7	Adreno 320 (QXGA/1080p)

Snapdragon 600 [\[edit\]](#)

Model Tier	Model Number	CPU Instruction Set	CPU	GPU
600	APQ8064T	ARMv7	1.7 Up To 1.9 GHz Quad-core Krait 300	Adreno 320 (QXGA/1080p)

<출처: Wikipedia>

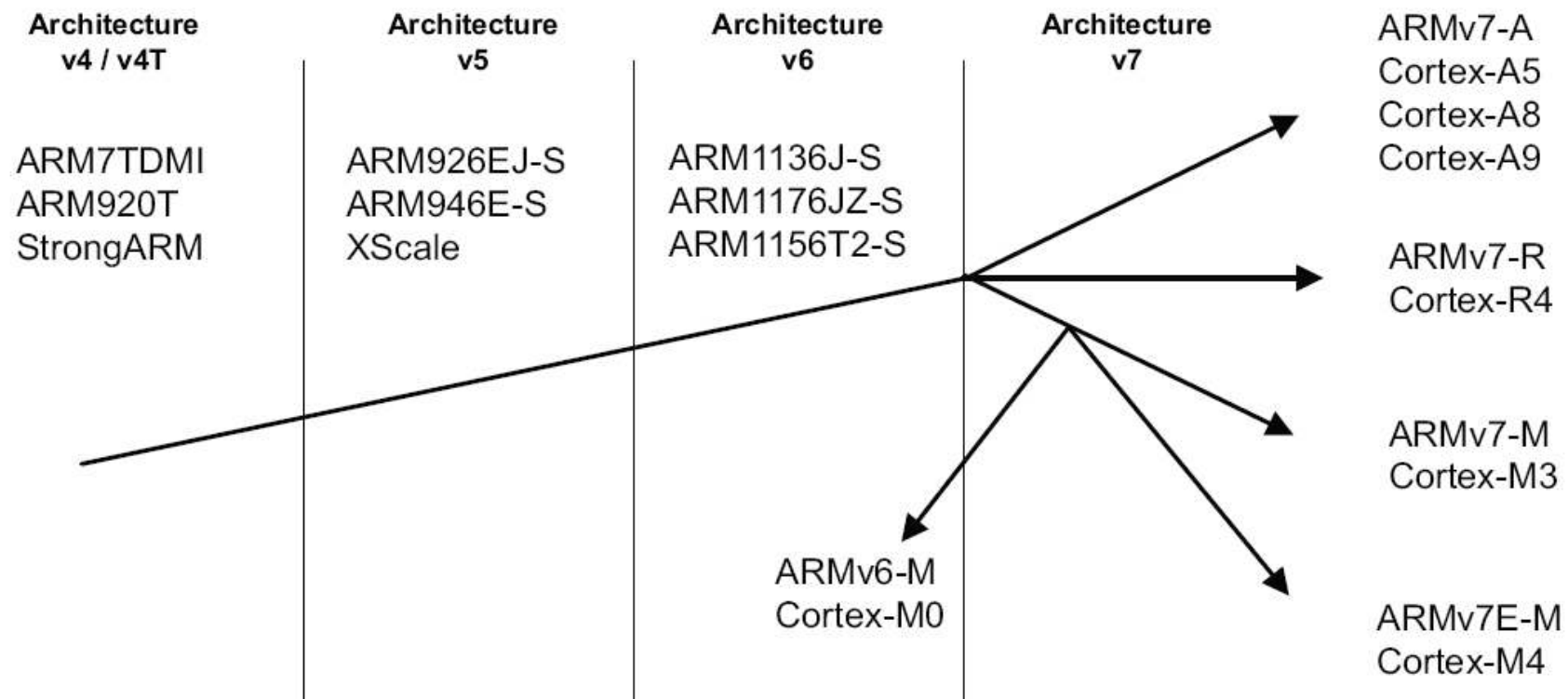
CISC vs. RISC

- CISC
 - H/W 의존적
 - 성능 최적화를 위해서는 칩 설계를 잘해야 함
- RISC
 - S/W 의존적
 - 성능이 컴파일러에 의해 결정

ARM Cortex – A/R/M

- A** The *Application* profile defines an architecture aimed at high performance processors, supporting a virtual memory system using a *Memory Management Unit* (MMU) and therefore capable of running complex operating systems. Support for the ARM and Thumb instruction sets is provided.
- R** The *Real-time* profile defines an architecture aimed at systems that need deterministic timing and low interrupt latency and which do not need support for a virtual memory system and MMU, but instead use a simpler memory protection unit (MPU).
- M** The *Microcontroller* profile defines an architecture aimed at lower cost/performance systems, where low-latency interrupt processing is vital. It uses a different exception handling model to the other profiles and supports only a variant of the Thumb instruction set.

Architecture and Family



Features

4T

Halfword and signed
halfword/byte support

System mode

Thumb instruction
set

5

Improved ARM/Thumb
Interworking

CLZ

Saturated arithmetic

DSP multiply-accumulate
Instructions

Extensions:
Jazelle (v5TEJ)

6

SIMD instructions

Multi-processing

v6 memory architecture

Unaligned data support

Extensions:
Thumb-2 (v6T2)
TrustZone (v6Z)
Multiprocessor (v6K)
Thumb only (v6-M)

