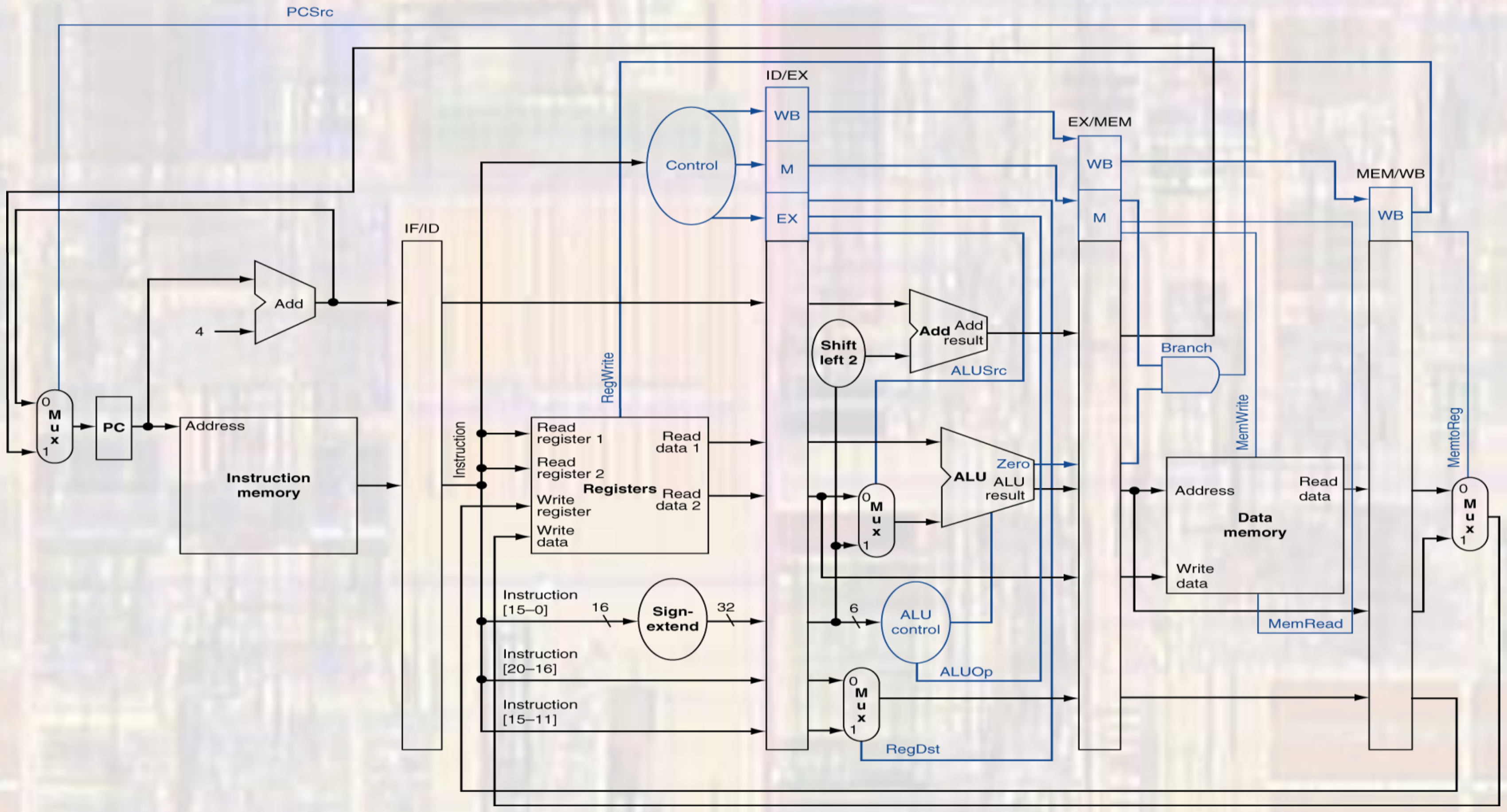
Processor Architecture Pipeline Hazards

Last modified 4/4/24

Pipeline Hazards

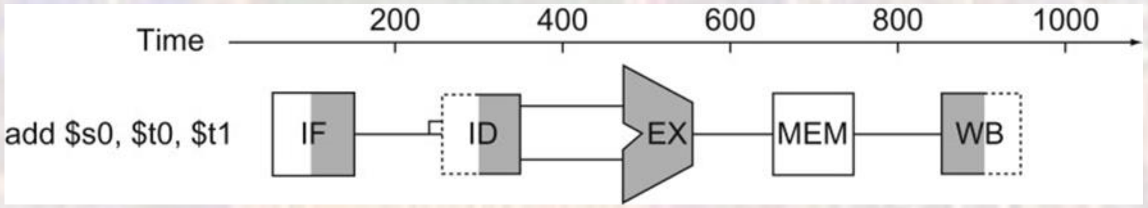
- Pipeline



Pipeline Hazards

- Pipeline Operation

WB actually occurs in the first half of the next clock cycle



- In this example
  - Reads are done from program memory and the register file
  - The ALU executes
  - The data memory is not used
  - Write is done on the register file

