Central Processing Unit

- **Sequential execution**
- **Pipelining & Instruction Level Parallelism**
- **Multiple Issue / Superscalar processors**

MIPS (Microprocessor without Interlocked Pipe Stages)

processing of the most common types of instructions:

add values in registers t1 & t2
store the result in reg. t0

add $t0, $t1, $t2

load data (word) to register

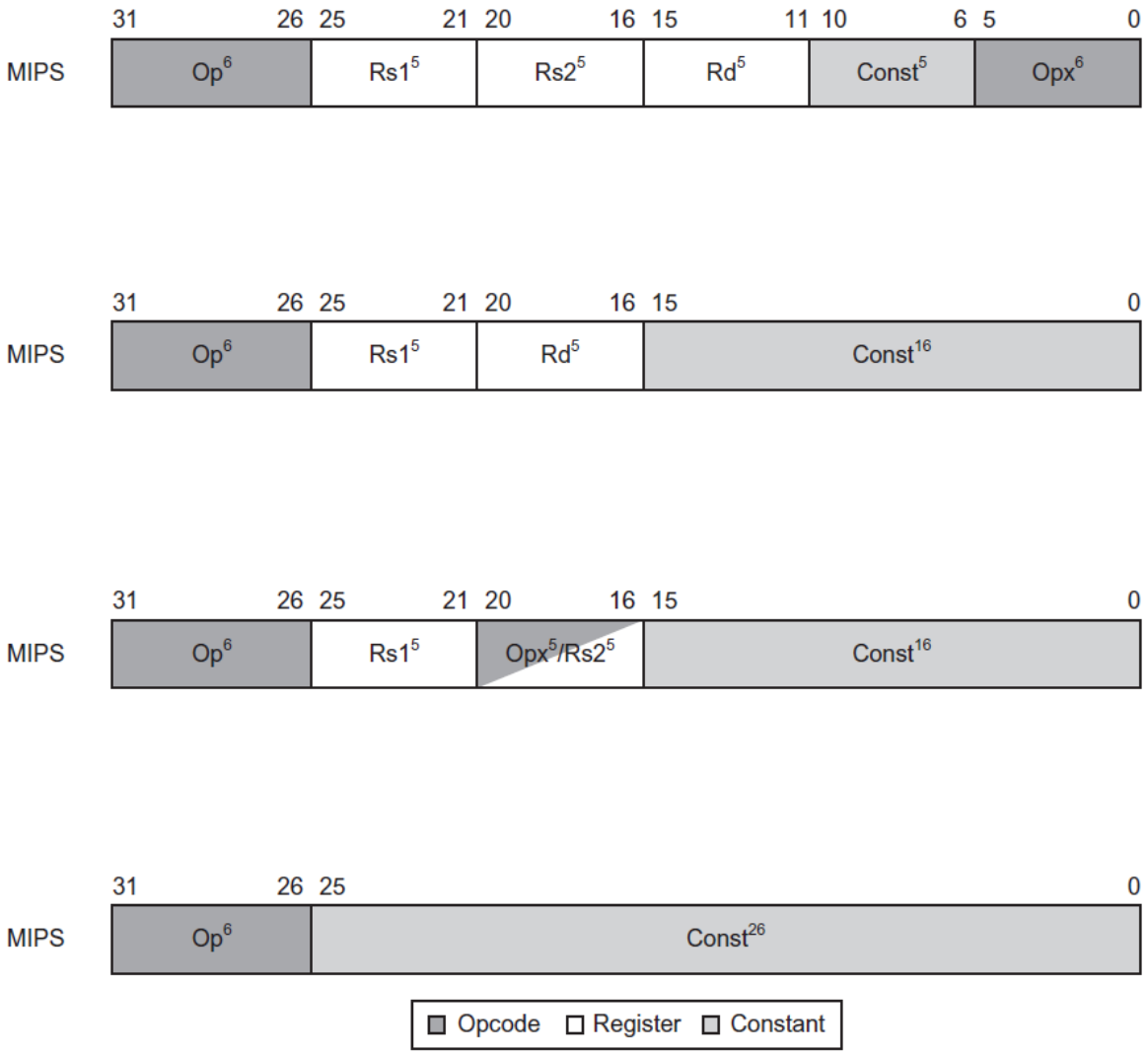
lw $t0, offset($t2)

conditional **b**ranch
if **eq**ual
(jump if equal)

beq $t0, $t1,label

unconditional
jump

coding schemes



Instruction cycle – MIPS

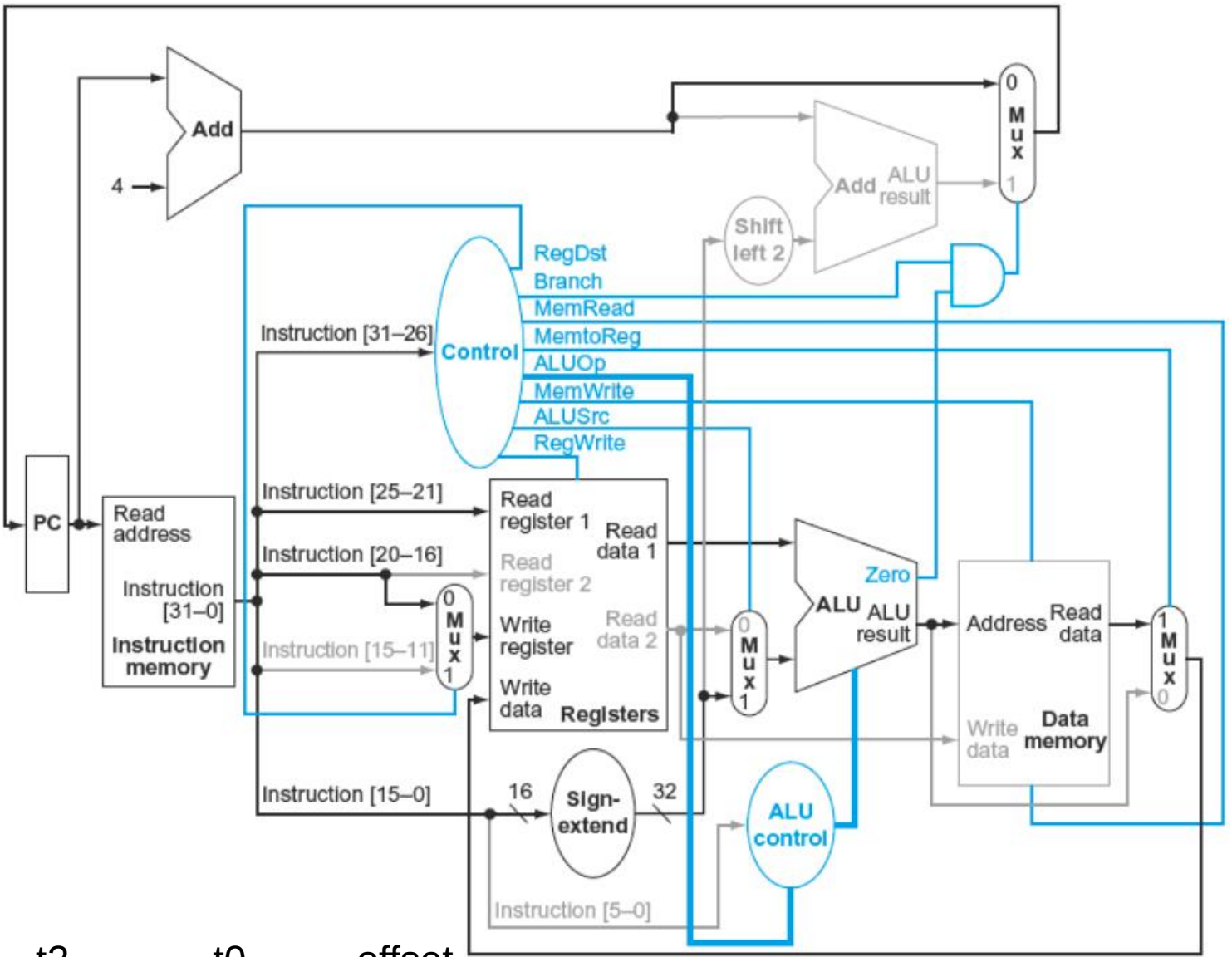
- IF – instruction fetch (from program memory), increment the Program Counter (PC)
- ID – instruction decode and register readout
- EX – execution (or address calculation)
- MEM – data memory access
- WB – write back – write the result back to the register(s)

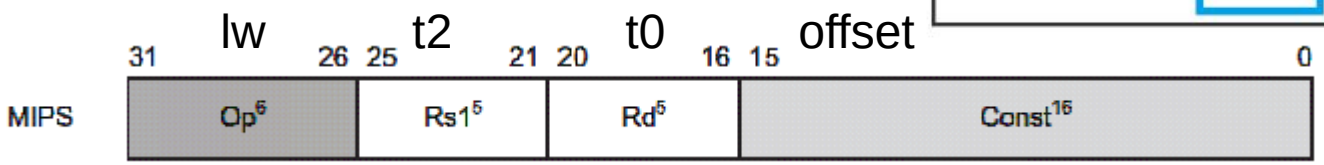
PC – Program Counter – (Intel x86: IP/EIP/RIP – Instruction Pointer)
holds the **address of the next instruction** to fetch!