7

Thumb technology

NEON

TrustZone

Profiles:
v7-A (Applications)
NEON

v7-R (Real-time)
Hardware divide
NEON

v7-M (Microcontroller)
Hardware divide
Thumb only

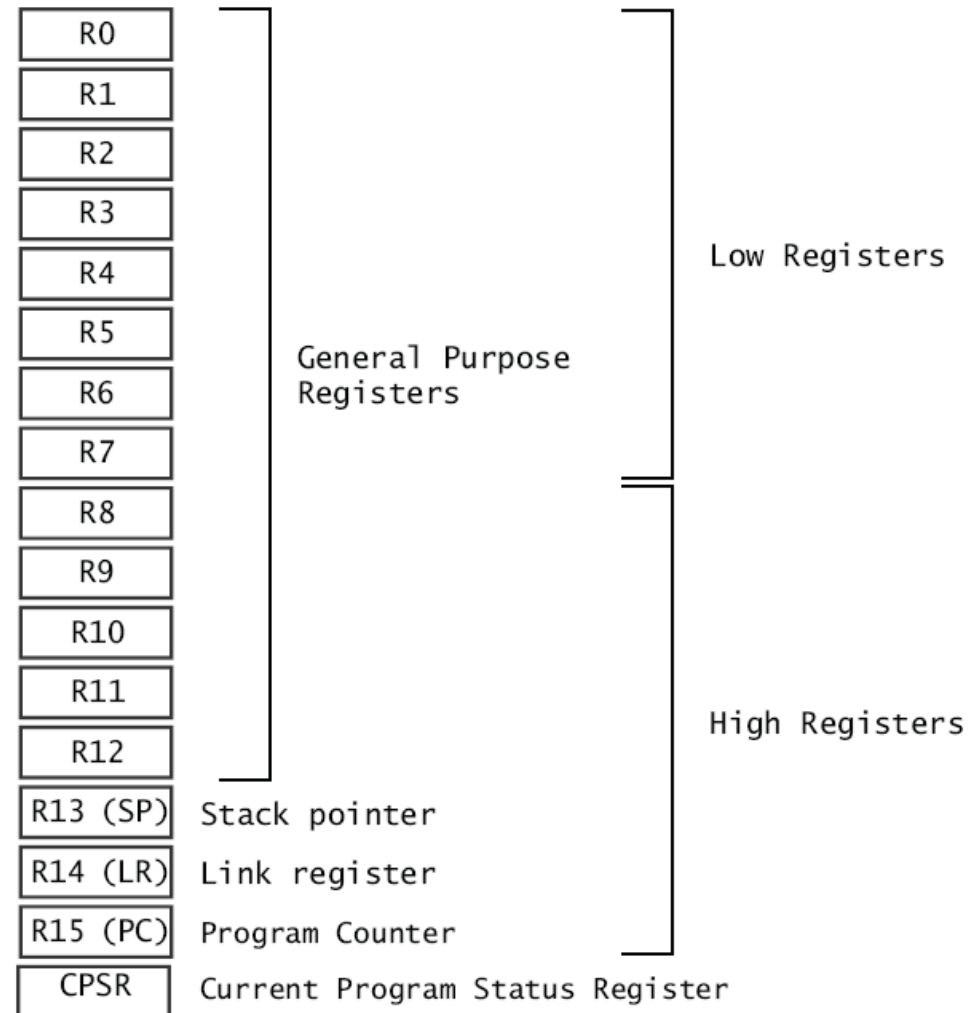
ARM processor mode

Mode	Mode encoding in the PSRs	Function
Supervisor (SVC)	10011	Entered on reset or when a Supervisor Call instruction (SVC) is executed
FIQ	10001	Entered on a fast interrupt exception
IRQ	10010	Entered on a normal interrupt exception
Abort (ABT)	10111	Entered on a memory access violation
Undef (UND)	11011	Entered when an undefined instruction executed
System (SYS)	11111	Privileged mode, which uses the same registers as User mode
User (USR)	10000	Unprivileged mode in which most applications run

Register set

R0	User mode R0-R7, R15 and CPSR	User mode R0-R12, R15 and CPSR	User mode R0-R12, R15 and CPSR	User mode R0-R12, R15 and CPSR	User mode R0-R12, R15 and CPSR	User mode R0-R12, R15 and CPSR
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8	R8					
R9	R9					
R10	R10					
R11	R11					
R12	R12					
R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)
R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)
R15 (PC)						
CPSR						
	SPSR	SPSR	SPSR	SPSR	SPSR	SPSR
User	FIQ	IRQ	ABT	SVC	UND	MON

Programmer Visible Registers

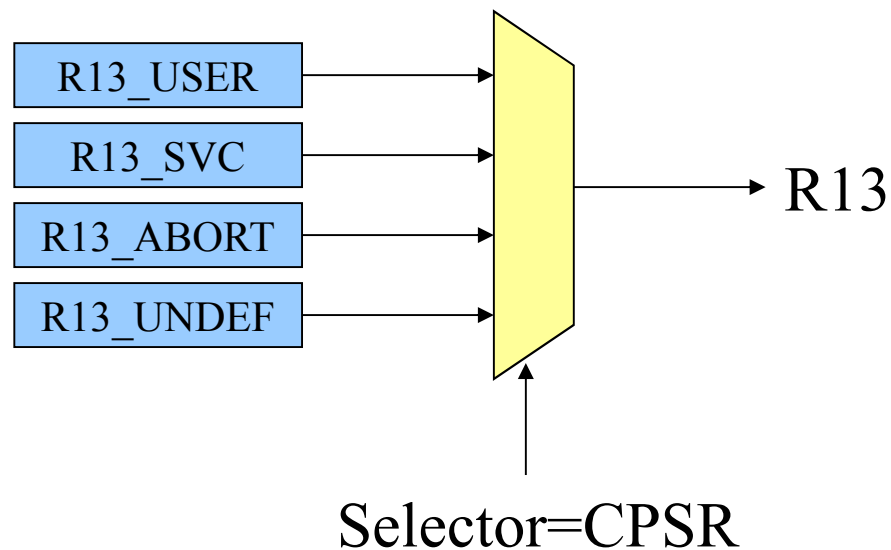


Register Set

- **Visible register set**
 - Registers that are visible during specific mode
 - 16x32bit registers are visible at any mode
 - Some registers are shared, some are not
- **Banked register**
 - Registers that share the same index
 - Only 1 of banked registers are visible at each mode
 - R13(SP) and R14(LR) are banked
 - FIQ has 5 additional banked registers
 - Register dump overhead is reduced at context switch

Banked Register

- Processor mode에 따라 다른 instance가 access되는 register 구조



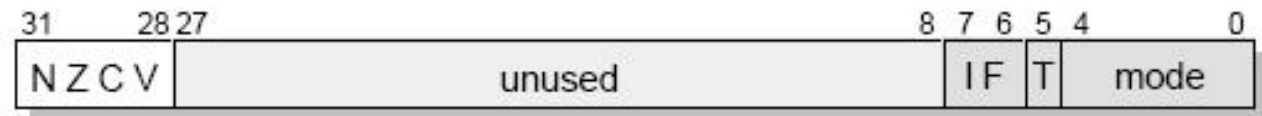
Special Registers

- **R13, Stack pointer**
 - Used when stack are implemented
 - Used when context switch occurs
 - Stores the stack pointer value of tasks
- **R14, Link Register**
 - Used when mode change with return occurs
 - Stores the return address (current PC)
- **R15, Program Counter**
 - Used to store current instruction address
 - A write to R15 is equivalent to branch instruction

CPSR – ARMv4

- **Flag bits – instruction's result**

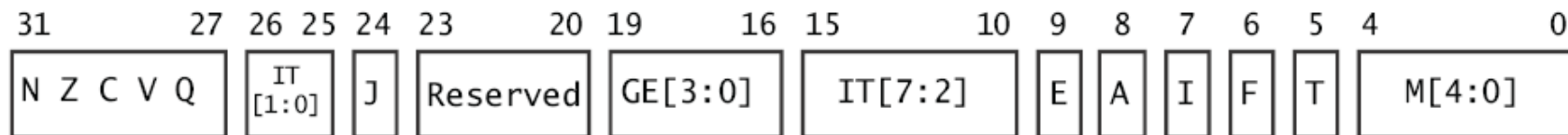
- **N: Negative**; the last ALU operation which changed the flags produced a negative result
- **Z: Zero**; the last ALU operation which changed the flags produced a zero result
- **C: Carry**; the last ALU operation which changed the flags generated a carry-out,
- **V: oVerflow**;



- **Control bits – cpu operating mode**

- Processor mode[0-4]
- Instruction set: Thumb mode
- Interrupt enables (I: IRQ, F: FIQ)