Pipeline Hazards

- Pipeline Hazards
  - Hazards are conditions where the next instruction cannot perform its assigned pipeline action in the next clock cycle
  - 3 types
    - Structural
    - Data
    - Control

Pipeline Hazards

- Structural Hazards
  - These hazards result from a **resource** conflict
  - Classic case is Harvard vs. vonNeuman memory architectures
    - vonNeuman architectures share a single memory for program and data
    - A lw or sw command requires access to data memory to load or store the data value
    - It would not be possible to fetch the appropriate instruction during this clock cycle since the memory would be in use
    - The IF would be stalled and a “bubble” would be created in the pipeline

Pipeline Hazards

- Structural Hazards

  - vonNeuman memory architecture

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	LW	2	3	Stall	4	5	6	7	8	9	10	11	12	13	14
ID		LW	2	3	bubble	4	5	6	7	8	9	10	11	12	13
EX			LW	2	3	bubble	4	5	6	7	8	9	10	11	12
MEM				LW	2	3	bubble	4	5	6	7	8	9	10	11
WB					LW	2	3	bubble	4	5	6	7	8	9	10

data memory access prevents a concurrent instruction fetch

Pipeline Hazards

- Data Hazards
  - These hazards result from a **dependence** of one instruction on another instruction still in the pipeline
  - Consider the following code snippit

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

    - The value of $s0 is needed to perform the subtraction

Pipeline Hazards

- Data Hazards

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

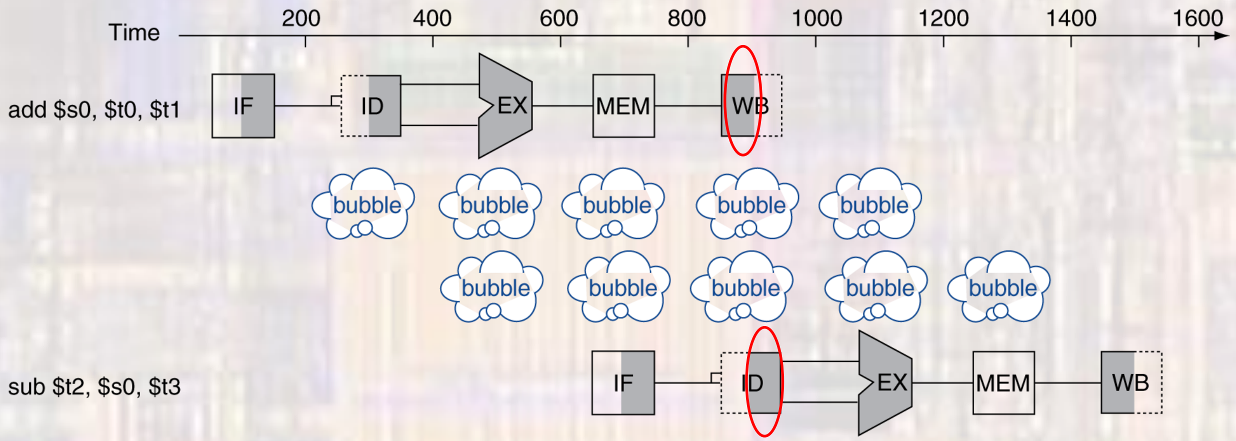
Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	sub	sub	sub	3	4	5	6	7	8	9	10	11	12	13
ID		add	stall	stall	sub	3	4	5	6	7	8	9	10	11	12
EX			add	bubble	bubble	sub	3	4	5	6	7	8	9	10	11
MEM				add	bubble	bubble	sub	3	4	5	6	7	8	9	10
WB					add	bubble	bubble	sub	3	4	5	6	7	8	9

- 2 clock cycle bubbles are created
- It would be 3 bubbles – except we can take advantage of our convention
  - writes occur in the first half of the clock cycle
  - reads occur in the second half of the clock cycle
  - the WB occurs during the same clock cycle as the register read

Pipeline Hazards

- Data Hazards

add $s0, $t0, $t1
sub $t2, $s0, $t3



- 2 clock cycle bubbles are created
- It would be 3 bubbles – except we can take advantage of our convention
  - writes occur in the first half of the clock cycle
  - reads occur in the second half of the clock cycle
  - the WB occurs during the same clock cycle as the register read