MIPS – load a 32bit word from address = (offset + t2) to register t0

lw $t0, offset($t2)

1. Instruction fetch
2. PC=PC+4
3. Read the value from register t2
4. Compute the address: t2+offset
5. Memory access
6. Store the contents in register t0





MIPS – simple register operation

add $t0, $t1, $t2

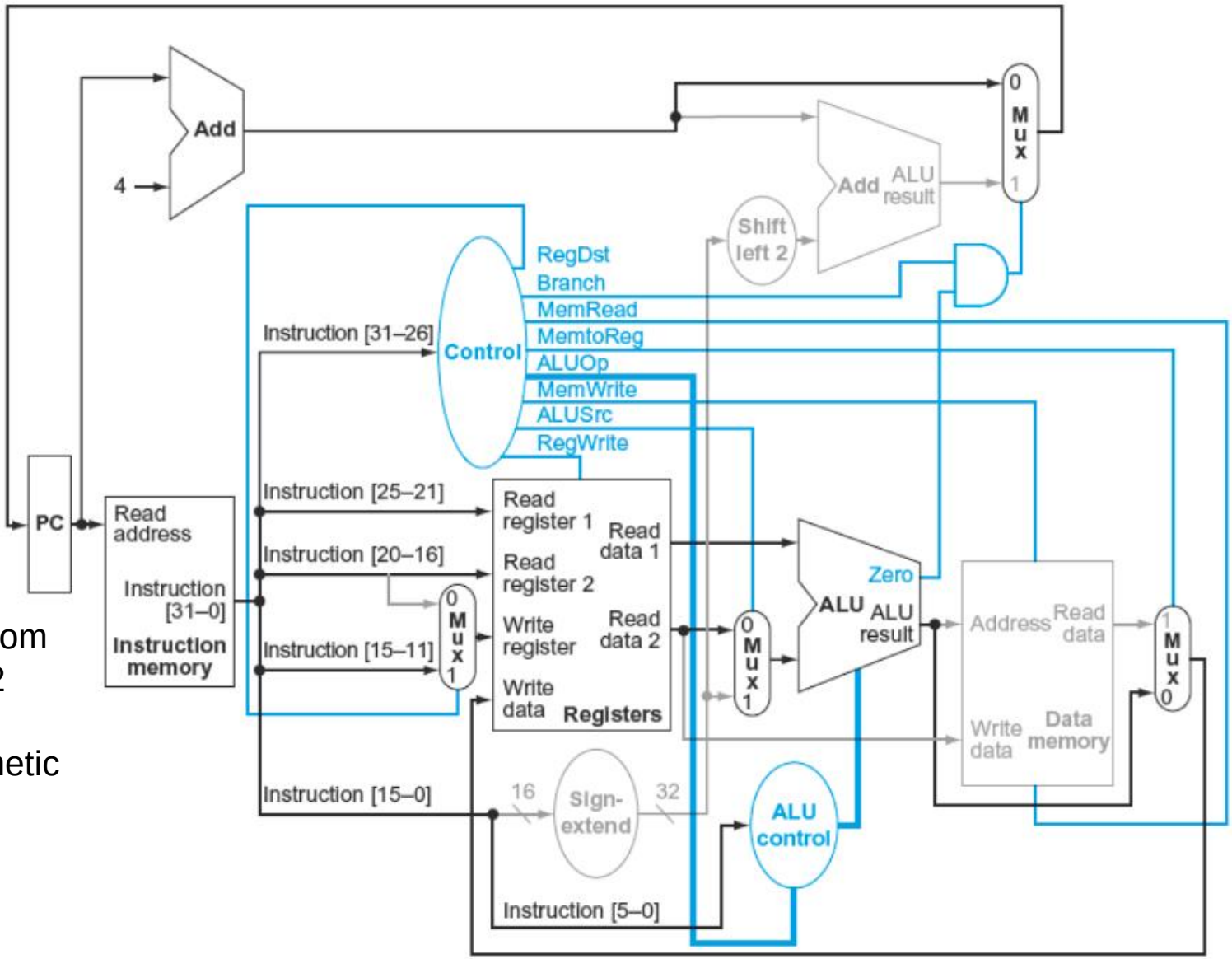
t0=t1+t2

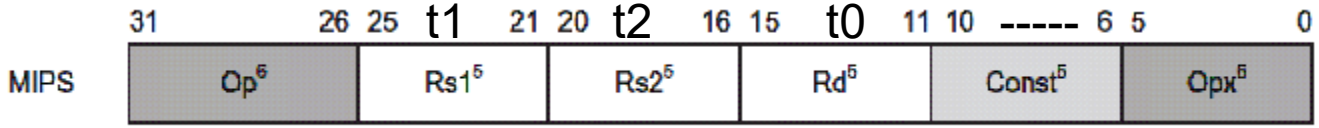
1. Instruction fetch
2. PC=PC+4

3. Read the values from registers t1 and t2

3. Perform the arithmetic operation

4. Store the result in t0





MIPS – conditional branch (conditional jump) – optimized!

beq $t0, $t1,label

compare registers:
t0 with t1
and jump to label
if equal

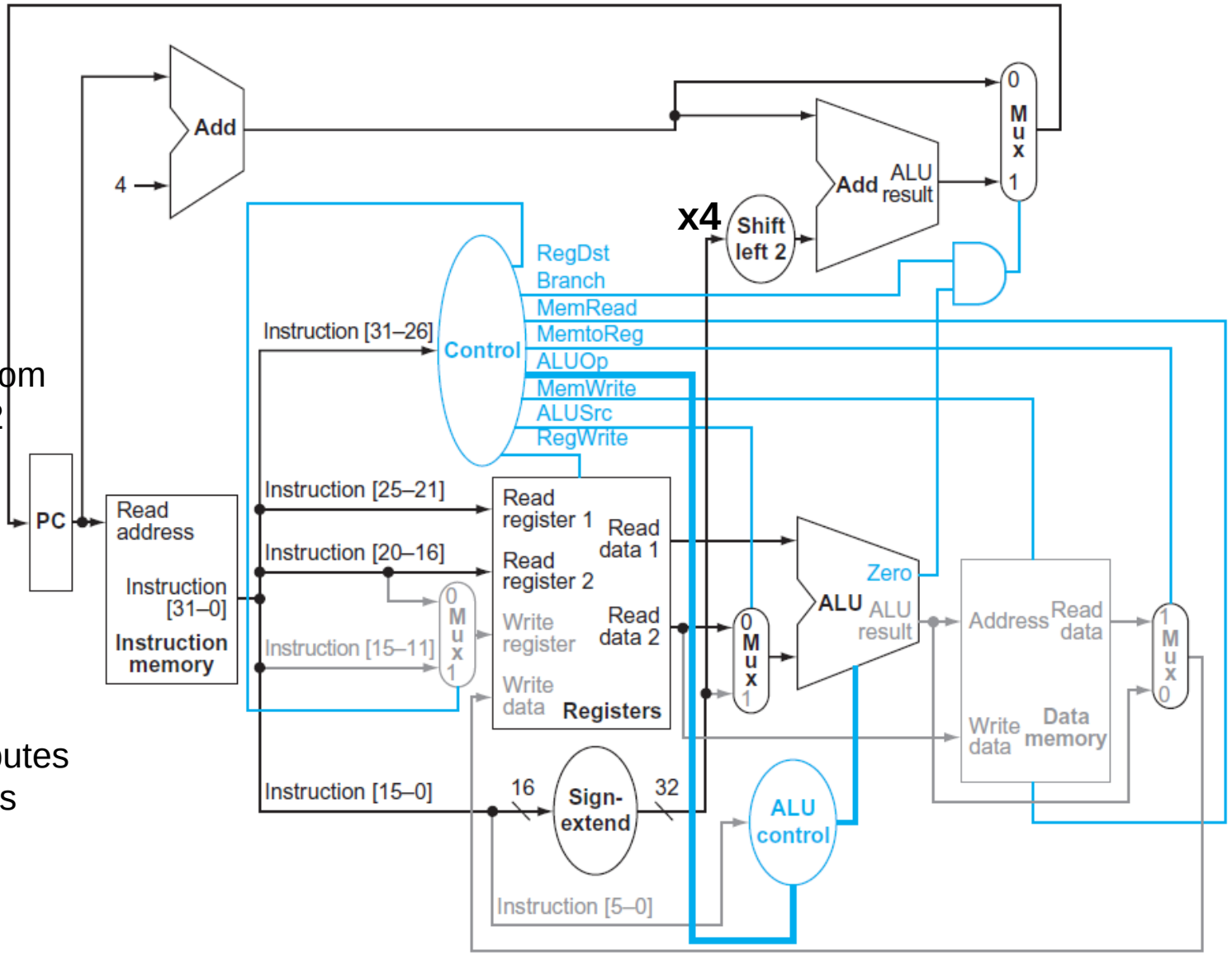
1. Instruction fetch
2. PC=PC+4
3. Read the values from
   registers t1 and t2

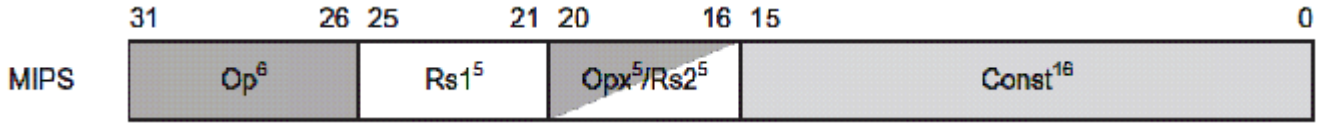
4. ALU: compare:
   t0 - t2
   set the ZERO flag

Simultaneously:

5. Second ALU computes
   destination address
   of the jump
   PC+(const*4)

6. If ZERO_flag=1
reprogram the PC
with destination address





MIPS – unconditional „far” jump

1. Instruction fetch
2. PC=PC+4

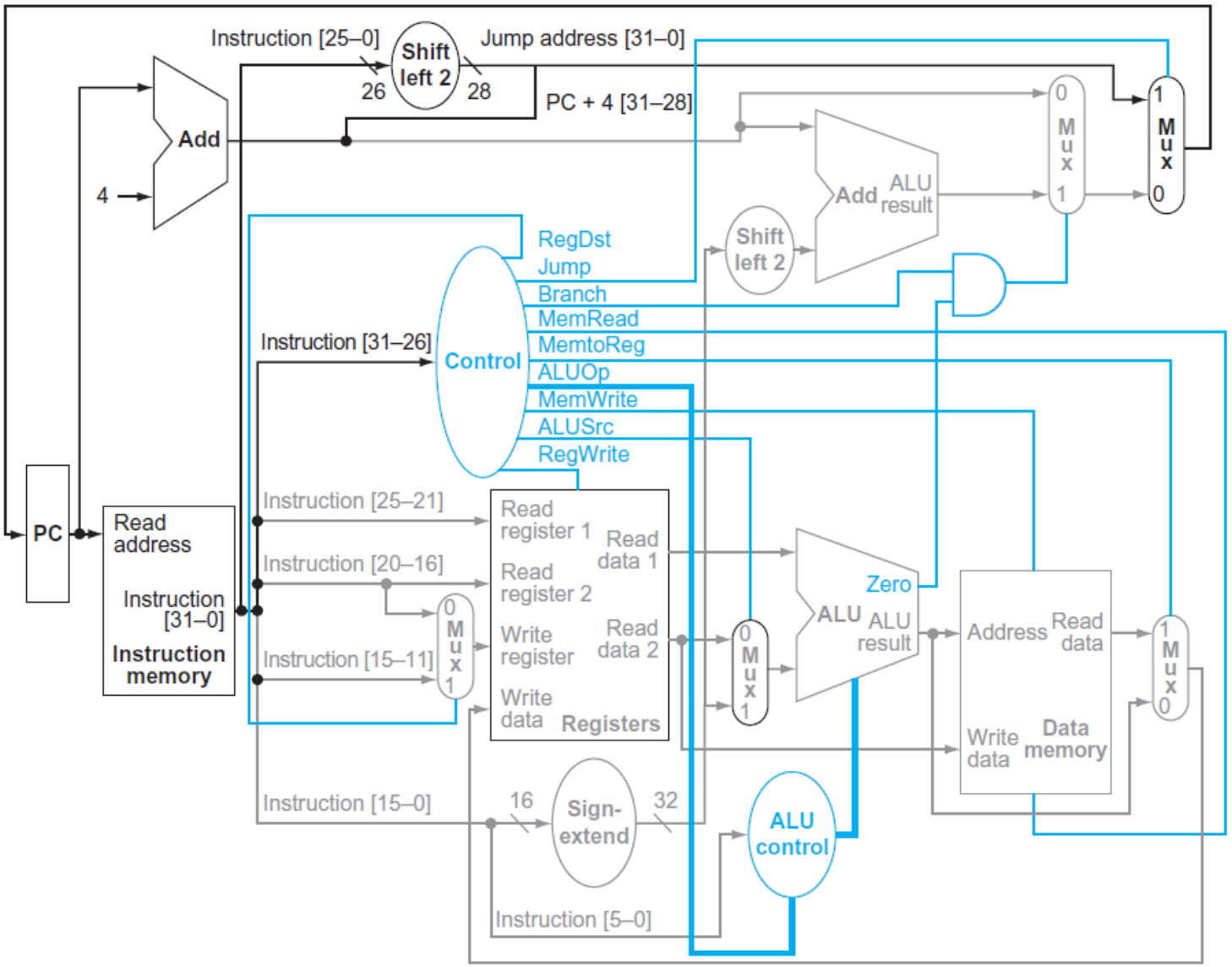
3. Destination address of the jump:

- lower 28bits:

26bits taken from the argument, then shifted 2 bits left

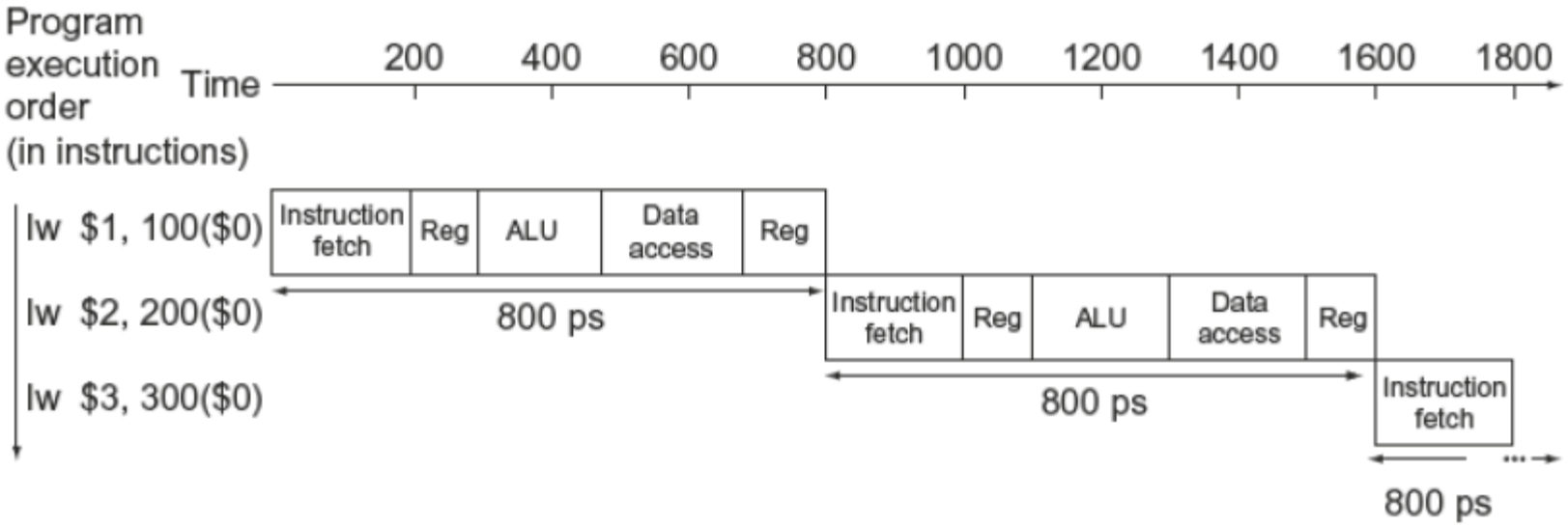
- higher 4bits directly copied from the PC

3. PC is set to the new value

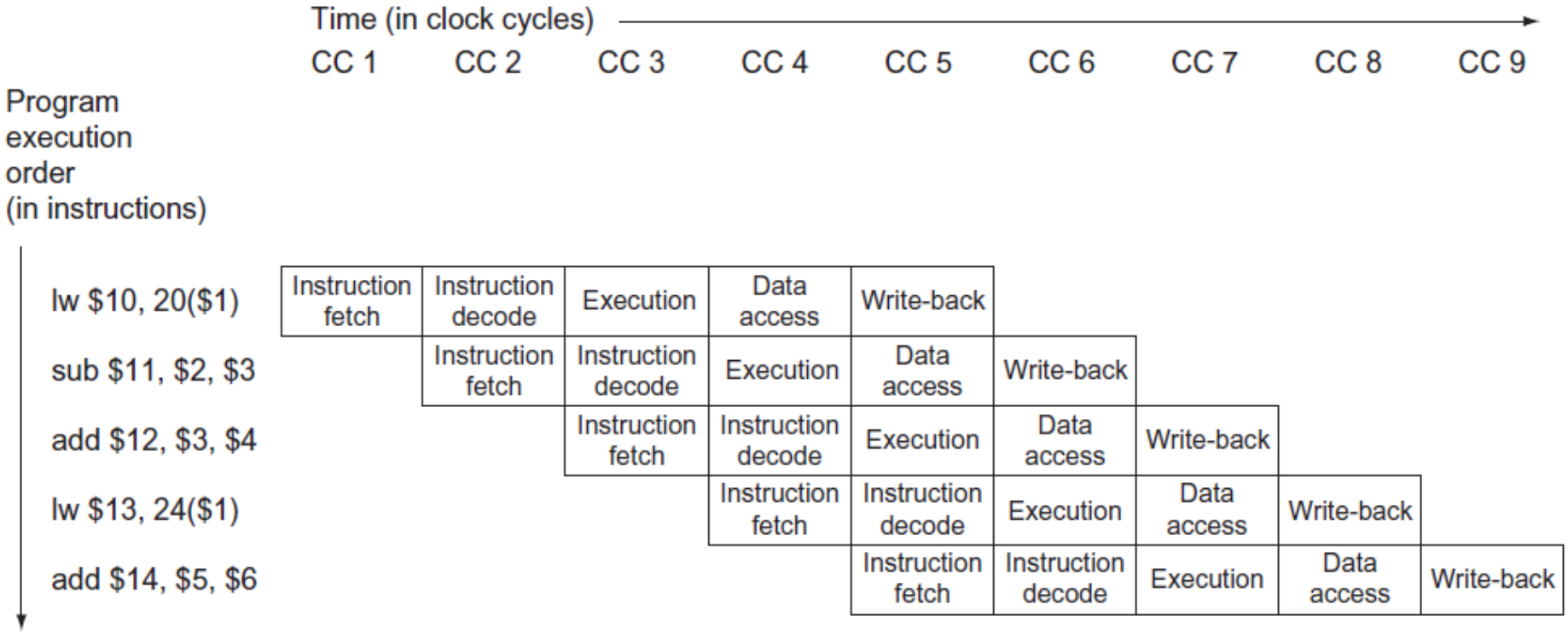


Sequential execution

- assumes that each instruction completes before the next one begins
- subsequent phases of instruction cycle are performed by different processor blocks
- single-cycle design is possible but inefficient:
  - clock cycle must have the same length for every instruction
  - the longest possible path in the processor determines the clock cycle. (clock cycle is equal to the worst-case delay for all instructions)



Pipelined execution



- data path is divided into multiple pipeline stages (here: 5)
- each instruction executes over multiple cycles (here: 5)
- consecutive instructions are overlapped in execution
- the last step of executing some instruction is finished in each clock cycle, so the throughput is 1 instruction per clock (IPC) cycle.