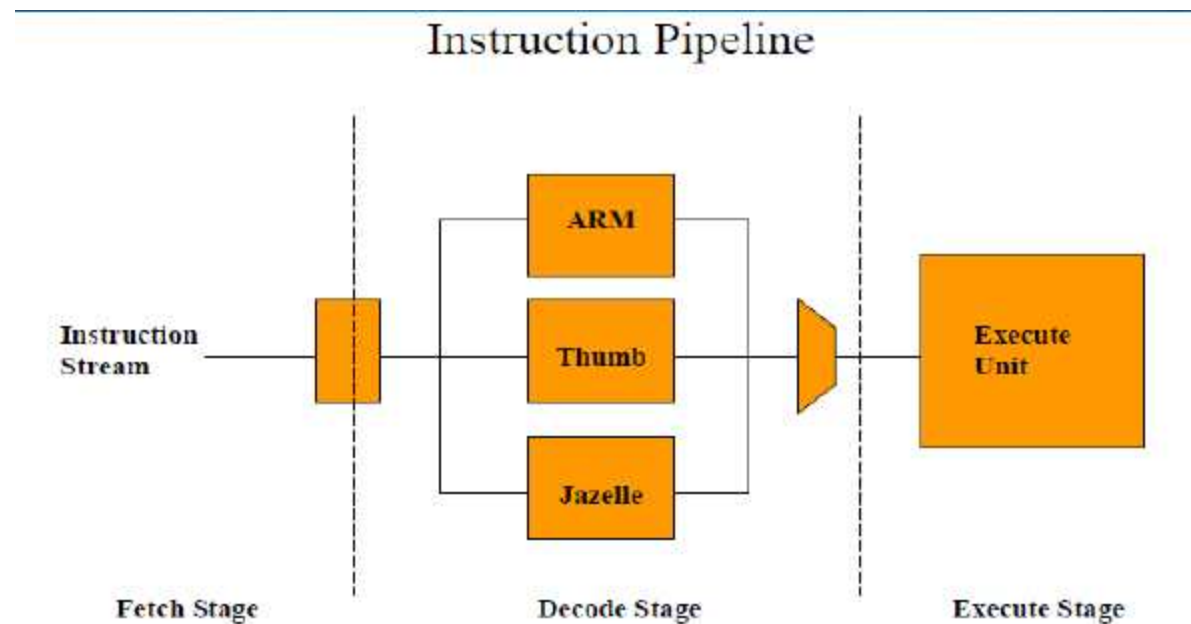
CPSR – ARMv7



- *N* – Negative result from ALU
- *Z* – Zero result from ALU
- *C* – ALU operation Carry out
- *V* – ALU operation oVerflowed
- *Q* – cumulative saturation (also described as “sticky”)
- *J* – indicates if processor is in Jazelle state
- *GE*[3:0] – used by some SIMD instructions
- *IT* [7:2] – If-Then conditional execution of Thumb-2 instruction groups
- *E* bit controls load/store endianness
- *A* bit disables imprecise data aborts
- *I* bit disables IRQ
- *F* bit disables FIQ
- *T* bit – *T* = 1 indicates processor in Thumb state
- *M*[4:0] – specifies the processor mode

Jazelle DBX (Direct Bytecode eXecution)

- Execute Java bytecode in hardware as a third execution state alongside the existing ARM and Thumb modes



Jazelle RCT (Runtime Compilation Target)

- A different technology and is based on ThumbEE mode and supports ahead-of-time (AOT) and just-in-time (JIT) compilation with Java
- Set modes
 - 2-bit: J, T at CPSR

J	T	ISA

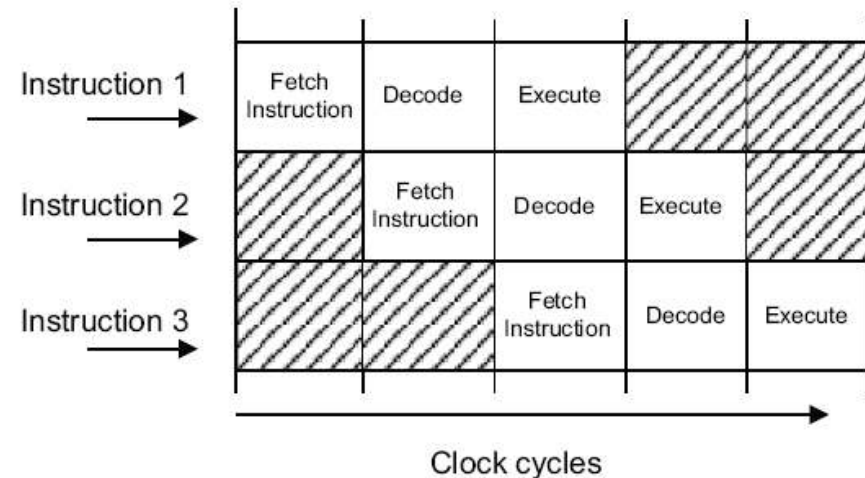
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	ThumbEE

Execute Java Byte Code

- Directly executed in Hardware
 - 95% of all bytecode instructions
 - Common, simple JVM instructions
 - Reduces number of interpretations
- Interpreted in Software
 - JVM instructions not implemented in Hardware
 - Interpreted into short sequence of optimized ARM instructions

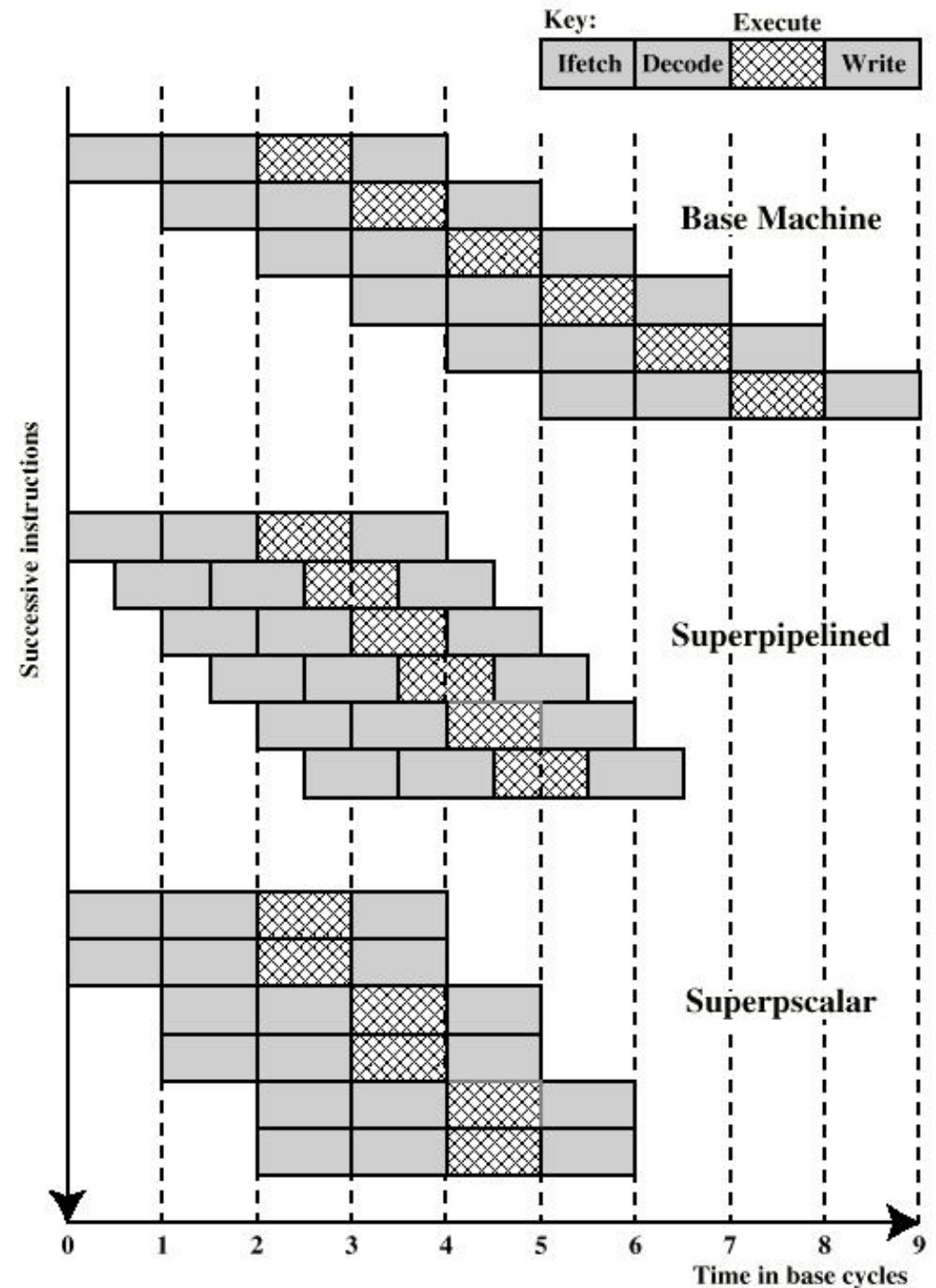
Instruction Fetch

Pipeline



- Instruction prefetch (deciding from which locations in memory instructions are to be fetched, and performing associated bus accesses).
- Instruction fetch (reading instructions to be executed from the memory system).
- Instruction decode (working out what instruction is to be executed and generating appropriate control signals for the datapaths).
- Register fetch (providing the correct register values to act upon).
- Issue (issuing the instruction to the appropriate execute unit).
- Execute (the actual ALU or multiplier operation, for example).
- Memory access (performing data loads or stores).
- Register write-back (updating processor registers with the results).