Pipeline Hazards

- Data Hazards

  - In many cases the compiler can avoid a data hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
or    $s2, $t0, $t1
and   $s3, $t0, $t3
add   $s4, $t1, $t3
```

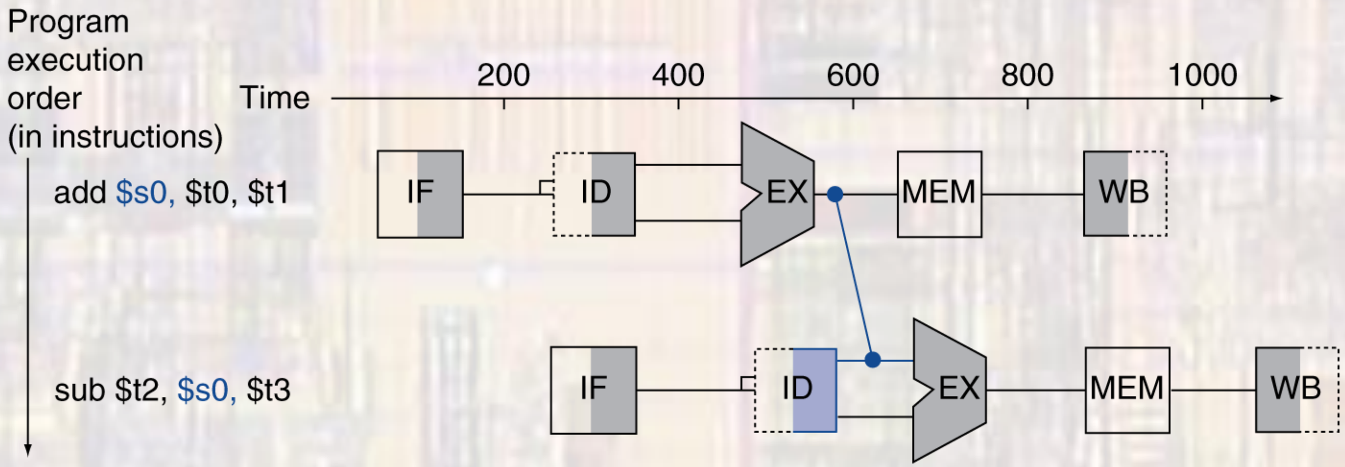
re-order the instruction to remove the hazard condition

```
add   $s0, $t0, $t1
or    $s2, $t0, $t1
and   $s3, $t0, $t3
add   $s4, $t1, $t3
sub   $t2, $s0, $t3
```

Pipeline Hazards

- Data Hazards

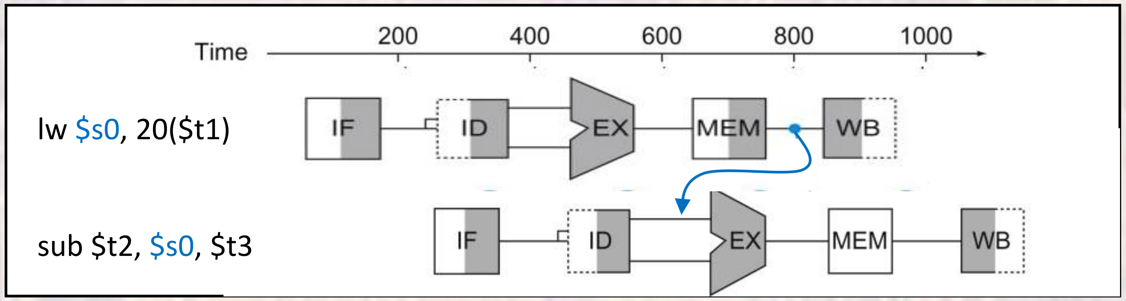
  - Hardware can also be used to avoid data hazards
    - called forwarding or bypassing
    - provide the needed data as soon as it is valid
    - requires extra circuitry

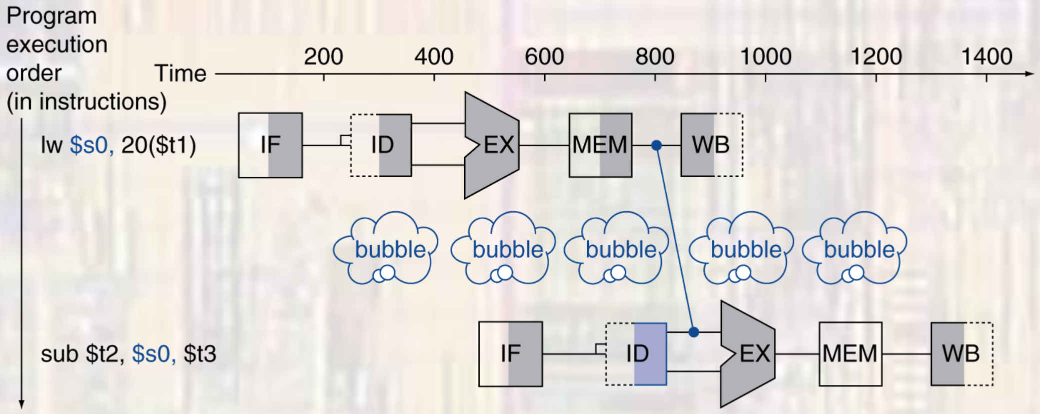


Pipeline Hazards

- Data Hazards

  - Hardware cannot avoid all data hazards
    - cannot go backwards in time !





Pipeline Hazards

- Data Hazards

  - Forwarding plus compiler optimizations can avoid additional data hazards

