Computer Organization and Architecture

Chapter 14

Instruction Level Parallelism and Superscalar Processors

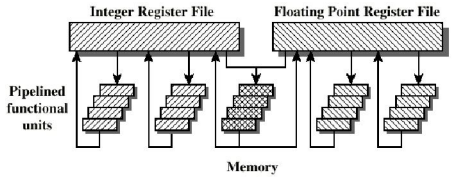
What does Superscalar mean?

- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently in separate pipelines
  - Instructions are not necessarily executed in the order in which they appear in a program
  - Processor attempts to find instructions that can be executed independently, even if they are out-of-order
  - Use additional registers and register renaming to eliminate some dependencies
- Equally applicable to RISC & CISC
- Quickly adopted and now standard approach for high-performance microprocessors

Why Superscalar?

- Term was coined in 1987; first such processors were roughly a year later
- Most machine operations are on scalar quantities (see RISC notes)
- So if we improve these operations we can get a significant overall improvement
- Two main ideas:
  - Execute instructions concurrently and independently in separate pipelines
  - Improve throughput of concurrent pipelines by allowing out-of-order execution

General Superscalar Organization



Speedup with superscalar architectures

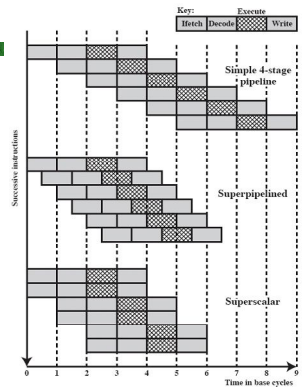
- Results vary considerably depending on hardware and applications being simulated

Reference	Speedup
[TJAD70]	1.8
[KUCK77]	8
[WEIS84]	1.58
[ACOS86]	2.7
[SOHI90]	1.8
[SMIT89]	2.3
[JOUP89b]	2.2
[LEE91]	7

Superpipelining

- An alternative approach to performance improvement
  - Many pipeline stages need less than half a clock cycle
  - So we can double the internal clock speed to get two tasks per external clock cycle (Example MIPS R4000)

Superscalar vs Superpipeline



Instruction Level Parallelism

- Instruction level parallelism is the degree on average by which the instruction of a program can be executed in parallel
- Achieved by:
  - Compiler based optimization
  - Hardware techniques
- Limited by:
  - True data dependency
  - Procedural dependency
  - Resource conflicts
  - Output dependency
  - Antidependency

True Data Dependency

```
ADD eax,ecx
MOV ebx, eax
```

- We can fetch and decode the second instruction in parallel with the first
- But we cannot execute the second instruction until the first is finished
  - The second instruction has to read the results of the first instruction
- Also called "flow dependency" or "read-after-write" dependency
- A fairly obvious general rule is that any instruction has to be delayed until its inputs are available

A note on terminology

- Everybody agrees that this instruction sequence has a data hazard, and that it is a "true data dependency"

```
ADD eax, ecx
MOV ebx, eax
```

- Unfortunately, in the literature some people describe this as "read after write" (RAW) while others describe it as "write after read" (WAR)
  - The RAW description describes the instruction sequence as it appears in the instruction stream and as it should be correctly executed by the processor. The Read MUST take place after the Write
  - The WAR description describes the hazard, i.e., it describes the incorrect execution sequence where the Write actually occurs after the read, so the result is not correct
- The textbook uses RAW in Ch. 12 and WAR in Ch. 14.
- We will use the RAW approach (describe the instruction stream as it should be executed)

True Data Dependency

```
ADD r1,r2    ;(r1 <- r1+r2;)
MOV r3,r1    ;(r3 <- r1;)
```

- Note that these instructions may not cause a delay in a simple pipeline
- But consider this sequence

```
MOV ebx, memvar
MOV eax, ebx
```

- Typical RISC processor requires 2 or more cycles to read from memory
- Can be hundredsof cycles if cache miss
- Pipeline is stalled until load completes

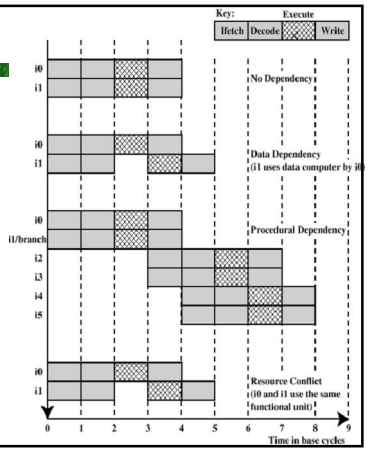
Procedural Dependency

- We cannot execute instructions after a branch in parallel with instructions before a branch
- Also, if the instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches
  - This suggests that superscalar techniques are more easily applied to RISC machines
  - CISC resolution is to divide the fetch/decode stages into very small stages and use a prefetch queue