Pipeline vs. Superpipelined vs. Superscalar



Limitations

- Instruction level parallelism
- Compiler based optimisation
- Hardware techniques
- Limited by
 - True data dependency
 - Procedural dependency
 - Resource conflicts

True Data Dependency

- ADD r1, r2 ($r1 := r1 + r2$;))
- MOVE r3, r1 ($r3 := r1$;))
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

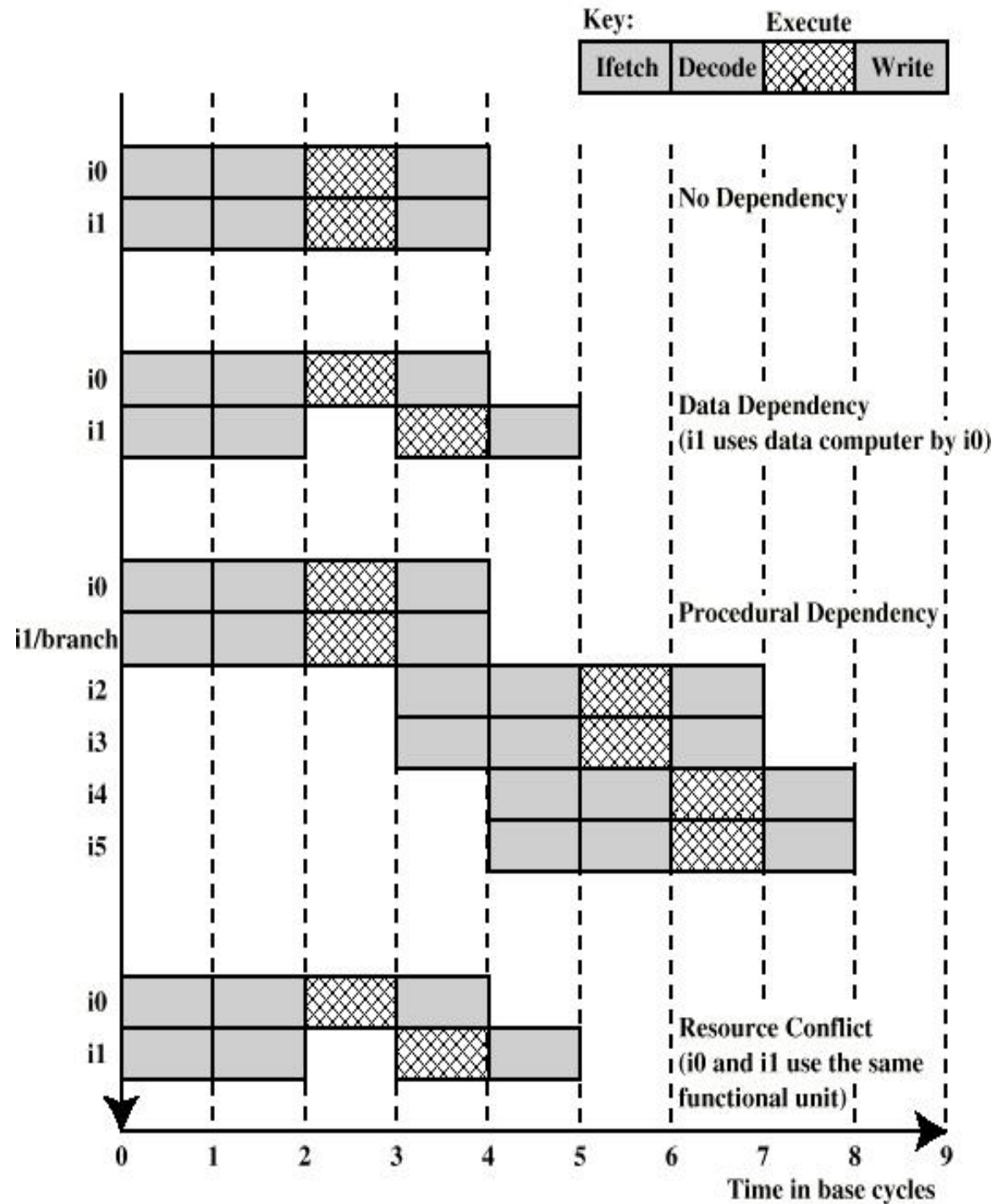
Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
 - e.g. two arithmetic instructions
- Can duplicate resources
 - e.g. have two arithmetic units

Effect of Dependencies



In-Order Execution

- Issue instructions in the order they occur
- Not very efficient
- May fetch >1 instruction
- Instructions must stall if necessary