Pipelined execution

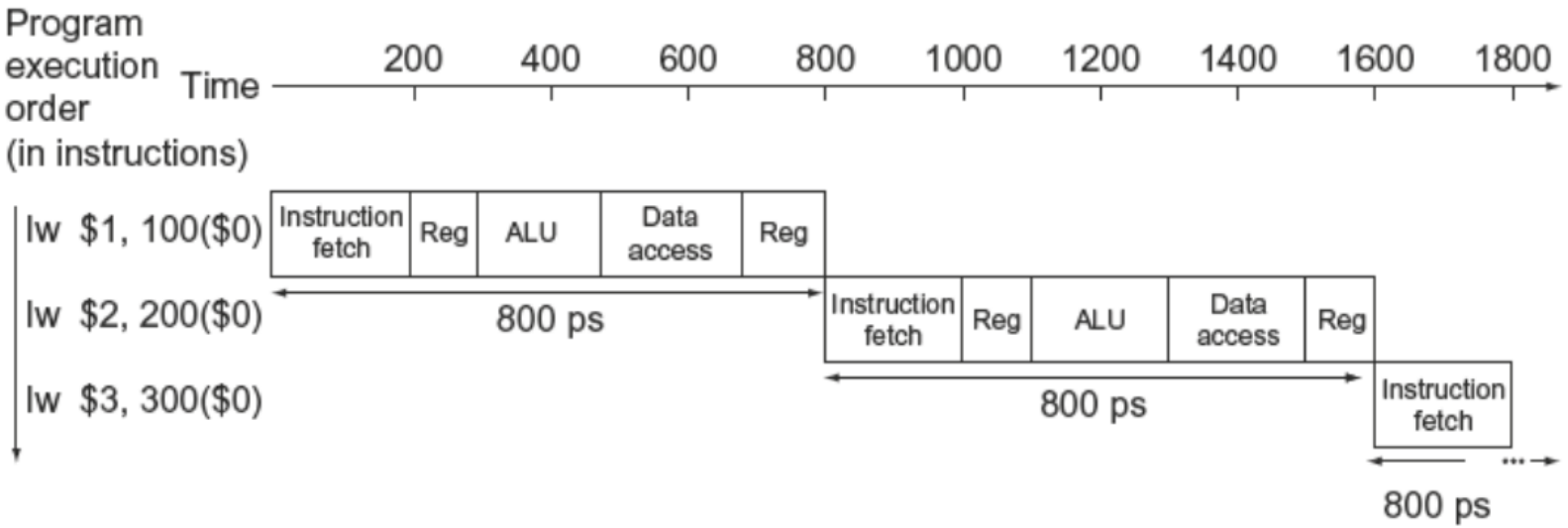
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

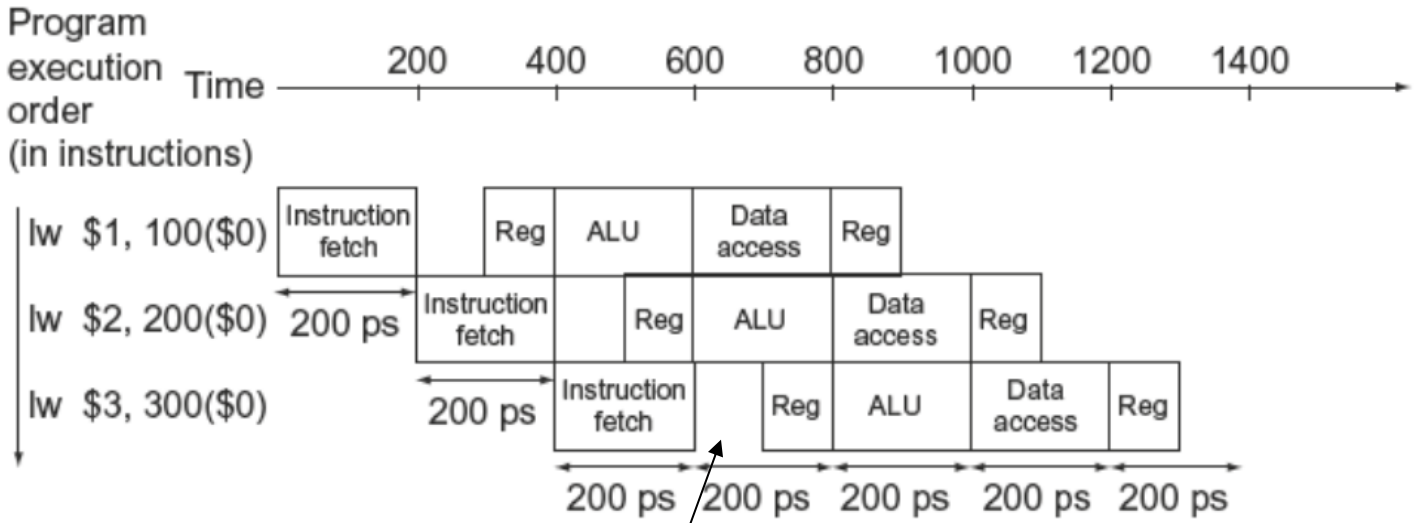
(Ideal) **Instruction Set Architecture for pipelining:**

Just a set of fast, short and simple instructions…

- instructions have (nearly) equal execution times
- instructions have the same length and a few simple encoding schemes
- memory operands only appear in load and store instructions
  (the rest uses registers as arguments)
- separated memory blocks and buses for instructions and data (Harvard arch.)

Pipelined execution





the pipeline cycle (frequency) has to be adjusted to longest phase/operation

Pipelined execution

3 instructions, sequential:

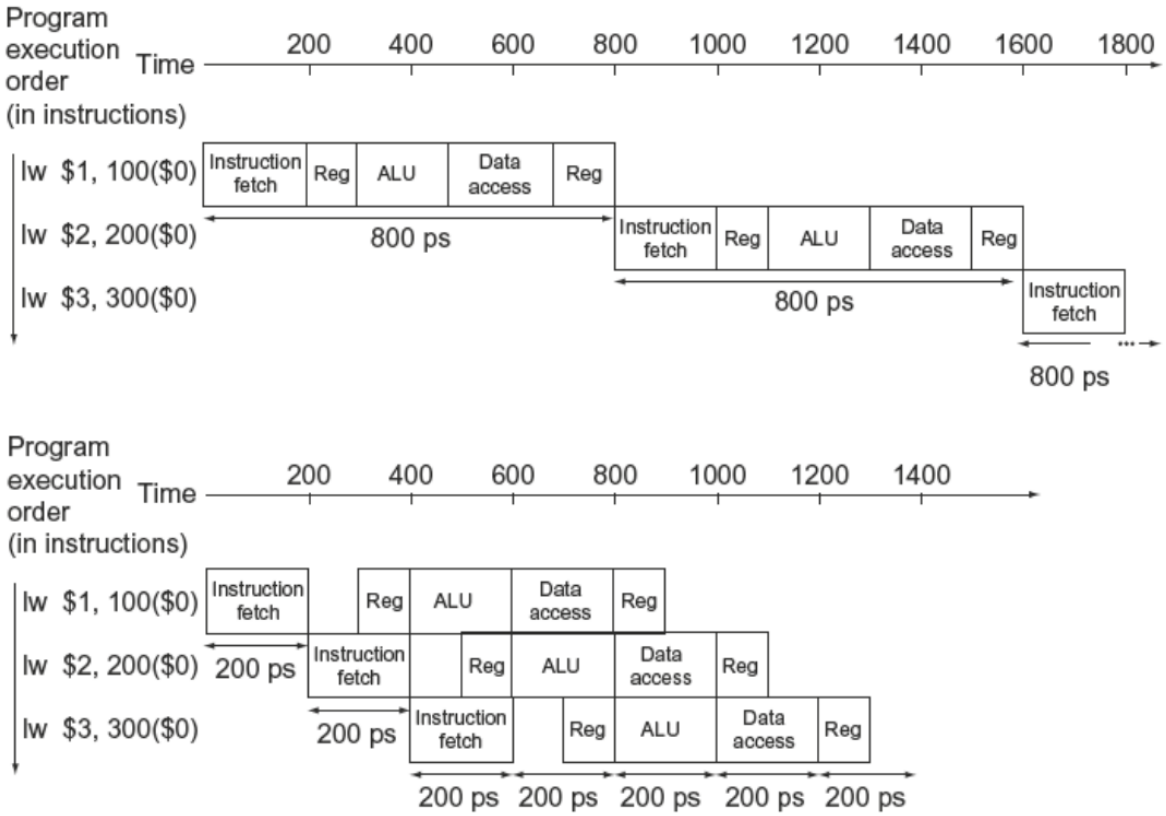
3*800ps=2400ps

3 instructions, pipelined:

1400ps

speed-up: 2400ps/1400ps = 1.7

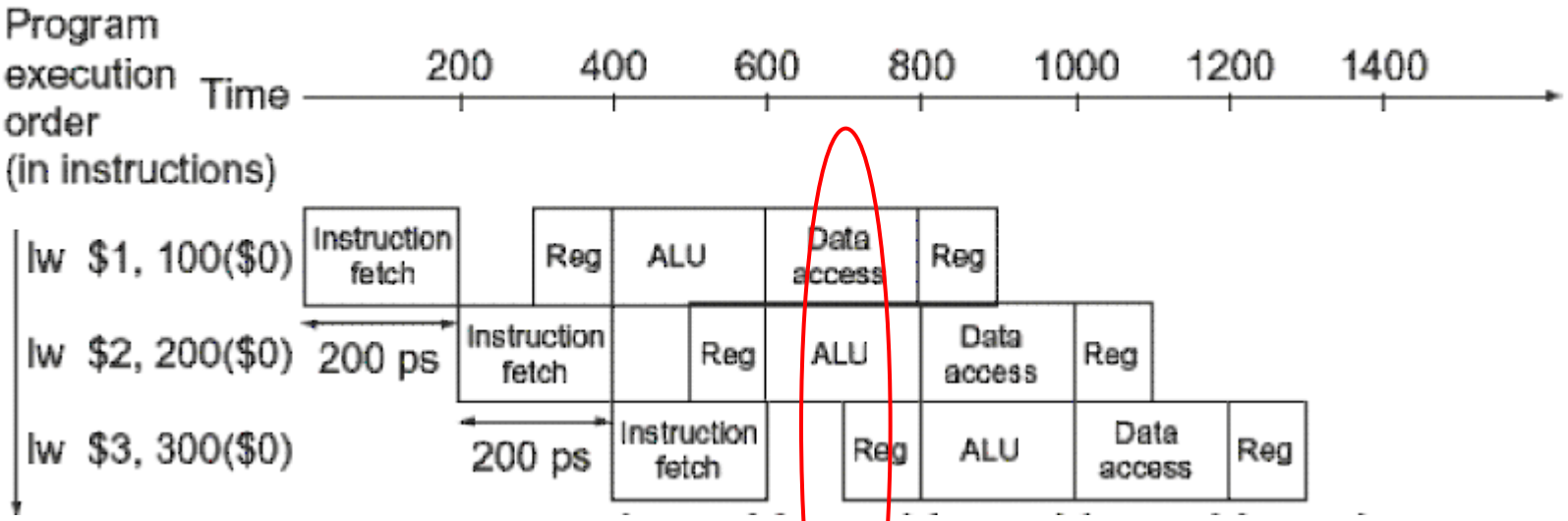
1000000 instr. sequential:
800ms
1000000 instr. pipelined:
approx. 200ms
approx. speed-up 4x