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
  - Memory
  - Cache
  - Register-file ports
  - Functional units (adder, shifter)
- A resource conflict has a similar effect to true data dependency
- But we can duplicate resources to reduce some contention whereas we can never eliminate a data dependency
- Slow operations can be divided into smaller pipeline stages to minimize delays

Effect of Dependencies



Design Issues

- Instruction level parallelism
  - Occurs when instructions in a sequence are independent and execution can be overlapped
  - Governed by data and procedural dependency

Parallel execution	Sequential Execution
`load r1,r2`	`add r3,r3,1`
`add r3,r3,1`	`add r4,r3,r2`
`add r4,r4,r2`	`store [r4], r0`

- Machine Parallelism
  - A measure of the ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines and speed and sophistication of measures taken to find independent instructions

Factors

- Instruction Level Parallelism
  - Instruction set architecture
  - Application program
  - Operation latency
- Machine Parallelism
  - Number of parallel pipelines
  - Ability to find independent instructions

Instruction Issue

- Instruction issue is the process of initiating instruction execution
  - Occurs when decoded instruction moved to first execute phase
- *Instruction issue policy* is the protocol used to issue instructions
- Processor looks ahead to locate instructions that can be executed
- 3 types of orderings are important:
  - Order in which instructions are fetched
  - Order in which instructions are executed
  - Order in which instructions change registers and memory

Instruction Issue Policy

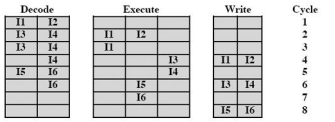
- To optimize pipeline execution processor will need to alter one or more orderings
- But the result of the computation must be correct
- Three general policy categories
  - In-order issue with in-order completion
  - In-order issue with out-of-order completion
  - Out-of-order issue with out-of-order completion

Example Machine

- Examples assume the following:
  - CPU has three functional units: two integer ALUs and one floating point ALU
  - The CPU can fetch and decode two instructions at a time
  - There are two instances of the write-back pipeline stage

In-Order Issue In-Order Completion

- Issue instructions in the order they occur
  - Not very efficient
  - Instructions must stall if necessary (conflict for functional unit or dependency)



(a) In-order issue and in-order completion

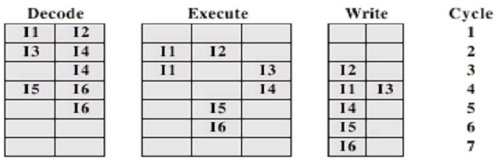
Assumptions and Constraints

I1 executes in 2 cycles I3,I4 conflict for func. unit

I5 depends on I4 result I5,I6 conflict for func.unit

In-Order Issue Out-of-Order Completion

- With out-of-order completion any number of instructions can be in the pipeline – to the limit of machine parallelism
- Here I2 runs to completion before I1
- As a result I3 completes earlier



Output (write-after-write) dependency

- Consider

  R3 <- R3 + R5 (I1)
  R4 <- R3 + 1 (I2)
  R3 <- R5 + 1 (I3)
  R7 <- R3 + R4 (I4)

  - I2 depends on result of I1 - data dependency
  - I4 depends on result of I3 – data dependency
  - What about I3 and I1?
  - If I3 completes before I1, the result from I1 will be wrong for I4
- The problem here that both I1 and I3 write to R3. In general writes must be completed in-order

Other complications

- With out-of-order completion processor interrupt and exception handling is more complex
- Instructions ahead of the current instruction may have already completed
- Note however that WAW dependencies can sometimes be resolved by simply using a different register

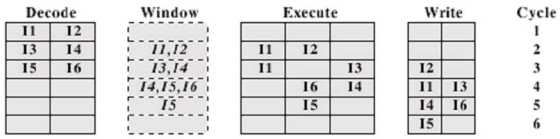
Out-of-Order Issue Out-of-Order Completion