```
          lw    $t1, 0($t0)
          lw    $t2, 4($t0)
stall  →  add   $t3, $t1, $t2
          sw    $t3, 12($t0)
          lw    $t4, 8($t0)
stall  →  add   $t5, $t1, $t4
          sw    $t5, 16($t0)
```

13 cycles

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
lw    $t4, 8($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

11 cycles

Pipeline Hazards

- Control Hazards

  - These hazards result from making a **decision** while other instructions continue to progress through the pipeline

  - Branch instructions are the most common example
    - don’t know whether to load the next instruction or not

  - three approaches
    - stall
    - predict
    - delay

Pipeline Hazards

- Control Hazards - stall

  - Do not load the next instruction into the pipeline

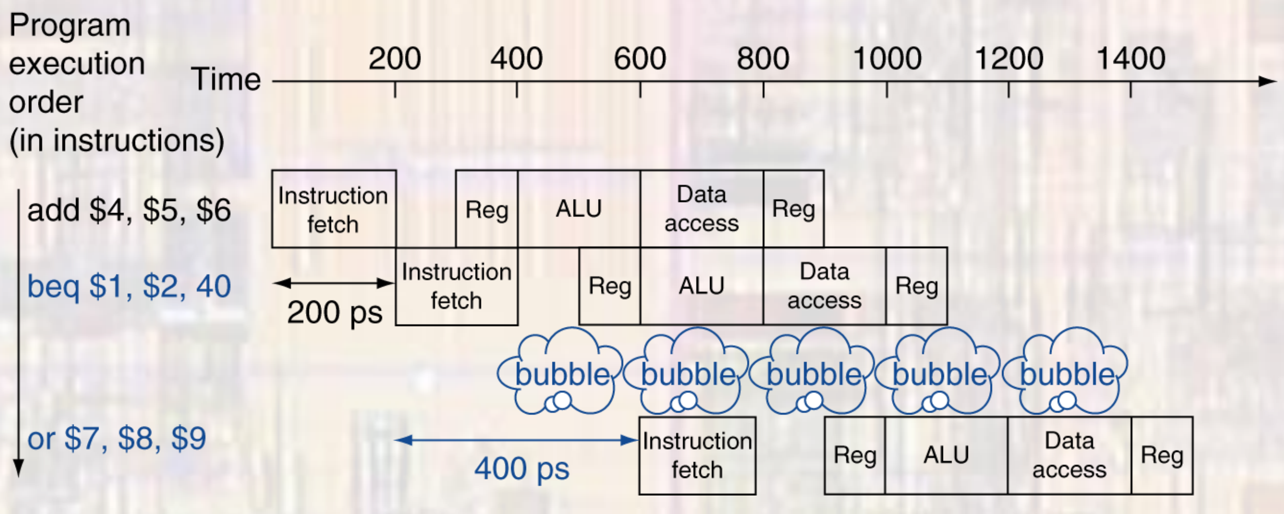
Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	beq	3	3	8	9	10	11	12	13	14	15	16	17	18
ID		add	beq	stall	stall	8	9	10	11	12	13	14	15	16	17
EX			add	beq	bubble	bubble	8	9	10	11	12	13	14	15	16
MEM				add	beq	bubble	bubble	8	9	10	11	12	13	14	15
WB					add	beq	bubble	bubble	8	9	10	11	12	13	14

  - during decode – know you have a branch
  - during execute – know if taking branch or not
    - PC will be updated
  - Next cycle – fetch the next instruction based on PC value

Pipeline Hazards

- Control Hazards - stall

  - Even if you add circuitry to detect the branch and update the PC all during the decode – can't avoid a stall



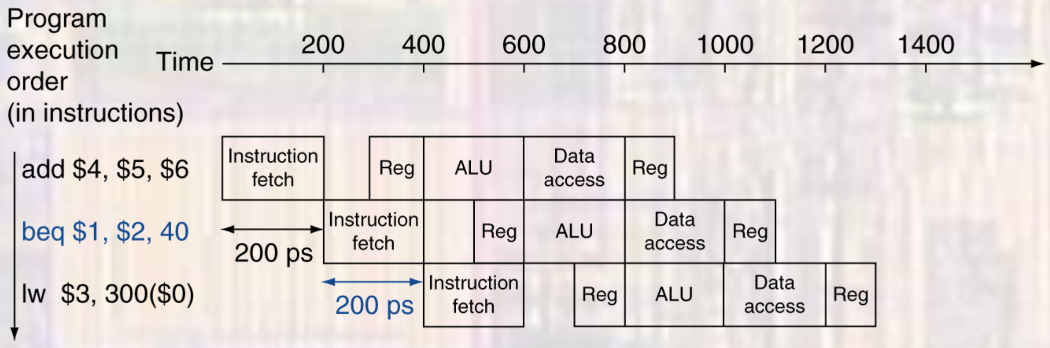
Pipeline Hazards

- Control Hazards - predict
  - Many algorithms
  - Simplest – assume branch will not be taken
    - no penalty if correct
    - stall only when wrong

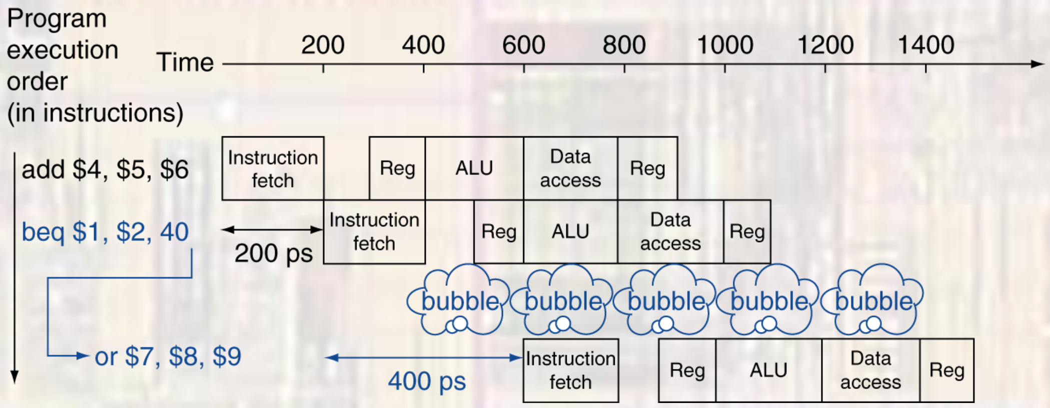
Pipeline Hazards

- Control Hazards – predict
  - Predict branch not taken

Branch Not Taken
Prediction correct!



Branch Taken
Prediction wrong!



Pipeline Hazards

- Control Hazards - predict
  - Static Branch Prediction
    - Predict backward branches - taken
    - Predict forward branches – not taken
    - Looping code
      - executes the loop 100 times
      - jumps out of the loop 1 time
  - Dynamic Branch Prediction
    - Keep track of recent branch behavior (for each branch)
    - Assume recent behavior will continue
    - When wrong – clear history and start over
    - Hardware intensive

Pipeline Hazards

- Control Hazards - delay
  - Delayed Decision
    - Pipeline always executes the instruction immediately after the branch
    - The branch then executes (only 1 cycle delay allowed)
    - Requires the next instruction to be **independent** of the branch decision
    - Compiler is designed to set this up

Pipeline Hazards

- Control Hazards - delay

  - Delayed Decision (assume HW to limit bubble to 1 cycle)