• under **ideal conditions** and with a **large number of instructions**,
the **speed-up** from pipelining **is approximately equal to the number of pipe stages**
e.g. a five-stage pipeline is nearly five times faster

Pipelined execution – problems…

structural hazards - instruction cannot execute in the proper clock cycle because the CPU does not support the combination of instructions that are set to execute



lw $4,400($0) | IF | REG | ALU | DA | REG |

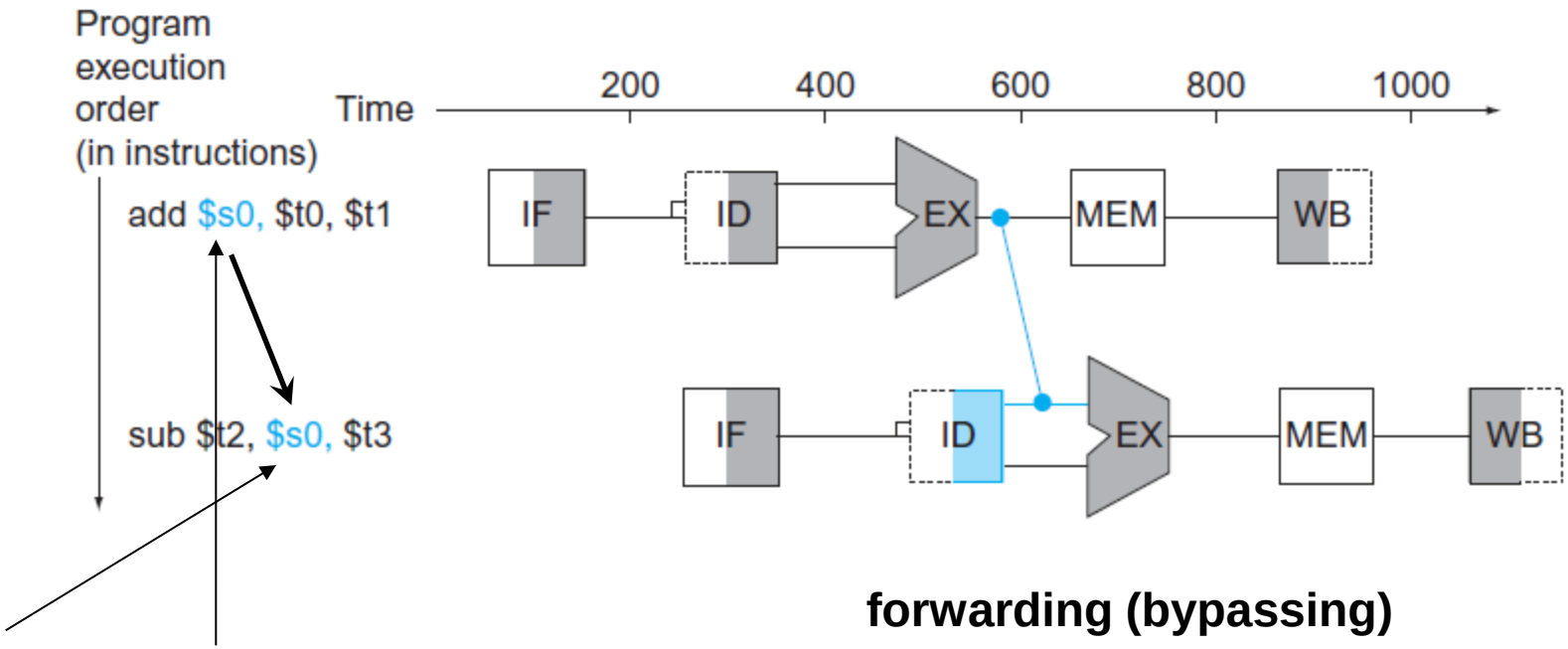
Concurrent memory access:
- load data to register
- instruction fetch

Solution: two memories and two buses for data and instructions…
Level 1 Cache - Harvard architecture!

Pipelined execution – problems…

data hazards / data dependencies

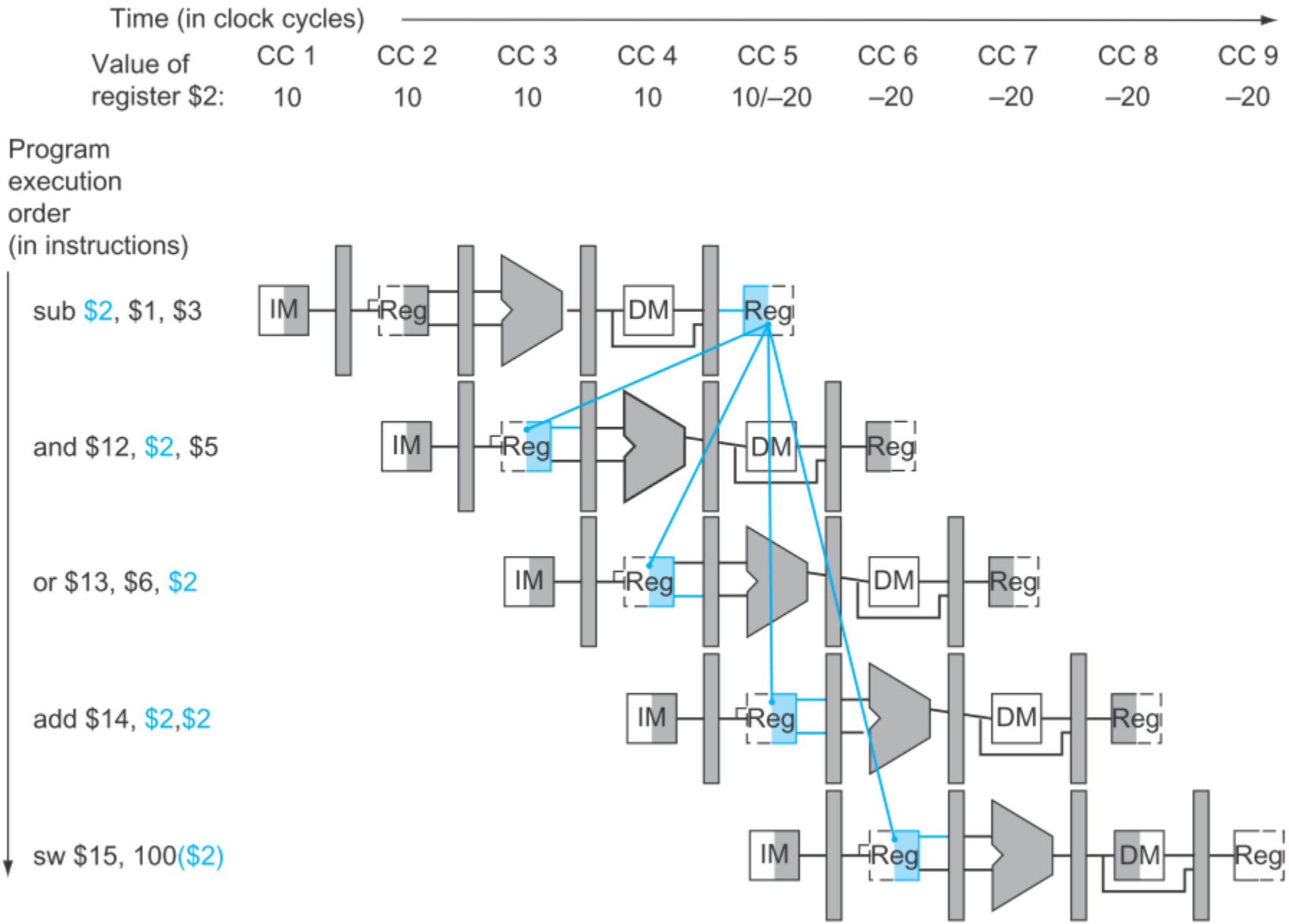
instruction cannot execute in the proper clock cycle because
data that is needed to execute the instruction is not yet available



forwarding (bypassing)

Read after Write – a true dependency:
argument of the 2nd instruction (sub) depends on the result of the previous one (add)