- With in-order issue the processor stops decoding instructions when a dependency or conflict is detected
  - Cannot look ahead of the point of conflict to find other instructions to execute
- To allow out-of-order issue decouple the decode pipeline from execution pipeline
  - Can continue to fetch, decode and place instructions in a buffer (the instruction window) 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

Instruction Window

- Buffer that holds decoded instructions is called the instruction window
- When a functional unit becomes available an instruction is issued to the execute unit
- Any instruction can be issued provided
  - It can execute in the available functional unit
  - No conflicts or dependencies block the instruction

Out-of-Order Issue Out-of-Order Completion (Diagram)



Assumptions and Constraints

I1 executes in 2 cycles I3,I4 conflict for func. unit

I5 depends on I4 result I5,I6 conflict for func.unit

Note that because I5 depends on I4, but I6 does not we can issue I6 before I5

Antidependency (write-after-read dependency)

- Same example as write-after-write:

  R3 <- R3 + R5 (I1)
  R4 <- R3 + 1 (I2)
  R3 <- R5 + 1 (I3)
  R7 <- R3 + R4 (I4)

  - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3
  - "Antidependency" used because constraint is similar to true data dependency but reversed: I3 destroys a value that I2 uses
  - To find antidependencies look for register overwriting instructions and examine prior instructions
  - Note that we also have
    - an output dependency with respect to I1 and I3
    - True data dependencies with in I2/I1 and I4/I3

Register contention

- True data dependencies and resource conflicts are attributable to the flow of data through a program and the sequence of execution
- Output dependencies and antidependencies are attributable to contention for registers and arise because the values in registers may not reflect the sequence of values dictated by program flow
- Compiler register optimization can increase contention by maximizing registers usage

Register Renaming

- One easy way to resolve resource contention is to increase or duplicate resources
- Allocate registers dynamically
  - i.e. actual registers are not specifically named
  - New register values are created when an instruction references a register as a dest operand
  - Subsequent instructions that access that value as a source operand are revised to refer to the register actually containing the value

Register Renaming example

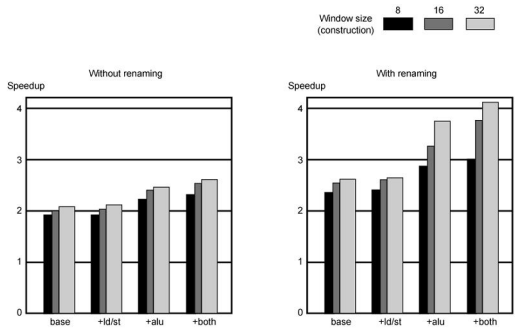
R3b <- R3a + R5a (I1)
R4b <- R3b + 1 (I2)
R3c <- R5a + 1 (I3)
R7b <- R3c + R4b (I4)

- Register reference without subscript refers to a logical register in the instruction
- With subscript it is a hardware register actually allocated
- Note R3a R3b R3c
- I3 can be issued immediately when EU is available

Machine Level Parallelism

- Three techniques for enhancing superscalar performance:
  - Duplication of Resources
  - Out of order issue
  - Register Renaming
- It is probably not worth duplicating functional units without register renaming
- In order to effectively utilize duplicated resources we need an instruction window that is large enough for effective lookahead (more than 8)

Speedups of Machine Organizations Without Procedural Dependencies



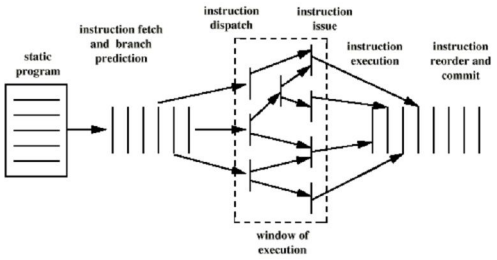
Branch Prediction

- The 80486 fetches both next sequential instruction after branch and branch target instruction
- Because there are two pipeline stages between fetch and execute we still have a two cycle delay when the branch is taken

RISC - Delayed Branch

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

Superscalar Execution



Instruction commit

- Some instructions may have been executed out of order
- Others may need to be discarded
- Do not update program-visible registers and permanent storage immediately
- Retain in temp storage until the sequential model would have executed them