```
add $t0,$t1,$t2
beq   $t1,$t2,-30
…
```

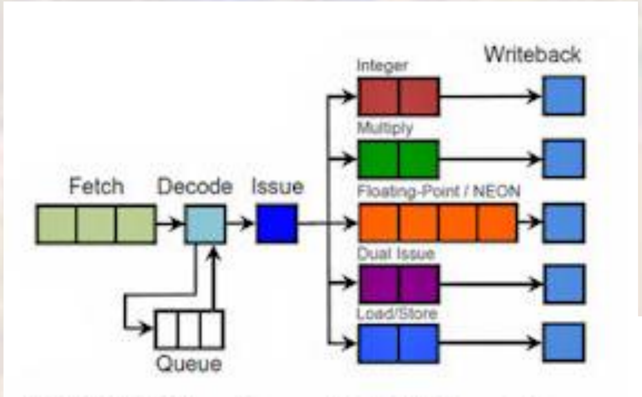
Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	beq	3	8	9	10	11	12	13	14	15	16	17	18	19
ID		add	beq	stall	8	9	10	11	12	13	14	15	16	17	18
EX			add	beq	bubble	8	9	10	11	12	13	14	15	16	17
MEM				add	beq	bubble	8	9	10	11	12	13	14	15	16
WB					add	beq	bubble	8	9	10	11	12	13	14	15

re-order

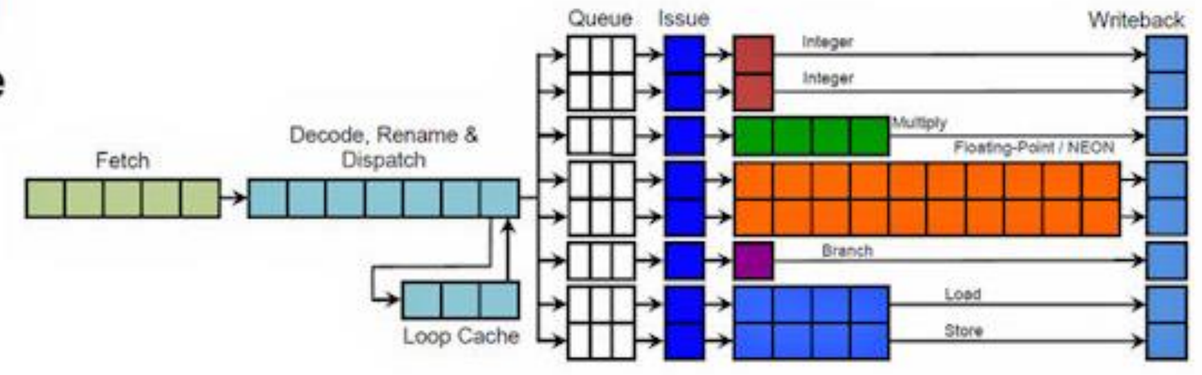
Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	beq	add	8	9	10	11	12	13	14	15	16	17	18	19	20
ID		beq	add	8	9	10	11	12	13	14	15	16	17	18	19
EX			beq	add	8	9	10	11	12	13	14	15	16	17	18
MEM				beq	add	8	9	10	11	12	13	14	15	16	17
WB					beq	add	8	9	10	11	12	13	14	15	16

Pipelining

- Superscalar
  - Parallelism at the micro-architecture level