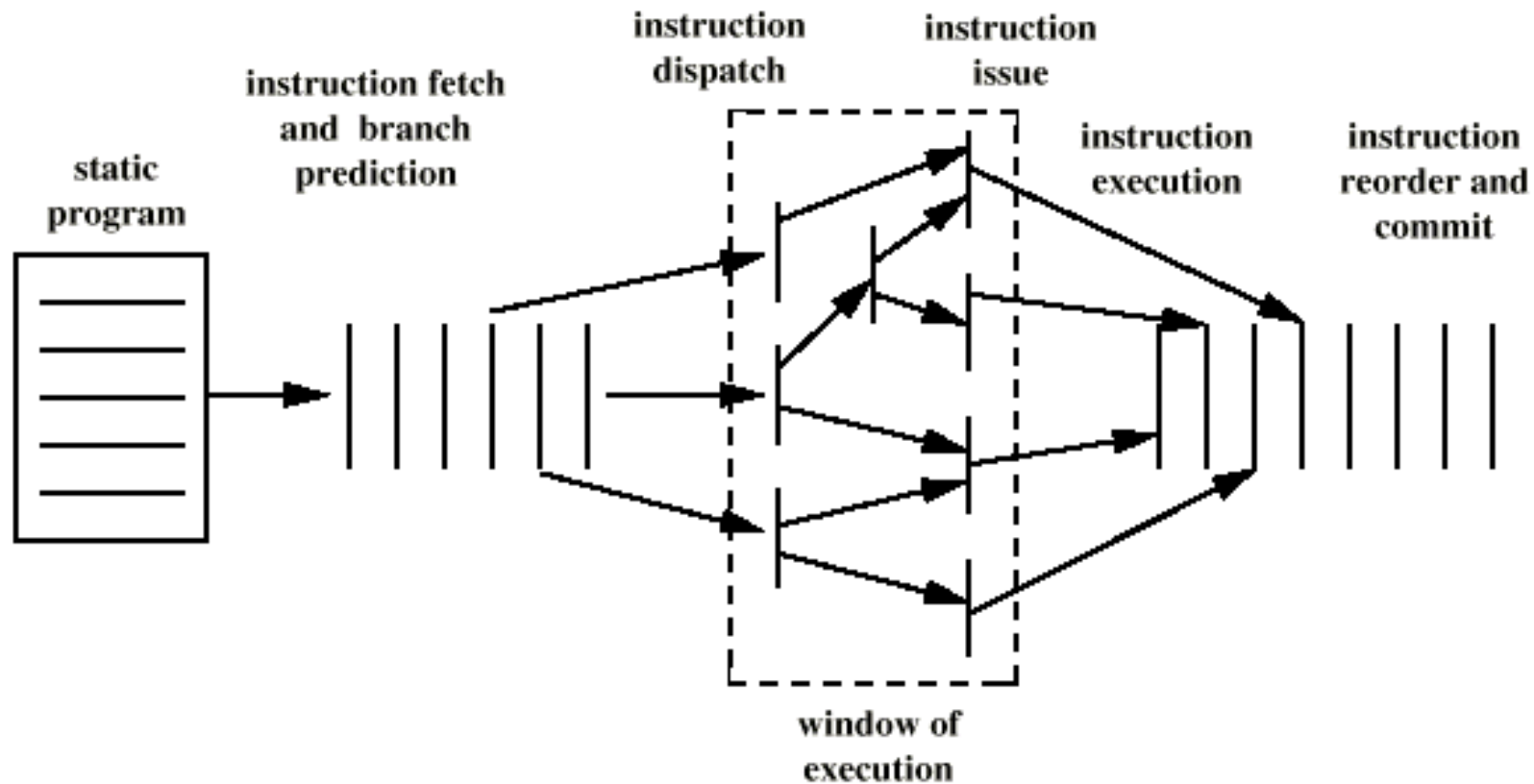
Out-of-Order Execution

- Output dependency
 - $R3 := R3 + R5;$ (I1)
 - $R4 := R3 + 1;$ (I2)
 - $R3 := R5 + 1;$ (I3)
 - I2 depends on result of I1 - data dependency
 - If I3 completes before I1, the result from I1 will be wrong - output (write-write) dependency

Out-of-Order Execution

- Decouple decode pipeline from execution pipeline
- Can continue to fetch and decode until this pipeline is full
- When a functional unit becomes available an instruction can be executed
- Since instructions have been decoded, processor can look ahead

Superscalar Execution



Superscalar Implementation

- Simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values
- Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

RISC - Delayed Branch

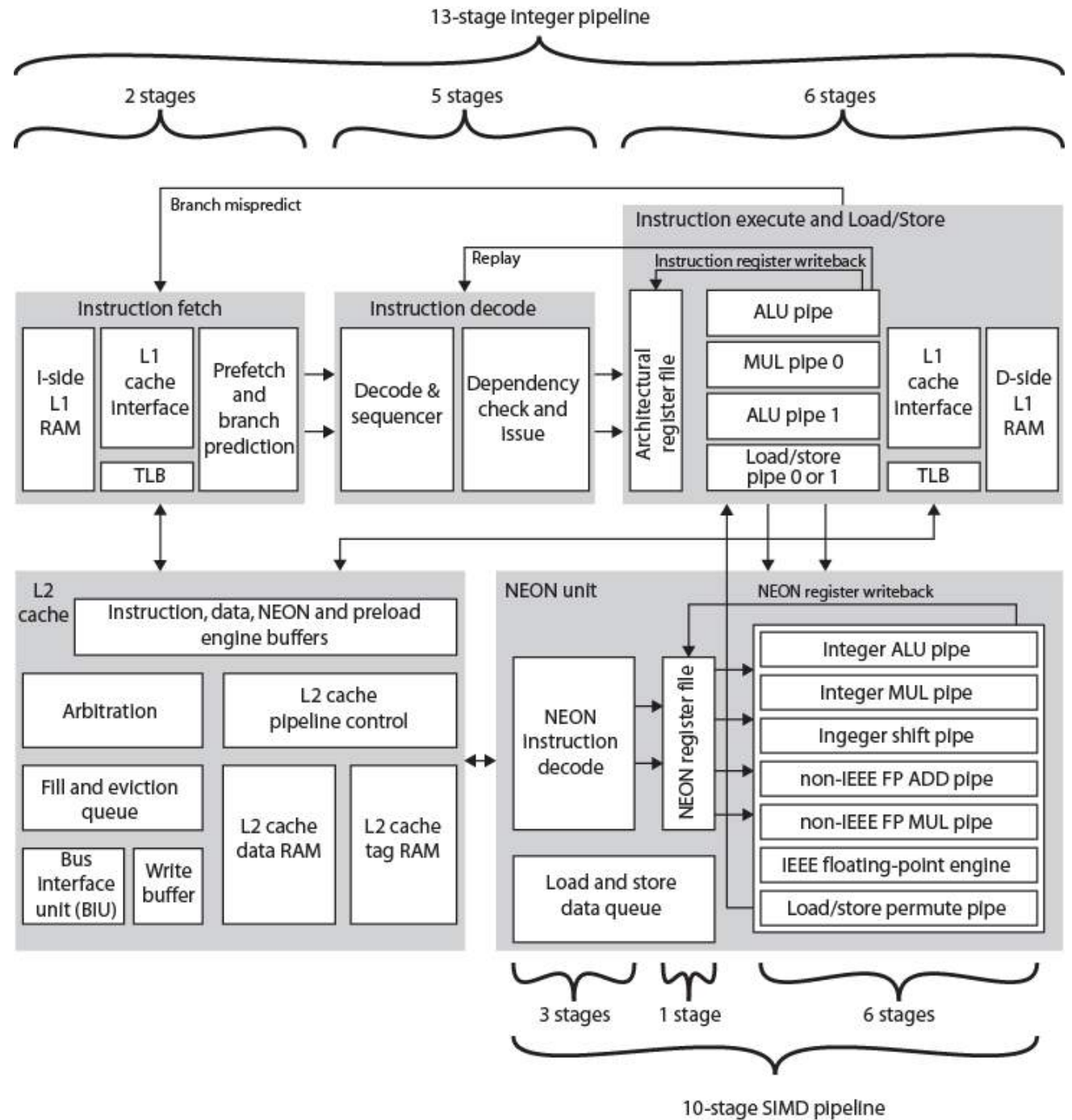
- Calculate result of branch before unusable instructions pre-fetched
- Always execute single instruction immediately following branch
- Keeps pipeline full while fetching new instruction stream
- Not as good for superscalar
 - Multiple instructions need to execute in delay slot
 - Instruction dependence problems
- Revert to branch prediction

Branch prediction

- Static branch prediction
 - the simplest branch prediction method
 - backward branches will be taken, and forward branches will not
 - reasonable prediction accuracy in loops
- Dynamic prediction
 - making use of history information about whether conditional branches were taken or not taken on previous execution
 - Branch Target Buffer (BTB) - 512 entries
 - Global History Buffer (GHB) - the strength and direction information about all of the recent branches.