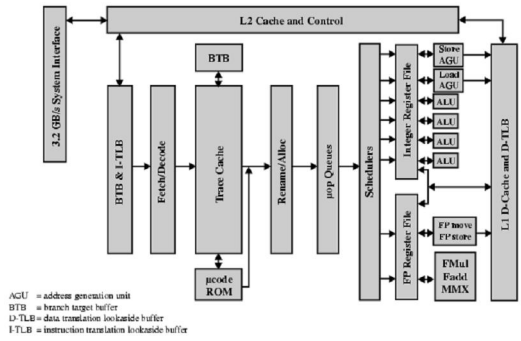
Superscalar Implementation Summary

- 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

Pentium 4

- The 80486 was a straightforward CISC machine
- Pentium added some superscalar components
  - Two separate integer execution units
- Pentium Pro was a full blown superscalar implementation
- Subsequent models refined & enhanced the superscalar design

Pentium 4 Block Diagram

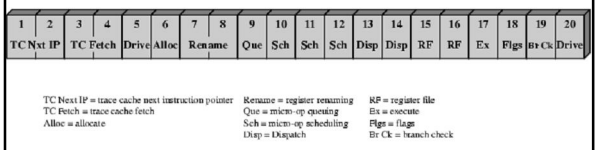


Pentium 4 Operation

1. Processor fetches instructions from memory in static program order.
2. Each instruction is translated into one or more fixed length RISC instructions (micro-operations)
3. Execute micro-ops on superscalar pipeline
   - micro-ops may be executed out of order
4. Processor commits results of micro-ops to register set in original program flow order

- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
  - Some micro-ops require multiple execution stages
    - Longer pipeline (up to 20 stages)
  - c.f. five stage pipeline on x86 up to Pentium

Pentium 4 Pipeline



Pentium 4 Front end

- The in-order front end is part of the machine that is outside the true pipeline
- It feeds into L1 instruction cache called trace cache (start of pipeline)
  - Processor operates from trace cache; cache miss causes L2 cache lookup and front end feed to trace cache
  - Fetch-decode unit uses branch target buffer (BTB) and instruction TLB to determine cache line in L2 cache
  - The I-TLB translates linear addresses into physical addresses that are presented to L2
- 64 bytes are fetched at a time; default sequential but can be altered via branch prediction and the BTB

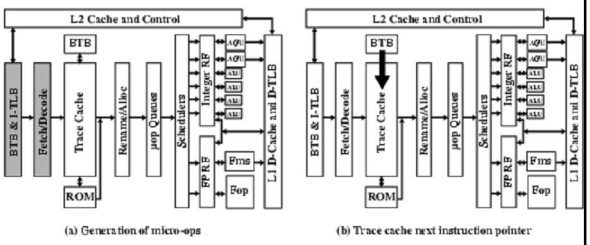
Generation of micro-ops

- The Fetch/decode unit scans instructions to determine instruction boundaries – some are as long as 18 bytes
- Each Pentium instruction is translated to 1-4 micro-ops (118 bit fixed length)
  - C.f. RISC 32-bit instruction
- Generated micro-ops are stored in trace cache

Trace Cache next instruction pointer

- First two pipeline stages deal with selection of instructions from trace cache
- This is a separate branch prediction mechanism from fetch/decode unit
- BTB used as described in Ch 13
- BTB is 4-way set associative cache with 512 lines
  - Address of branch is tag
  - BTB includes last dest address and 4-bits of history
- BTB is a two-stage branch prediction mechanism with 4 bits of history data because pipeline penalties are large
- Static prediction for new branches
  - Not IP-relative (RET) predict taken
  - IP-relative backward branches: predict taken
  - IP-relative forward branches: predict not taken

Pentium 4 Pipeline Operation (1)



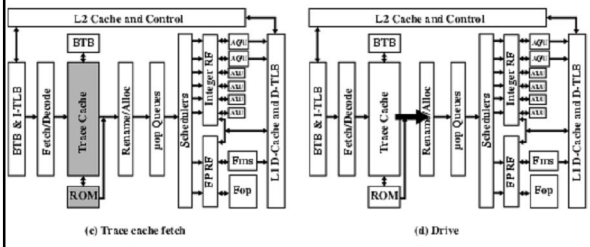
Trace Cache Fetch

- Trace caches takes micro-ops and assembles them into program-ordered sequences called traces
- Micro-ops are fetched sequentially subject to branch prediction logic
- A few instruction require more than 4 micro-ops
  - These are transferred to microcode ROM for sequencing
  - Microcode ROM is a programmed control unit that contains 5+ micro-op sequences for complex machine instructions

Drive

- The fifth stage delivers decoded sequences from trace cache / microcode ROM to the register renaming and allocation module

Pentium 4 Pipeline Operation (2)