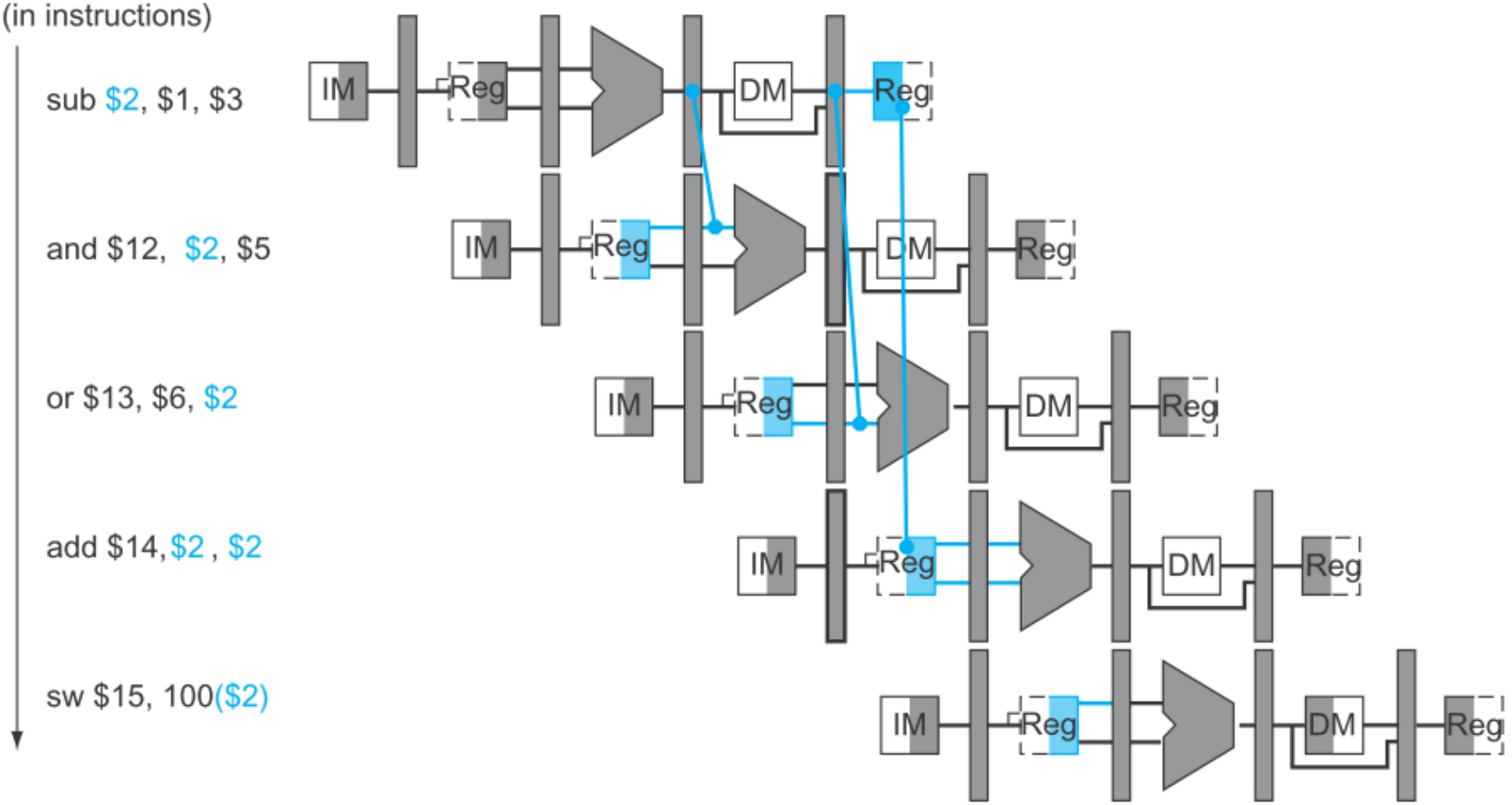
data hazards / data dependencies



Time (in clock cycles) →

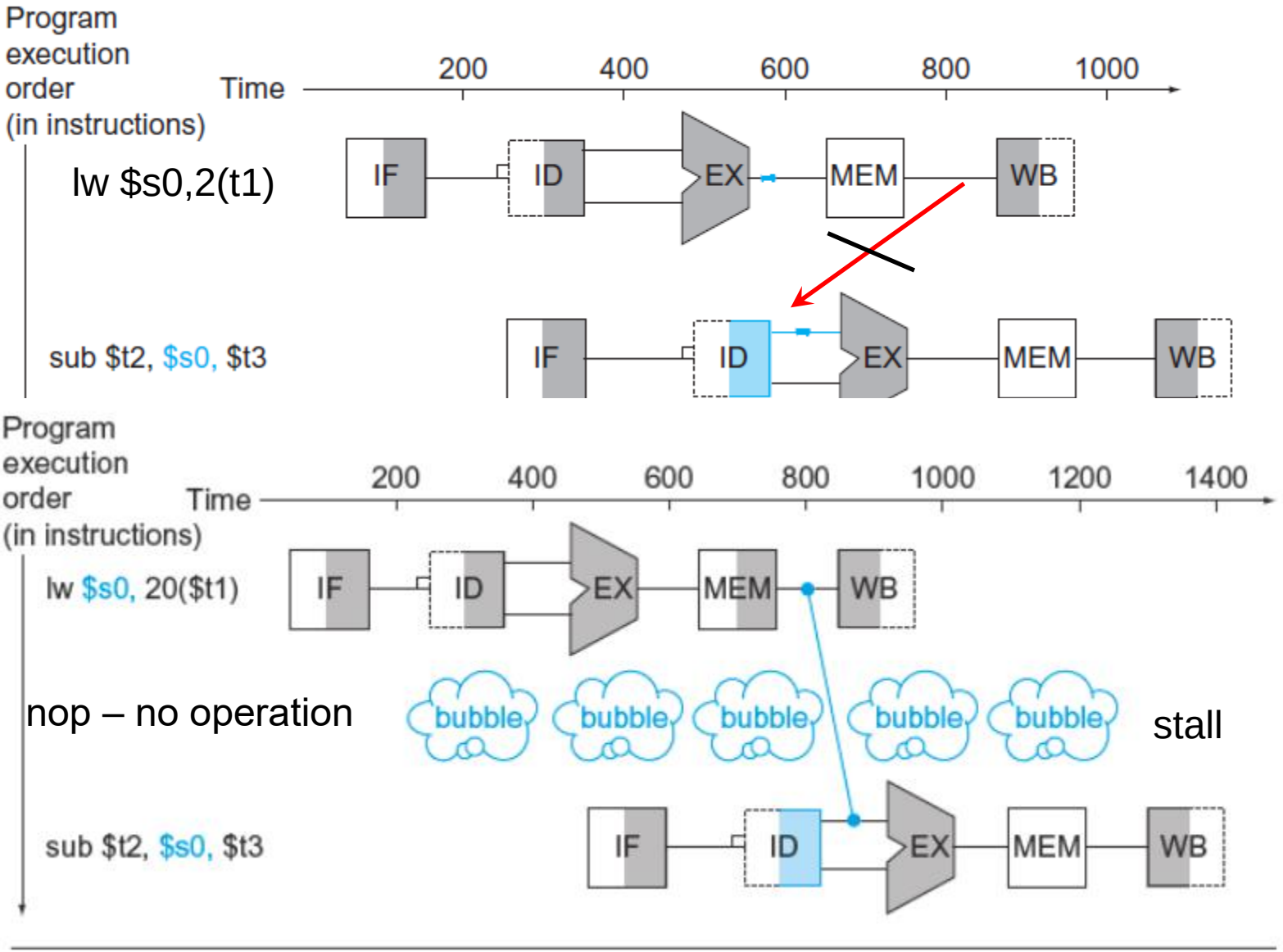
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

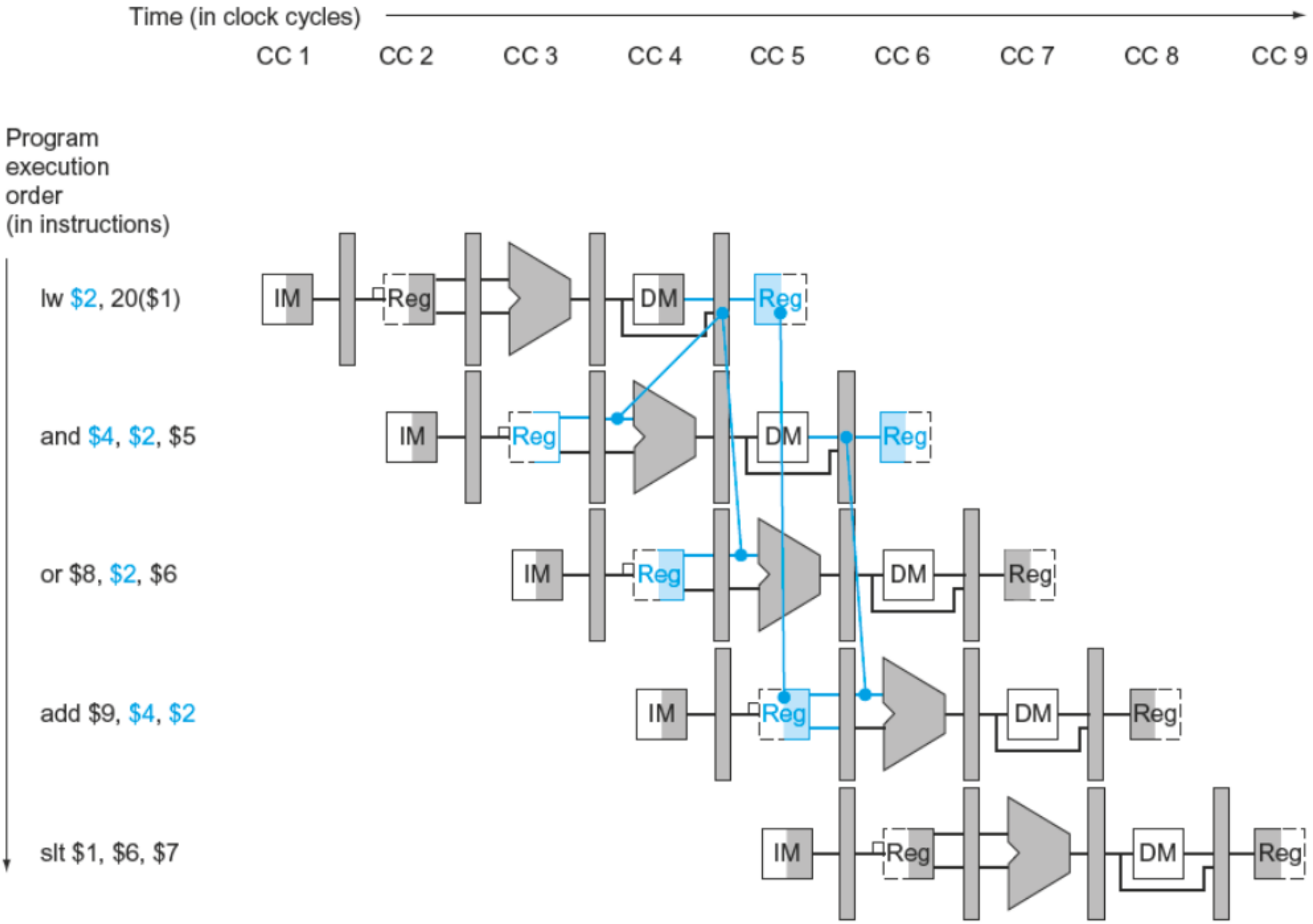
Program execution order (in instructions)