ARM CORTEX-A8

- ARM refers to Cortex-A8 as application processors
- Embedded processor running complex operating system
 - Wireless, consumer and imaging applications
 - Mobile phones, set-top boxes, gaming consoles automotive navigation/entertainment systems
- Three functional units
- Dual, in-order-execution, 13-stage pipeline
 - Keep power required to a minimum
 - Out-of-order execution needs extra logic consuming extra power
- Separate SIMD (single-instruction-multiple-data) unit
 - 10-stage pipeline

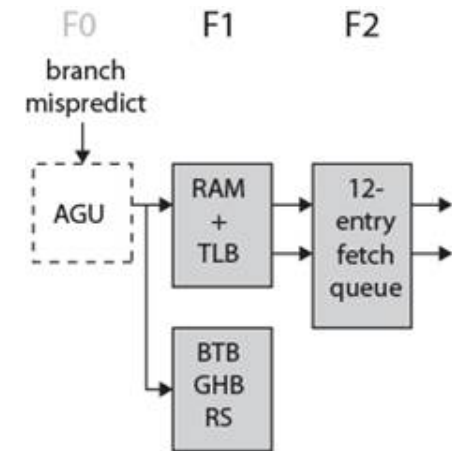


Instruction Fetch Unit

- Predicts instruction stream
- Fetches instructions from the L1 instruction cache
 - Up to four instructions per cycle
- Into buffer for decode pipeline
- Fetch unit includes L1 instruction cache
- Speculative instruction fetches
- Branch or exceptional instruction cause pipeline flush

Instruction Fetch Unit

- Stages:
- F0 address generation unit generates virtual address
 - Normally next sequentially
 - Can also be branch target address
- F1 Used to fetch instructions from L1 instruction cache
 - In parallel fetch address used to access branch prediction arrays
- F3 Instruction data are placed in instruction queue
 - If branch prediction, new target address sent to address generation unit
- Two-level global history branch predictor
 - Branch Target Buffer (BTB) and Global History Buffer (GHB)
- Return stack to predict subroutine return addresses
- Can fetch and queue up to 12 instructions
- Issues instructions two at a time



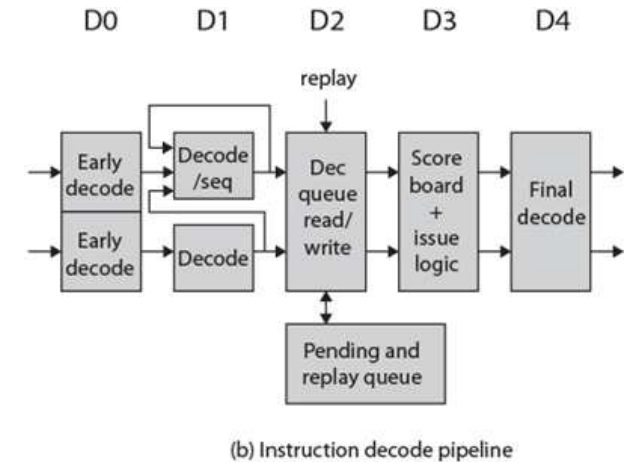
(a) Instruction fetch pipeline

Instruction Decode Unit

- Decodes and sequences all instructions
- Dual pipeline structure, *pipe0* and *pipe1*
 - Two instructions can progress at a time
 - Pipe0 contains older instruction in program order
 - If instruction in pipe0 cannot issue, pipe1 will not issue
- Instructions progress in order
- Results written back to register file at end of execution pipeline
 - Prevents WAR hazards
 - Keeps tracking of WAW hazards and recovery from flush conditions straightforward
 - Main concern of decode pipeline is prevention of RAW hazards

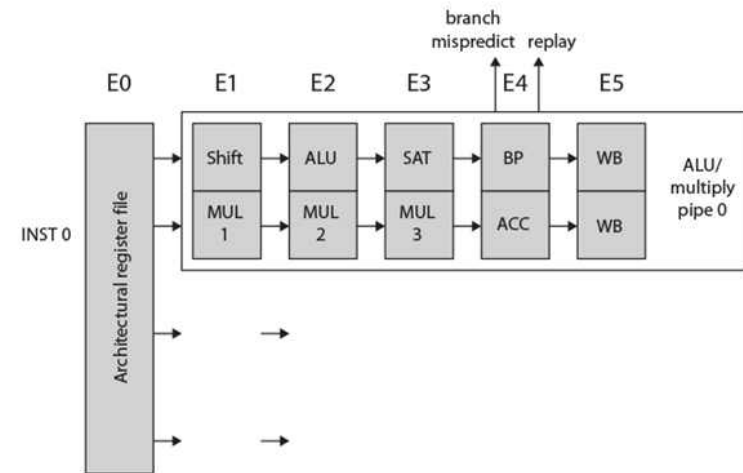
Instruction Processing Stages

- D0 Thumb instructions decompressed and preliminary decode is performed
- D1 Instruction decode is completed
- D2 Write instruction to and read instructions from pending/replay queue
- D3 Contains the instruction scheduling logic
 - Scoreboard predicts register availability using static scheduling
 - Hazard checking
- D4 Final decode for control signals for integer execute load/store units



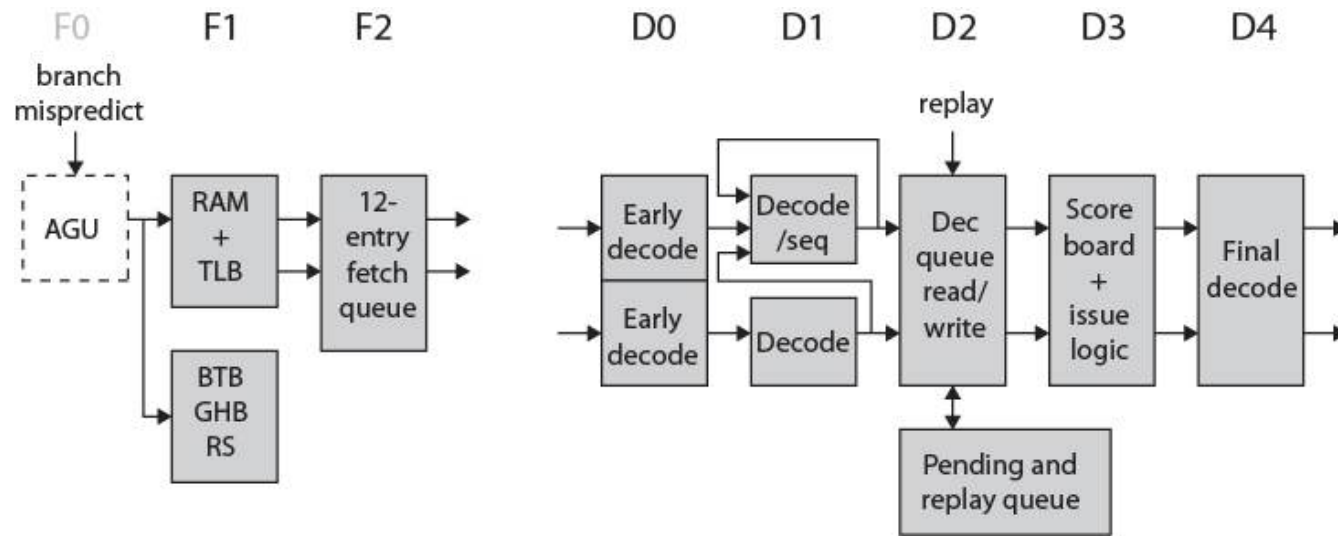
Integer Execution Unit

- Two symmetric (ALU) pipelines, an address generator for load and store instructions, and multiply pipeline
- Pipeline stages:
- E0 Access register file
 - Up to six registers for two instructions
- E1 Barrel shifter if needed.
- E2 ALU function
- E3 If needed, completes saturation arithmetic
- E4 Change in control flow prioritized and processed
- E5 Results written back to register file
- Multiply unit instructions routed to pipe0
 - Performed in stages E1 through E3
 - Multiply accumulate operation in E4