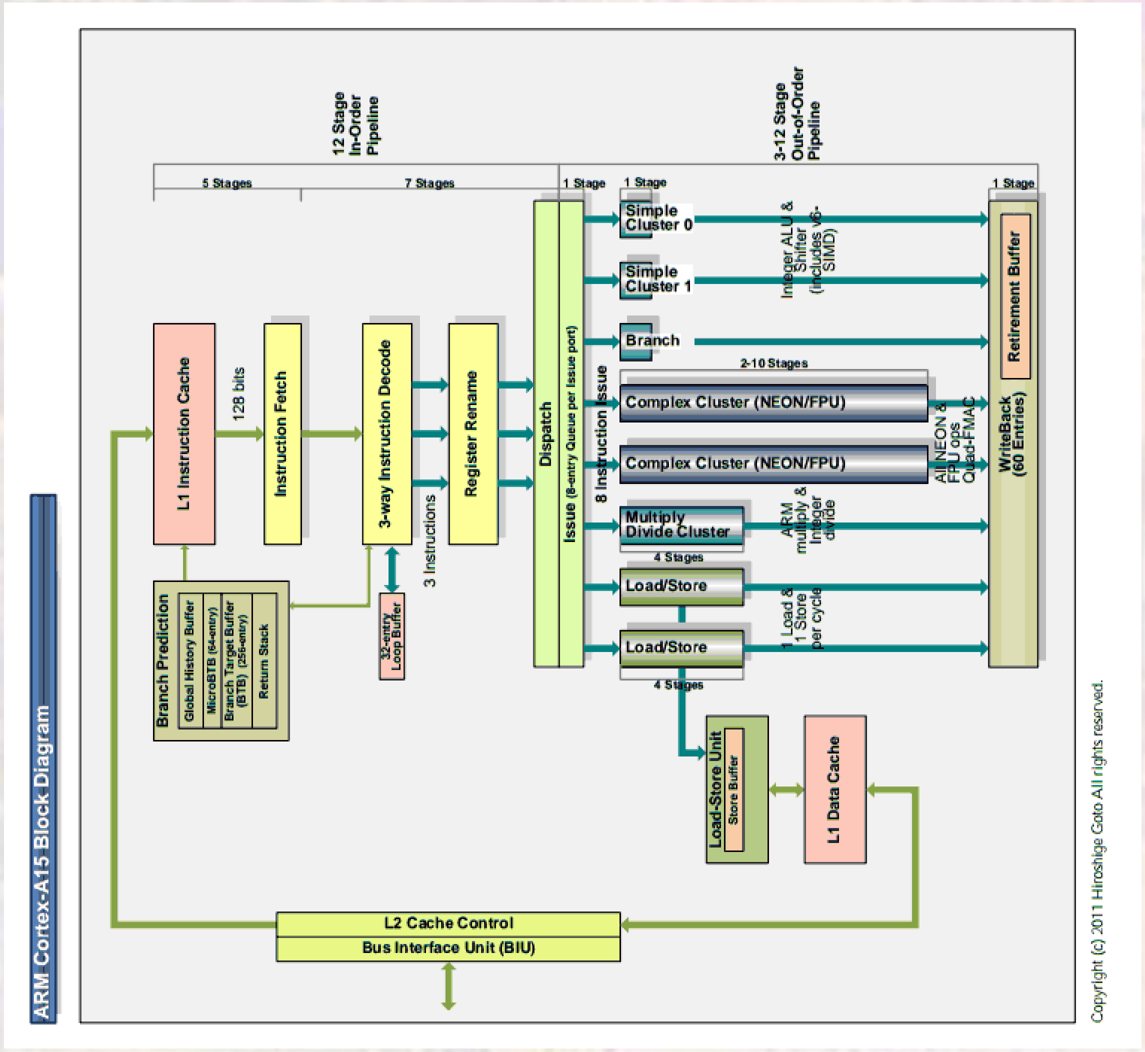
ARM Cortex-A7 Pipeline



ARM Cortex-A15 Pipeline

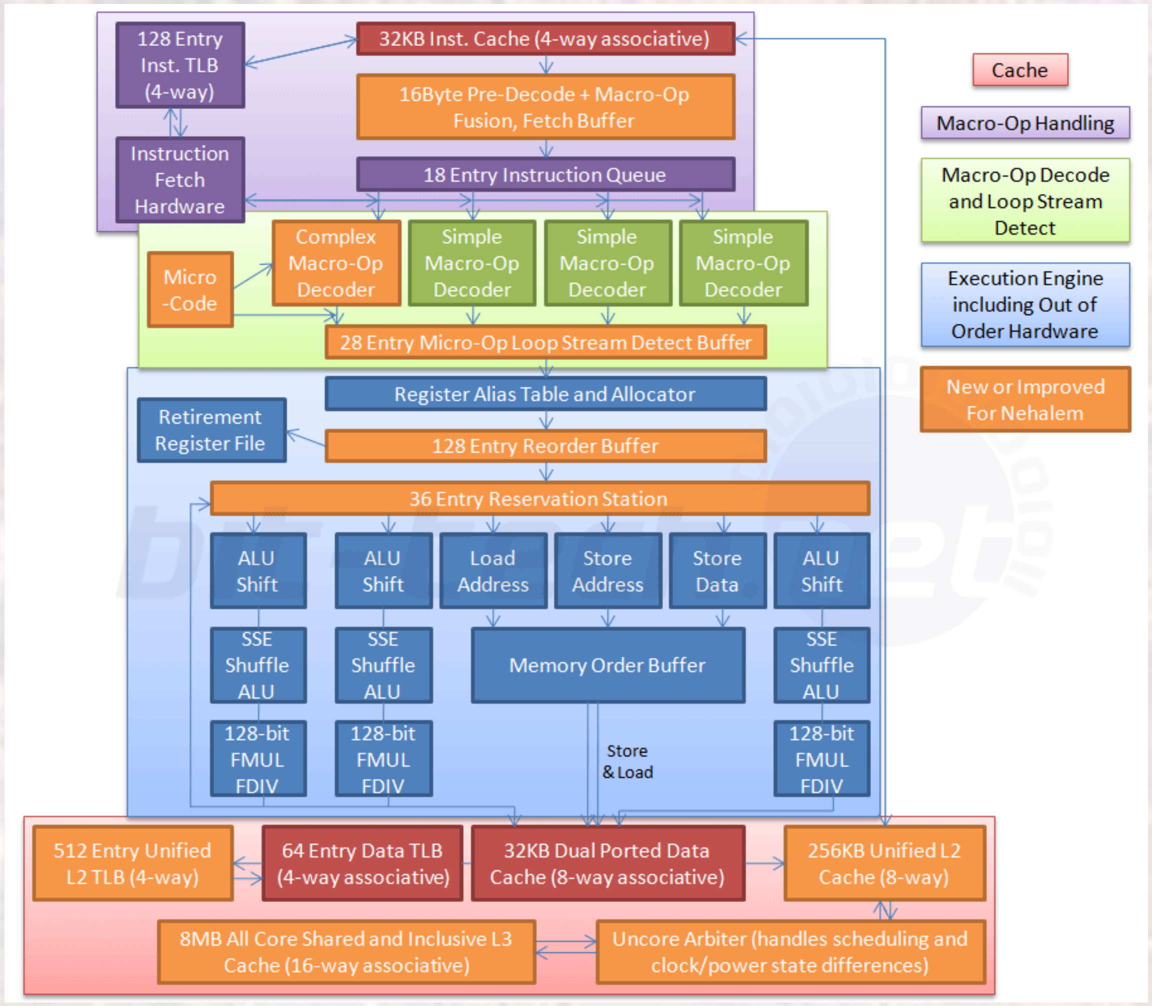
Pipelining

- Processor Architecture



Pipelining

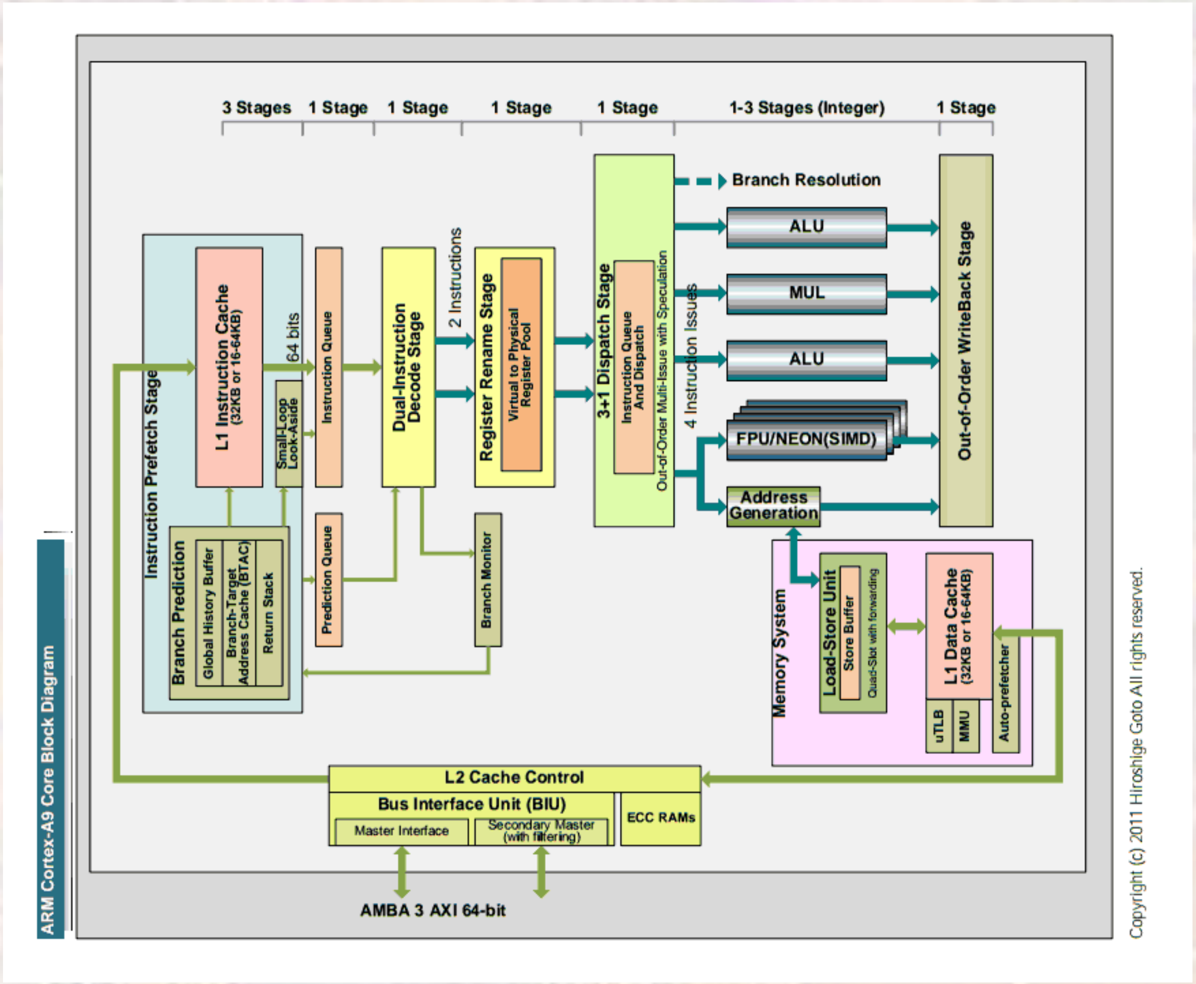
- Processor Architecture



128 Entry Inst. TLB (4-way)
32KB Inst. Cache (4-way associative)
16Byte Pre-Decode + Macro-Op Fusion, Fetch Buffer
Instruction Fetch Hardware
18 Entry Instruction Queue
Micro -Code
Complex Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
28 Entry Micro-Op Loop Stream Detect Buffer
Register Alias Table and Allocator
Retirement Register File
128 Entry Reorder Buffer
36 Entry Reservation Station
ALU Shift
ALU Shift
Load Address
Store Address
Store Data
ALU Shift
SSE Shuffle ALU