Out of order execution logic

- Reorders micro-ops for fast execution
- Allocate stage allocates resources for three micro-ops per clock cycle:
  - If a needed resource (e.g., register) is not available stall the pipeline
  - Allocate Reorder Buffer (ROB) entry which tracks completion status of micro-ops (up to 126 can be in-process at any time)
  - Allocate one of 128 integer or FP registers entries for result data value OR a load or store buffer (pipeline can handle 48 loads and 24 stores)
  - Allocate an entry in one of the two micro-op queues for instruction schedulers

Reorder Buffer (ROB)

- ROB is a circular buffer that handles up 126 micro-ops and contains the 128 hardware registers
- Each buffer entry contains:
  - Status: scheduled, dispatched or completed
  - Memory addr: instruction address
  - Micro-op
  - Alias register: redirects reference of one of 16 visible registers to one of 128 hardware registers
- Instructions enter ROB in order but are dispatched out-of-order
- Criteria for dispatch: EU and all necessary data items are available
- Instructions are retired from the ROB in-order

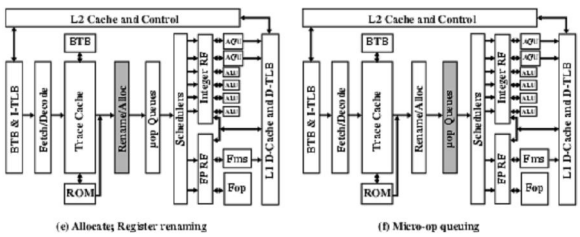
Register renaming

- Rename stage maps 16 architectural registers (8 gen purpose + 8 FP) into 128 hardware registers
- This stage removes output dependencies and antidependencies and preserves read-after-write dependencies (true data dependencies)

Micro-op queueing

- Two micro-op queues hold micro-ops until room in scheduler is available
  - One queue for memory ops (loads and stores)
  - All other instructions in second queue
- Queues are FIFO but independent – no order is maintained between queues

Pentium 4 Pipeline Operation (3)

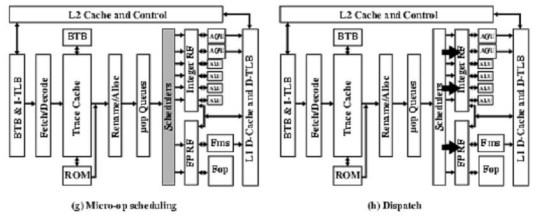


(e) Allocate, Register renaming (f) Micro-op queuing

Scheduling and Dispatching

- Schedulers dispatch micro-ops for execution
  - Look for micro-ops with all operands available (check status indicators)
  - Dispatch to appropriate EU when available
  - Up to six micro-ops can be dispatched per cycle
  - When more than 6 are available FIFO order is used
- Four ports attach schedulers to EUs
  - Port 0: complex integer operations and floating point
  - Port 1: simple integer ops and branch mispredictions
  - Ports 2,3: loads and stores

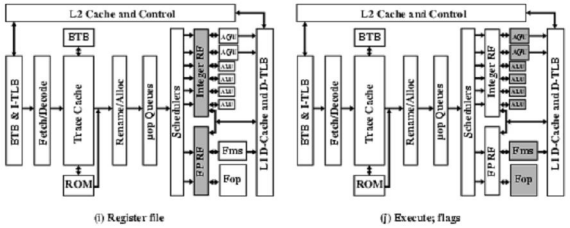
Pentium 4 Pipeline Operation (4)



Execution Units

- Integer and FP register files, L1 data cache are the source for execution units
- Separate pipeline stage computes flags

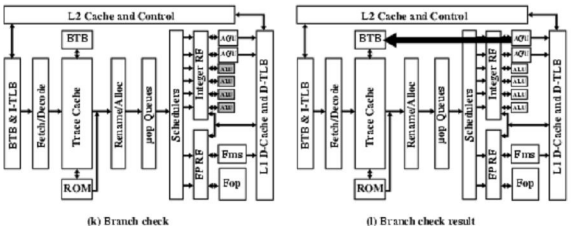
Pentium 4 Pipeline Operation (5)



Branch Check

- Branch checking compares actual branch result to prediction
- Mispredicted branches require pipeline cleanup
- Proper branch dest is provided to branch predictor during a drive stage
- Restart pipeline from new target address

Pentium 4 Pipeline Operation (6)