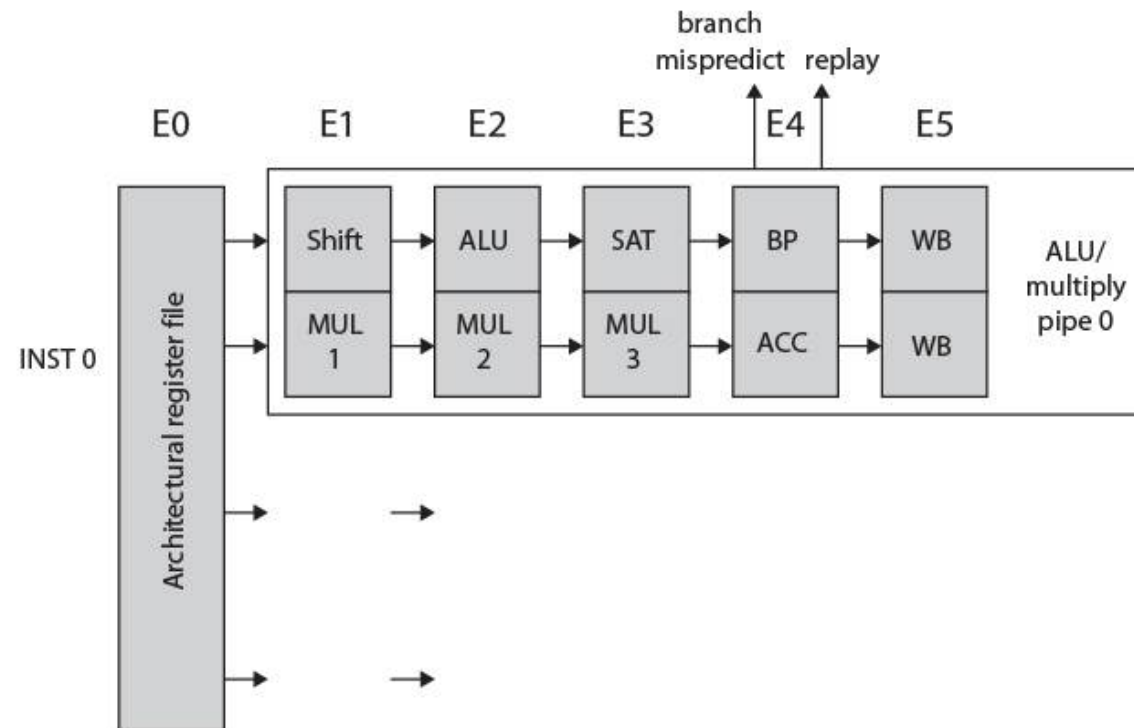
Load/store pipeline

- Parallel to integer pipeline
- E1 Memory address generated from base and index register
- E2 address applied to cache arrays
- E3 load, data returned and formatted
- E3 store, data are formatted and ready to be written to cache
- E4 Updates L2 cache, if required
- E5 Results are written to register file



(a) Instruction fetch pipeline

(b) Instruction decode pipeline



Instruction Sets

Instruction Sets

- ARM (32-bit instructions)
 - the original ARM instruction set
- Thumb
 - only 16-bit instructions
 - For building much smaller programs
- Thumb-2 technology
 - a mix of 16-bit and 32-bit instructions
 - performance similar to that of ARM, with code size similar to that of Thumb
 - Do not need to mode switching between ARM and Thumb
 - 16-bit versions being generated by default

Interworking between ARM state and Thumb state

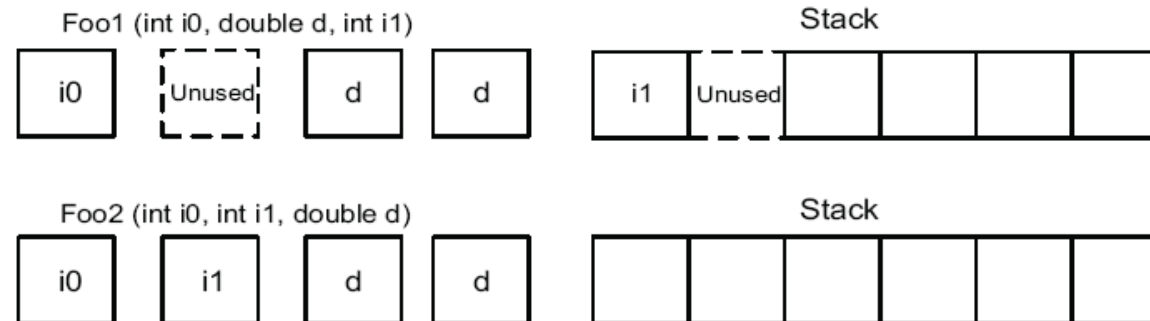
- ARM and Thumb code can be mixed, if the code conforms to the requirements of the ARM and Thumb Procedure Call Standards
- T bit is 1 == Thumb state / T bit is 0 == ARM
- when the T bit is modified, it is also necessary to flush the instruction pipeline
- To avoid problems with instructions being decoded in one state and then executed in another

Procedure Call Standard

- Argument registers
R0-R3 (a1-a4)
- Callee-saved registers
R4-R8, R10/R11
- Register which have
a dedicated role











R0	a1	argument 1/scratch register/result
R1	a2	argument 2/scratch register/result
R2	a3	argument 3/scratch register/result
R3	a4	argument 4/scratch register/result
R4	v1	register variable
R5	v2	register variable
R6	v3	register variable
R7	v4	register variable
R8	v5	register variable
R9	tr/sb/v6	static base/ register variable
R10	s1/v7	stack limit/stack chunk handle/register variable
R11	FP/v8	frame pointer/register variable
R12	IP	scratch register/new -sb in inter-link-unit calls
R13	SP	Lower end of the current stack frame
R14	LR	link register/scratch register
R15	PC	program counter






Efficient parameter passing




- Foo1
 - 8-byte argument (double d) must be passed in an even and consecutive odd register
 - 8-byte aligned the stack
- Foo2
 - More efficient than Foo1

Register Set – Thumb mode

THUMB State General Registers and Program Counter					
System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	 SP_fiq	 SP_svc	 SP_abt	 SP_und	 SP_fiq
LR	 LR_fiq	 LR_svc	 LR_abt	 LR_und	 LR_fiq
PC	PC	PC	PC	PC	PC