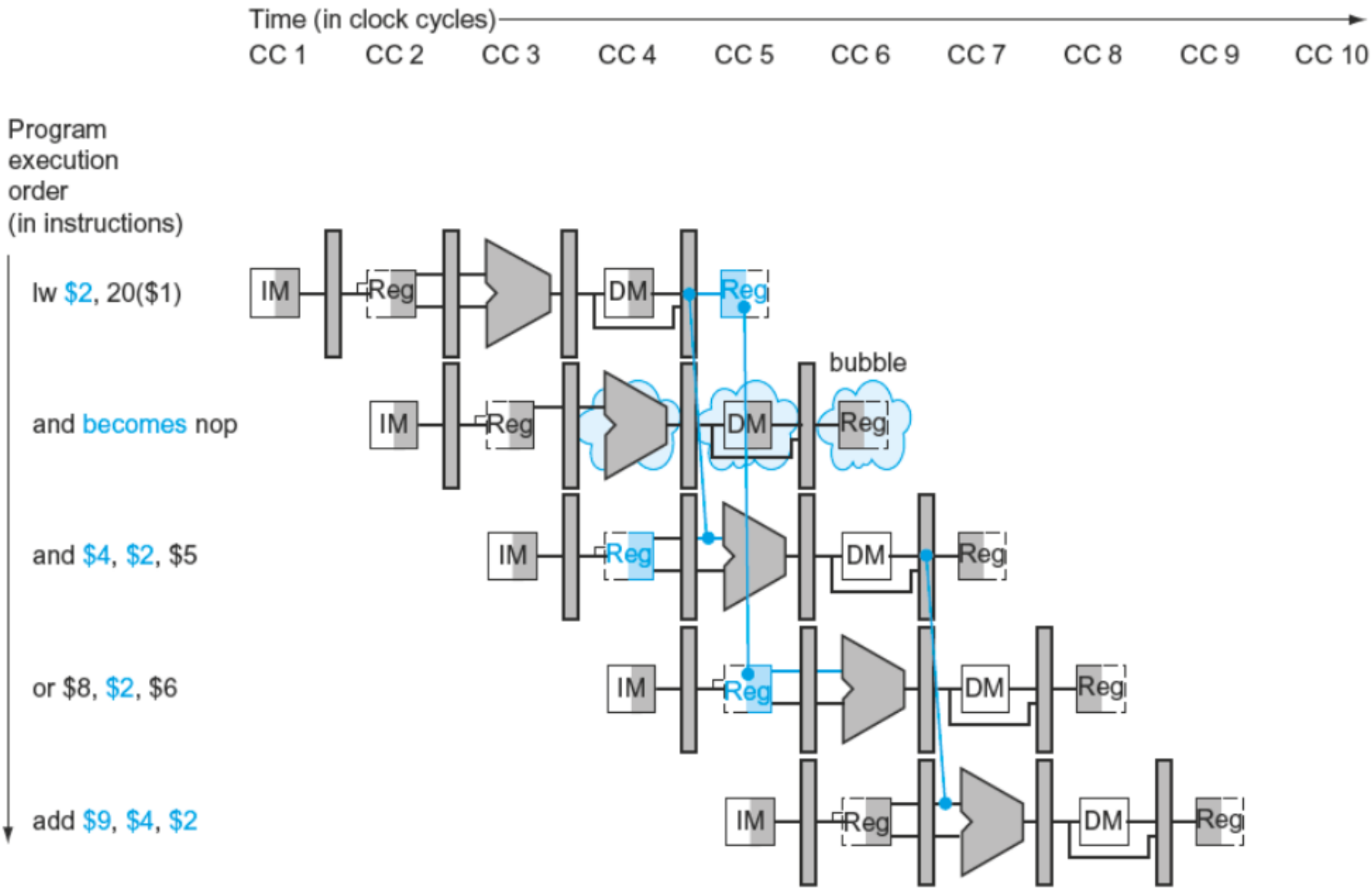


Pipelined execution – problems…

data hazards



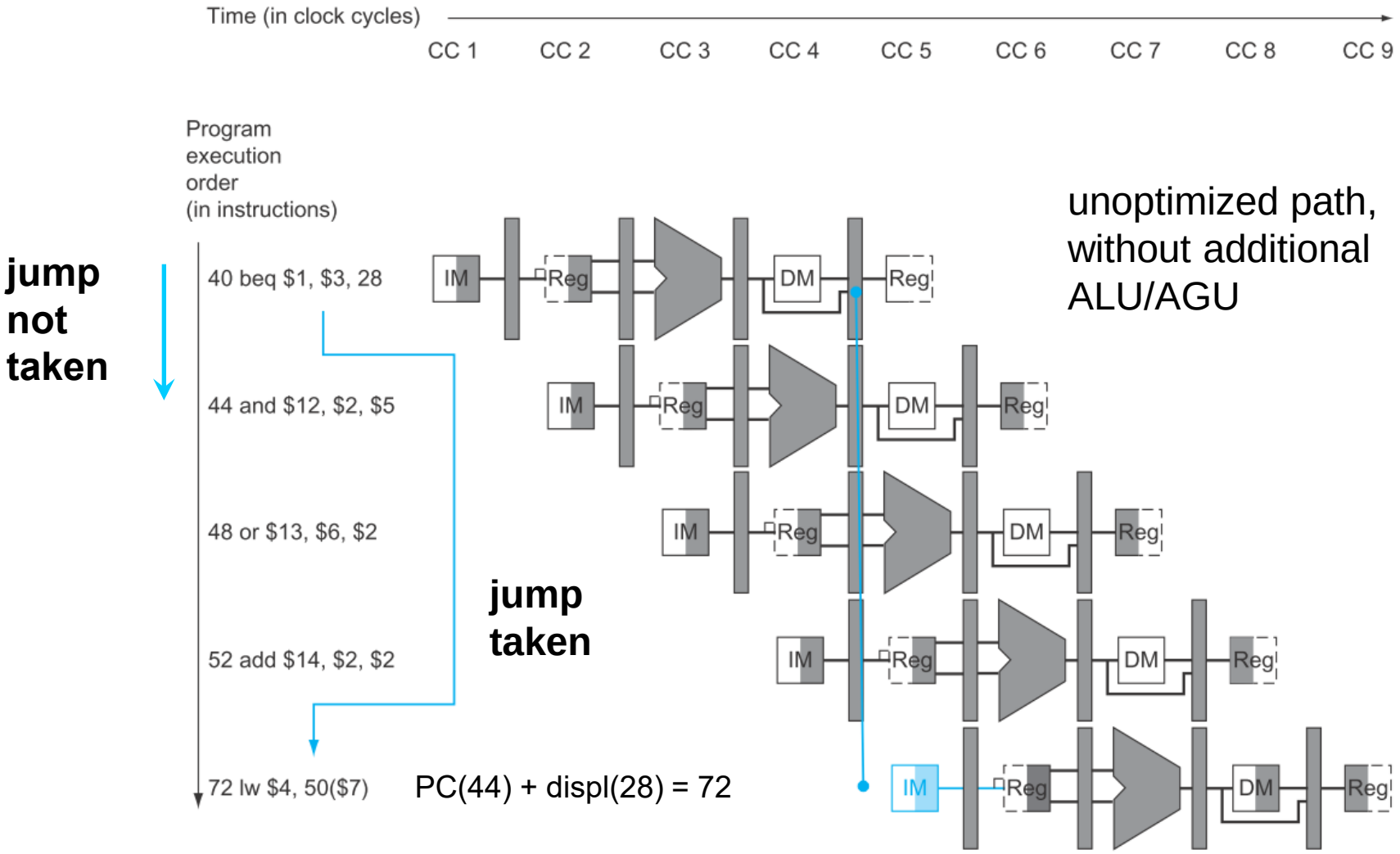

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM
Reg


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
CC 10
Program execution order (in instructions)
lw $2, 20($1)
and becomes nop
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
IM
Reg
DM
bubble

Pipelined execution – problems…

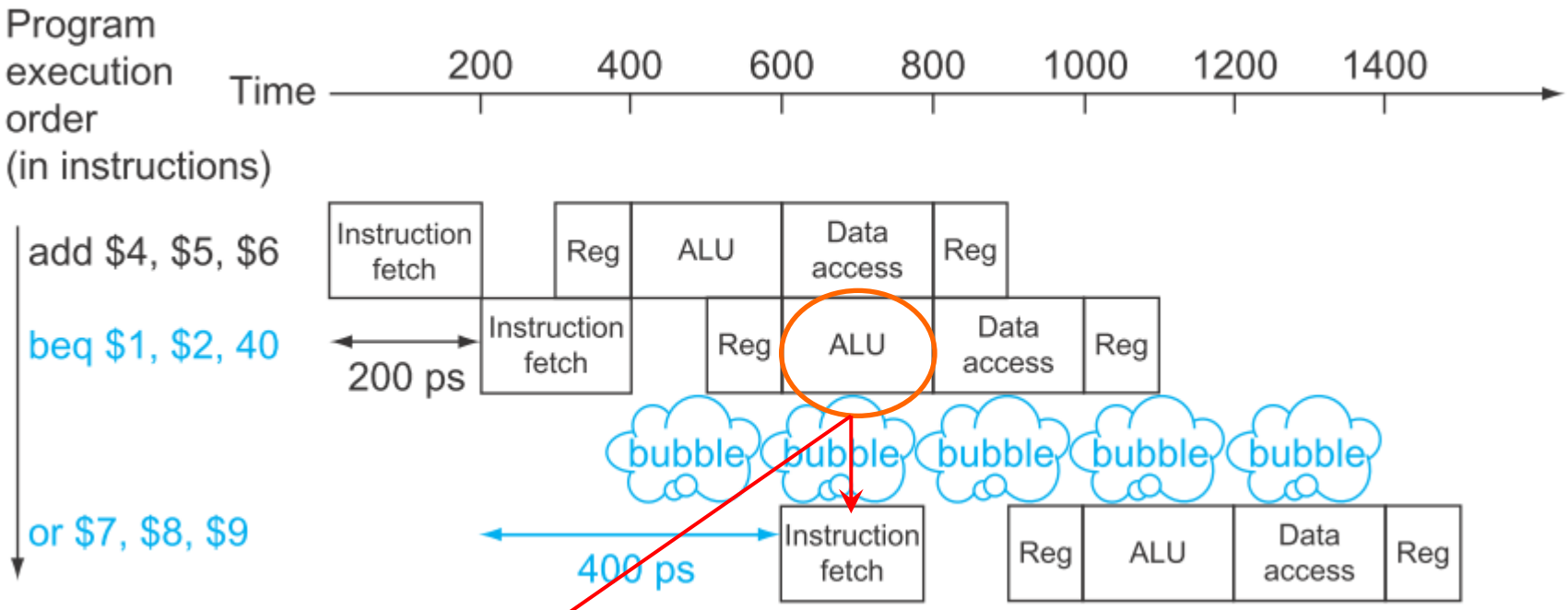
control hazards / branch hazards - CPU needs to make a decision based on the results of one instruction while others instructions are executing…

E.g. where to fetch the next instruction from?