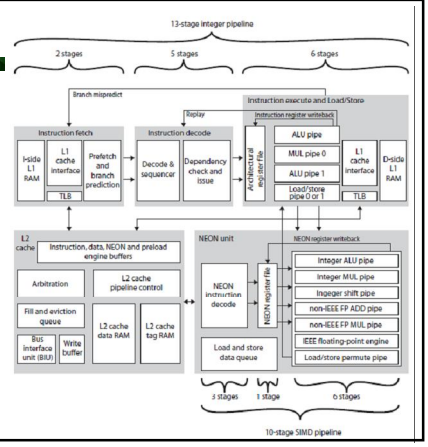


ARM Cortext A-8

- Recent ARM implementations have introduced superscalar techniques
- Cortex A-8 is in the ARM family that ARM refers to as “application processors”
  - An embedded processor running a complex operating system for wireless, consumer and imaging applications
    - Mobile phones
    - Set-top boxes
    - Gaming consoles
    - GPS navigation
    - Automobile entertainment systems

Cortex A-8

Logical View



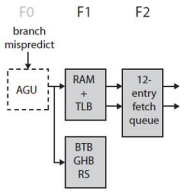
Key Points

- Dual in-order issue 13-stage pipeline
  - In-order issue was selected to minimize power consumption
  - Out-of-order issue can require huge amounts of circuitry (and hence power)
- SIMD Unit uses a separate 10-stage pipeline

Instruction Fetch Unit

- Predicts instruction stream
- Fetches instructions from L1 cache and places them in a buffer for the decode pipeline
- Fetch unit includes the L1 instruction cache
- There may be several unresolved branches in the pipeline, so fetches are speculative
- Pipeline Stages:
  - F0 Address Generation Unit (AGU) generates a new virtual address (sequential or branch target)
  - F1 calculated address used to fetch instructions from L1 cache. In parallel the fetch address is used to access branch prediction arrays
  - F2 Instructions are placed in queue. If instruction results in branch prediction, new target address is sent to AGU

Instruction Fetch Pipeline



(a) Instruction fetch pipeline

- Up to 12 instructions can be fetched and queued
- Instructions are issued to decode buffer two at a time
- Queue enables prefetch ahead of the integer pipeline

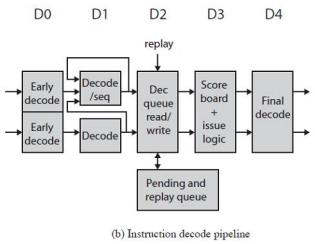
BHB and GHB

- Processor implements a 2-level global branch predictor: Branch Target Buffer (BTB) with Global History Buffer (GHB)
  - Accessed in parallel with instruction fetches
  - 512 entry BTB indicates if current fetch address contains a branch and stores its branch address
  - With BTB hit the branch is predicted and the GHB is accessed
    - Contains 4096 2 bit counters that encode direction of branch
    - Indexed by 10-bit history of the direction of the last 10 branches encountered and 4 bits of the PC
  - In addition to dynamic branch predictor a return stack predicts subroutine return addresses
    - Return stack has 8 32-bit entries that store link register and ARM/Thumb state of caller
    - When a return is predicted taken the return stack provides the last pushed address and state

Instruction Decode Unit

- Has a dual pipeline (pipe0 and pipe1) that processes two instructions at a time
  - Pipe0 always contains the older instruction in program order so pipe0 must always issue before pipe1
  - Issued instructions proceed in-order through execution pipeline
  - In-order issue means no WAR hazards; WAW is easy to handle so main concern is prevention of RAW hazards

Decode Unit



(b) Instruction decode pipeline

Decode Stages D0 and D1

- D0 Thumb instructions are decompressed into 32-bit ARM instructions; then preliminary decode
- D1 completes decoding
  - In first two stages the instruction type, source and dest operands, resource requirements are determined
  - A few complex instructions are broken into smaller instructions that are sequenced through execution

Decode Stage D2

- D2 writes instructions into /reads from the pending/replay queue
  - Replay queue deals with the effect of memory on instruction timing. Instructions are scheduled into D3 based on prediction of when source operand will be available
  - Any stall from memory results in minimum 8-cycle delay, can be much longer
  - To deal with stalls a recovery mechanism is used to flush all subsequent instructions in execution queue and reissue or "replay" them
    - instructions are copied into replay queue before issued
    - Removed as they write back their results and retire
    - With a Replay signal instructions are retrieved from queue and they re-enter the pipeline