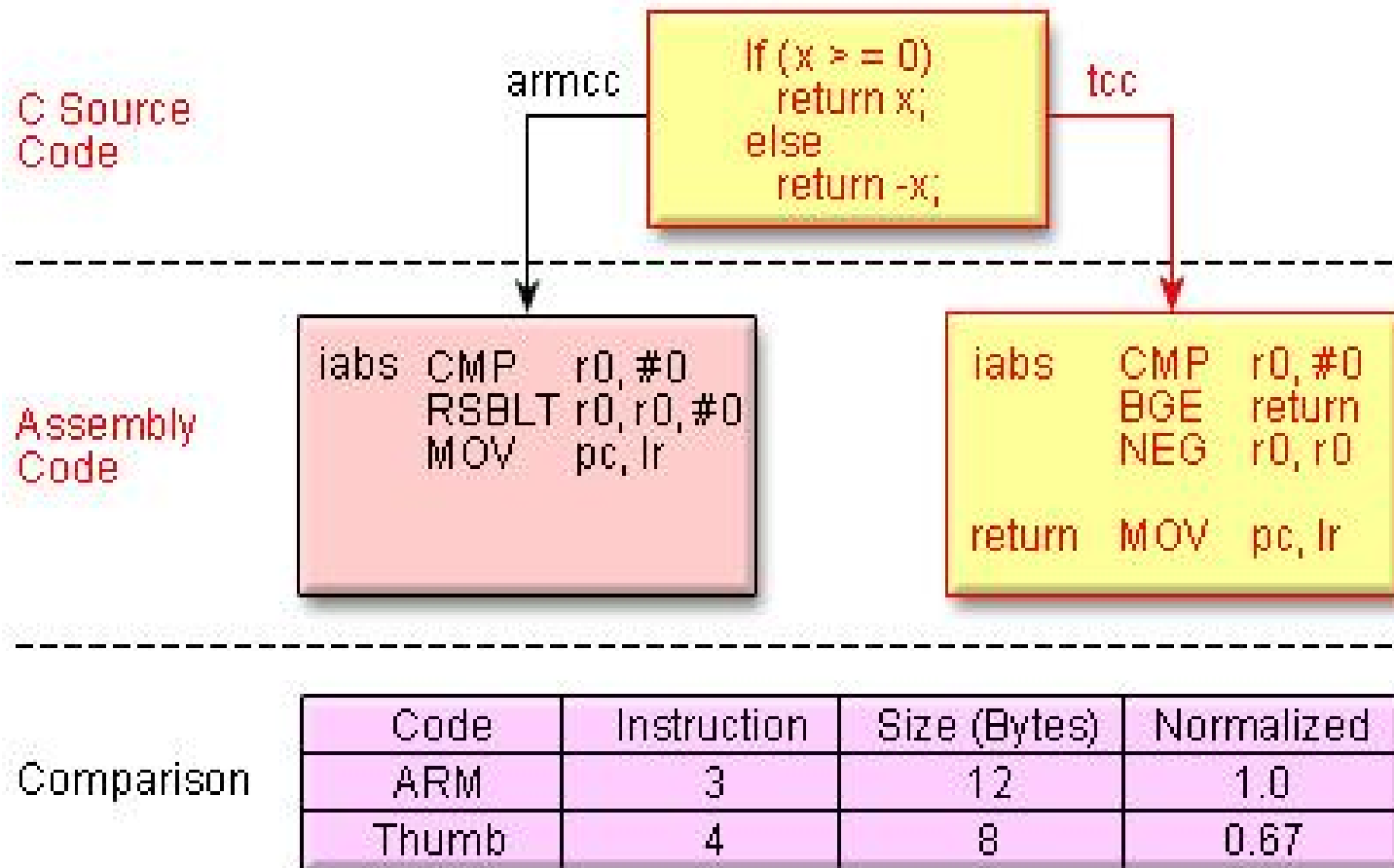
THUMB State Program Status Registers					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und

 = banked register

Thumb 16-bit Instructions

- 32-bit ARM instruction set의 압축형
 - 외부 메모리 사이에서 더 낮은 밴드폭을 가진다.
 - 코드 밀도를 높인다.
- A Thumb enabled ARM
 - 32-bit ARM과 16-bit Thumb instructions을 같이 사용
 - ARM and Thumb code 사이에서 상호 데이터 교환가능
 - branch with exchange (BX) instruction을 통해 상태 변화
- Instruction만이 16-bit이다.

Thumb Benefits



Thumb Benefits - *continued*

- Thumb programs typically are:
 - ~30% smaller than ARM programs
 - ~30% faster when accessing 16-bit memory
- Thumb reduces 32-bit system to 16-bit cost:
 - Consumes less power
 - Requires less external memory

Memory instructions

- LDR (Load Register)
 - LDR Rd, [Rn, Op2]
- STR (Store Register)
 - STR Rd, [Rn, Op2]
- Size
 - B for Byte / H for Halfword (16bits) /
D for doubleword (64 bits)

Addressing modes

- Register addressing: (1)
- Pre-indexed addressing: (2), (3)
- Pre-indexed with write-back: (4)
- Post-index with write-back: (5)

```
(1) LDR    R0, [R1] @ address pointed to by R1
(2) LDR    R0, [R1, R2] @ address pointed to by R1 + R2
(3) LDR    R0, [R1, R2, LSL #2] @ address is R1 + (R2*4)
(4) LDR    R0, [R1, #32]! @ address pointed to by R1 + 32, then R1:=R1 + 32
(5) LDR    R0, [R1], #32 @ read R0 from address pointed to by R1, then R1:=R1 + 32
```

Register addressing

- Ex) $r0 = 0x00000000$
 $r1 = 0x00009000$
 $mem32[0x00009000] = 0x01010101$
 $mem32[0x00009004] = 0x02020202$

LDR R0, [R1]

- R0 = 0x01010101
- R1 = 0x00009000

Pre-indexed addressing

- Ex)
 r0 = 0x00000000
 r1 = 0x00009000
 mem32[0x00009000] = 0x01010101
 mem32[0x00009004] = 0x02020202

LDR r0, [r1, #4]

- R0 = 0x02020202
- R1 = 0x00009000

Pre-indexed with write-back

- Ex)
 r0 = 0x00000000
 r1 = 0x00009000
 mem32[0x00009000] = 0x01010101
 mem32[0x00009004] = 0x02020202

LDR r0, [r1, #4]!

- R0 = 0x02020202
- R1 = 0x00009004

Post-index with write-back

- Ex)
 r0 = 0x00000000
 r1 = 0x00009000
 mem32[0x00009000] = 0x01010101
 mem32[0x00009004] = 0x02020202

LDR r0, [r1], #4

- R0 = 0x01010101
- R1 = 0x00009004

Multiple transfers

- LDM / STM

주소지정방식	설명	시작주소	끝주소	Rn!
IA	Increment after	R_n	$R_n + 4 * N - 4$	$R_n + 4 * N$
IB	Increment before	$R_n + 4$	$R_n + 4 * N$	$R_n + 4 * N$
DA	Decrement after	$R_n - 4 * N + 4$	R_n	$R_n - 4 * N$
DB	Decremtn before	$R_n - 4 * N$	$R_n - 4$	$R_n - 4 * N$

Multiple transfers

- LDMIA

- Ex)

```
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000
mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
```

```
LDMIA r0!, {r1 - r3}
```

- Results

```
r0 = 0x0008001c
r1 = 0x00000001
r2 = 0x00000002
r3 = 0x00000003
```

Multiple transfers

- LDMIA - start

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	R3 = 0x00000000
	0x80014	0x00000002	R2 = 0x00000000
R0 = 0x80010 →	0x80010	0x00000001	R1 = 0x00000000
	0x8000c	0x00000000	

Multiple transfers

- LDMIA - end

Address pointer	Memory address	Data	
	0x80020	0x00000005	
R0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	R3 = 0x00000003
	0x80014	0x00000002	R2 = 0x00000002
	0x80010	0x00000001	R1 = 0x00000001
	0x8000c	0x00000000	

Multiple transfers

- LDMIB

- Ex)

```
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000
mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
```

```
LDMIB. r0!, {r1 - r3}
```

- Results

```
r0 = 0x0008001c
r1 = 0x00000002
r2 = 0x00000003
r3 = 0x00000004
```

Multiple transfers

- LDMIB - start

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	R3 = 0x00000000
	0x80014	0x00000002	R2 = 0x00000000
R0 = 0x80010 →	0x80010	0x00000001	R1 = 0x00000000
	0x8000c	0x00000000	

Multiple transfers

- LDMIB - end

Address pointer	Memory address	Data	
	0x80020	0x00000005	
R0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	R3 = 0x00000004
	0x80014	0x00000002	R2 = 0x00000003
	0x80010	0x00000001	R1 = 0x00000002
	0x8000c	0x00000000	

Multiple transfers

- Matching

STM	LDM
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Multiple transfers

- LDM

- Syntax

`LDM{addr_mode}{cond} Rn{!},reglist{^}`

- ! if present, specifies that the final address is written back into Rn
 - ^ (in a mode other than User or System) means one of two possible special actions
 - data is transferred into the User mode registers instead of the current mode registers (without PC)
 - the normal multiple register transfer happens and the SPSR is copied into the CPSR (with PC)