SSE Shuffle ALU
Memory Order Buffer
SSE Shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & Load
128-bit FMUL FDIV
512 Entry Unified L2 TLB (4-way)
64 Entry Data TLB (4-way associative)
32KB Dual Ported Data Cache (8-way associative)
256KB Unified L2 Cache (8-way)
8MB All Core Shared and Inclusive L3 Cache (16-way associative)
Uncore Arbiter (handles scheduling and clock/power state differences)
Cache
Macro-Op Handling
Macro-Op Decode and Loop Stream Detect
Execution Engine including Out of Order Hardware
New or Improved For Nehalem

Pipelining

- Modern Example



3 Stages
1 Stage
1 Stage
1 Stage
1 Stage
1-3 Stages (Integer)
1 Stage
Instruction Prefetch Stage
L1 Instruction Cache (32KB or 16-64KB)
64 bits
Small-Loop Look-Aside
Instruction Queue
Dual-Instruction Decode Stage
2 Instructions
Register Rename Stage
Virtual to Physical Register Pool
3+1 Dispatch Stage
Instruction Queue And Dispatch
Out-of-Order Multi-Issue with Speculation
4 Instruction Issues
Branch Resolution
ALU
MUL
ALU
FPU/NEON(SIMD)
Address Generation
Out-of-Order WriteBack Stage
Branch Prediction
Global History Buffer
Branch-Target Address Cache (BTAC)
Return Stack
Prediction Queue
Branch Monitor
Memory System
Load-Store Unit
Store Buffer
Quad-Slot with forwarding
L1 Data Cache (32KB or 16-64KB)
uTLB
MMU
Auto-prefetcher
L2 Cache Control
Bus Interface Unit (BIU)
Master Interface
Secondary Master (with filtering)
ECC RAMs
AMBA 3 AXI 64-bit
ARM Cortex-A9 Core Block Diagram
Copyright (c) 2011 Hiroshige Goto All rights reserved.