Decode Stages D3-D4

- D3 Instruction scheduling logic. A scoreboard (see appendix I.4) predicts register availability. Hazard checking is performed
- F4 Final decode for control signals required by the integer execute and the load/store units

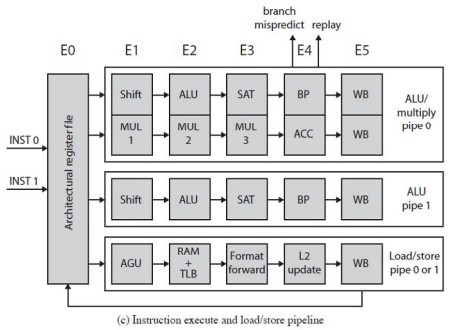
Memory System Effects on Instruction Timing

Replay event	Delay	Description
Load data miss	8 cycles	**1.** A load instruction misses in the L1 data cache. **2.** A request is then made to the L2 data cache. **3.** If a miss also occurs in the L2 data cache, then a second replay occurs. The number of stall cycles depends on the external system memory timing. The minimum time required to receive the critical word for an L2 cache miss is approximately 25 cycles, but can be much longer because of L3 memory latencies.
Data TLB miss	24 cycles	**1.** A table walk because of a miss in the L1 TLB causes a 24-cycle delay, assuming the translation table entries are found in the L2 cache. **2.** If the translation table entries are not present in the L2 cache, the number of stall cycles depends on the external system memory timing.
Store buffer full	8 cycles plus latency to drain fill buffer	**1.** A store instruction miss does not result in any stalls unless the store buffer is full. **2.** In the case of a full store buffer, the delay is at least eight cycles. The delay can be more if it takes longer to drain some entries from the store buffer.
Unaligned load or store request	8 cycles	**1.** If a load instruction address is unaligned and the full access is not contained within a 128-bit boundary, there is a 8-cycle penalty. **2.** If a store instruction address is unaligned and the full access is not contained within a 64-bit boundary, there is a 8-cycle penalty.

Integer Execution Unit

- Consists of
  - two symmetric ALU pipelines
  - Address generator for load and store instructions
  - Multiply pipeline
  - Execution also performs register write-back
- Operations
  - Executes all integer ALU and multiply operations including flag generation
  - Generals virtual addresses for loads and stores and the base write-back value when required
  - Supplies data for stores and forwards data and flags
  - Processes branches and returns and evaluates instruction condition codes

Integer Execution Unit



(c) Instruction execute and load/store pipeline

Integer Instructions

- Can use pipe0 or pipe1. Stages are
  - E0 Access register file; reads up to six registers for two instructions
  - E1 Barrel shifter performs function if needed
  - E2 ALU unit performs operation
  - E3 if needed saturation unit performs function
  - E4 handles changes in control flow, including branch misprediction, exceptions and memory system replays
  - E5 write back results to register file
- If multiply unit is used, instruction can only execute in pipe0

Load/Store Pipeline

- Stages are
  - E0 Access register file; reads up to six registers for two instructions
  - E1 AGU generates memory address from base and index registers
  - E2 address applied to the cache
  - E3 for LOADs data are returned for forwarding to ALU or MUL unit. For store data are prepared for writeback to cache
  - E4 updates L2 cache if needed
  - E5 write back results to register file

Dual Issue Restrictions

Restriction type	Description	Example	Cycle	Restriction

Load/store resource hazard	There is only one LS pipeline. Only one LS instruction can be issued per cycle. It can be in pipeline 0 or pipeline 1	LDR r5, [r6]
		STR r7, [r8]
		MOV r9, r10
			1
			2
			2
				
				Wait for LS unit
				Dual issue possible

Multiply resource hazard	There is only one multiply pipeline, and it is only available in pipeline 0.	ADD r1, r2, r3
		MUL r4, r5, r6
		MUL r7, r8, r9
			1
			2
			3
				
				Wait for pipeline 0
				Wait for multiply unit

Branch resource hazard	There can be only one branch per cycle. It can be in pipeline 0 or pipeline 1. A branch is any instruction that changes the PC.	BX r1 BEQ 0x1000 ADD r1, r2, r3	1 2 2	 Wait for branch Dual issue possible

Data output hazard	Instructions with the same destination cannot be issued in the same cycle. This can happen with conditional code.	MOVEQ r1, r2
		MOVNE r1, r3
		
		LDR r5, [r6]
			1
			2
			
			2
				
				Wait because of output dependency
				
				Dual issue possible

Data source hazard	Instructions cannot be issued if their data is not available. See the scheduling tables for source requirements and stages results.	ADD r1, r2, r3
		ADD r4, r1, r6
		LDR r7, [r4]
			1
			2
			4
				
				Wait for r1
				Wait two cycles for r4

Multi-cycle instructions	Multi-cycle instructions must issue in pipeline 0 and can only dual issue in their last iteration.	MOV r1, r2
		LDM r3, {r4-r7}
		LDM (cycle 2)
		LDM (cycle 3)
		
		ADD r8, r9, r10
			1
			2
			3
			4
			
			4
				Wait for pipeline 0, transfer r4
				Transfer r5, r6
				Transfer r7
				Dual issue possible on last transfer

Example Sequence with Scheduling-1

Cycle	Program Counter	Instruction	Timing Description
1	0x00000ed0	BX r14	Dual issue pipeline 0
1	0x00000ee4	CMP r0,#0	Dual issue in pipeline 1
2	0x00000ee8	MOV r3,#3	Dual issue pipeline 0
2	0x00000eec	MOV r0,#0	Dual issue in pipeline 1
3	0x00000ef0	STREQ r3,[r1,#0]	Dual issue in pipeline 0, r3 not needed until E3
3	0x00000ef4	CMP r2,#4	Dual issue in pipeline 1
4	0x00000ef8	LDRLS pc,[pc,r2,LSL #2]	Single issue pipeline 0, +1 cycle for load to pc, no extra cycle for shift since LSL #2
5	0x00000f2c	MOV r0,#1	Dual issue with 2nd iteration of load in pipeline 1
6	0x00000f30	B {pc}+8	#0xf38 dual issue pipeline 0
7	0x00000f38	STR r0,[r1,#0]	Dual issue pipeline 1
7	0x00000f3c:	LDR pc,[r13],#4	Single issue pipeline 0, +1 cycle for load to pc
8	0x0000017c	ADD r2,r4,#0xc	Dual issue with 2nd iteration of load in pipeline 1
9	0x00000180	LDR r0,[r6,#4]	Dual issue pipeline 0
9	0x00000184	MOV r1,#0xa	Dual issue pipeline 1

Example Sequence with Scheduling-2

Cycle	Program Counter	Instruction	Timing Description
12	0x00000188	LDR r0,[r0,#0]	Single issue pipeline 0: r0 produced in E3, required in E1, so +2 cycle stall
13	0x0000018c	STR r0,[r4,#0]	Single issue pipeline 0 due to LS resource hazard, no extra delay for r0 since produced in E3 and consumed in E3
14	0x00000190	LDR r0,[r4,#0xc]	Single issue pipeline 0 due to LS resource hazard
15	0x00000194	LDMFD r13!,{r4-r6,r14}	Load multiple loads r4 in 1st cycle, r5 and r6 in 2nd cycle, r14 in 3rd cycle, 3 cycles total
17	0x00000198	B {pc}+0xda8	#0xf40 dual issue in pipeline 1 with 3rd cycle of LDM
18	0x00000f40	ADD r0,r0,#2 ARM	Single issue in pipeline 0
19	0x00000f44	ADD r0,r1,r0 ARM	Single issue in pipeline 0, no dual issue due to hazard on r0 produced in E2 and required in E2

SIMD and Floating Point Pipeline

- SIMD and floating point operations are decoded in the integer pipeline, and then processed in a separate 10-stage pipeline called the NEON unit
- See

http://www.arm.com/products/processors/technologies/neon.php

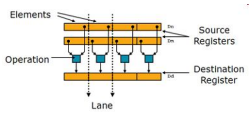
NEON technology can accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis by at least 3x the performance of ARMv5 and at least 2x the performance of ARMv6 SIMD.

NEON technology is cleanly architected and works seamlessly with its own independent pipeline and register file.

NEON technology is a 128 bit SIMD (Single Instruction, Multiple Data) architecture extension for the ARM Cortex™-A series processors, designed to provide flexible and powerful acceleration for consumer multimedia applications, delivering a significantly enhanced user experience. It has 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide.

NEON cont'd

- NEON instructions perform "Packed SIMD" processing:
  - Registers are considered as vectors of elements of the same data type
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single precision floating point
  - Instructions perform the same operation in all lanes



NEON Pipeline