Pipelined execution – problems…

control hazards / branch hazards



Stalling the pipeline until the branch is complete is too slow…

beq $1,$2,40 – jump to location (PC+40) if (contents of) register $1 == register $2

Optimization:
destination address of the jump is calculated simultaneously during comparison,
both tasks are performed by the dedicated ALUs. (or AGU – Address Generation Unit).
See slide #6 for details.

Pipelined execution – problems…

Speculative Execution
Simple (static) Branch Prediction

- assume that branch is not taken

- just continue execution down the sequential instruction stream

- if jump is taken - the instructions that are being fetched and decoded must be discarded (**pipeline flush/refill**) – we pay the **penalty** - extra time!

- execution continues at the target address of the branch…

Pipelined execution – problems…

Dynamic Branch Prediction

- try to predict branch result on the basis of recent behaviour (e.g. loops…)

- branch history table contains information (1-2 bits) whether the branch was recently taken, or not.

- fetching begins in the predicted direction

- pipeline needs to be flushed in the case of misprediction

Eager Execution

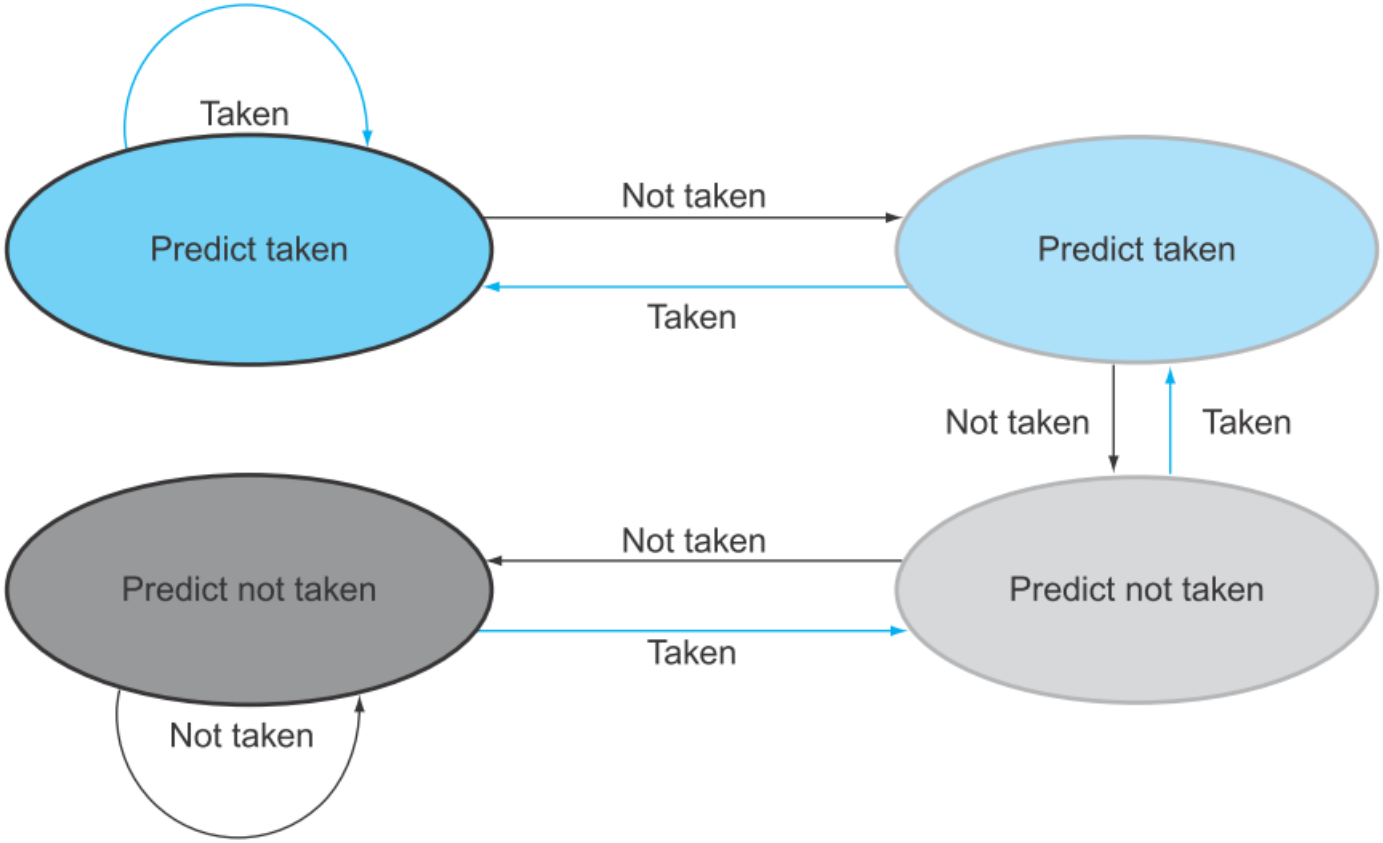
- the both sides of the branch are fetched and processed

- after evaluation of condition, one side is discarded

Pipelined execution – problems…

2 bit dynamic branch prediction scheme

```
for (x=0;x<1000000;x++)
    for (y=0;y<1000000;y++)
        A[x][y]=…
```



Pipelined execution – problems…

control hazards / branch hazards

Out Of Order (OOO) processing/execution

Just an idea*:

```
…
1 MOV   %ecx,%ebx       #an „independent” instruction
2 CMP   $10,%eax        #(result of the comparison does not depend on
3 JE    label           #on the result of MOV operation)
4 SHL   $2,%eax
…
```

```
…
1 CMP   $10,%eax
2 MOV   %ecx,%ebx
3 JE    label
4 SHL   $2,%eax
…
```

*modern processors sometimes perform so-called macro-op fusion: certain pairs of instructions (e.g. cmp & cond_jmp) are fused and handled as one operation.

Pipelined execution – problems…

control hazards / branch hazards

if (eax > 5) then eax=ebx else eax=ecx end_if

```
1          CMP     $5,%eax
2          JBE     else
3          MOV     %ebx,%eax
4          JMP     end_if
5 else:    MOV     %ecx,%eax
6 end_if: …
```

conditional execution:

```
1          CMP     $5,%eax      #set the flags
2          MOV     %ebx,%eax    #mov does not modify the flags
3.         CMOVBE  %ecx,%eax    #conditional move (eax:=ecx) if below or equal
```

+ reduces the number of conditional jumps in a program
- all instructions have to be fetched

In the case of longer blocks – much more instructions to fetch
(one part just passes through the pipeline – results are ignored)

Conditional Execution

One unusual feature of ARM core is that **every instruction** has the option of executing conditionally - depending on the condition codes (flags)

Greatest Common Divisor:

```
while (a != b) {
      if (a > b) a = a – b;
      else b = b – a;       }
```

ARM

```
gcd:
        CMP     r0, r1
        SUBCS   r0, r0, r1      ;subtract only when Carry flag is Set
        SUBCC   r1, r1, r0      ;subtract only when Carry flag is Cleared
        BNE     gcd
```

THUMB2 (ARM-Cortex)

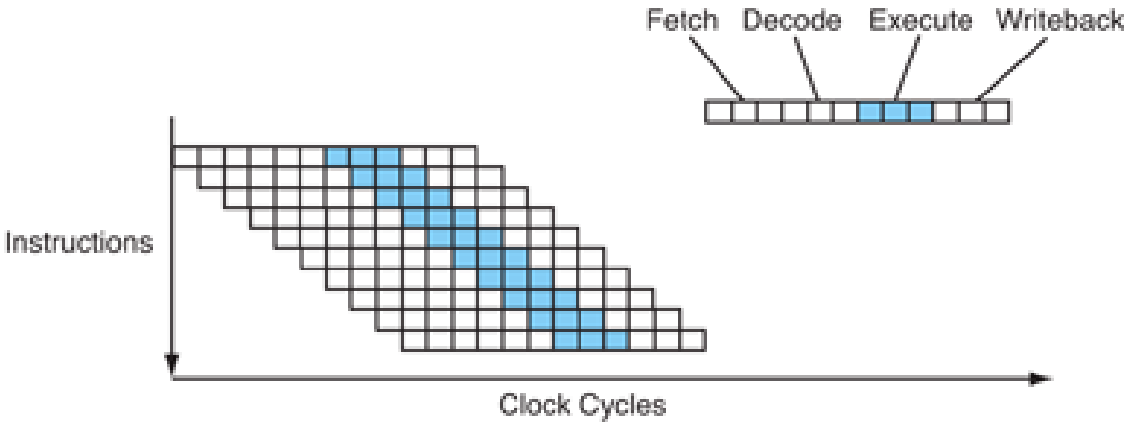
```
gcd:
        CMP     r0,r1
        ITE     CS              ;IF THEN ELSE
        SUBCS   r0,r0,r1        ;one instruction in THEN section
        SUBCC   r1,r1,r0        ;one in ELSE
        BNE     gcd
```

„Standard” pipelined execution:

Performance:
• ideal/theoretical: 1 CPI (clocks per instruction)
• practical 1.2 CPI (or 0.83 IPC instructions per clock)

max. clocking frequency is limited by the longest stage (phase, operation) in the pipeline

Superpipelined processor - pipeline is divided into large number of short, simple stages



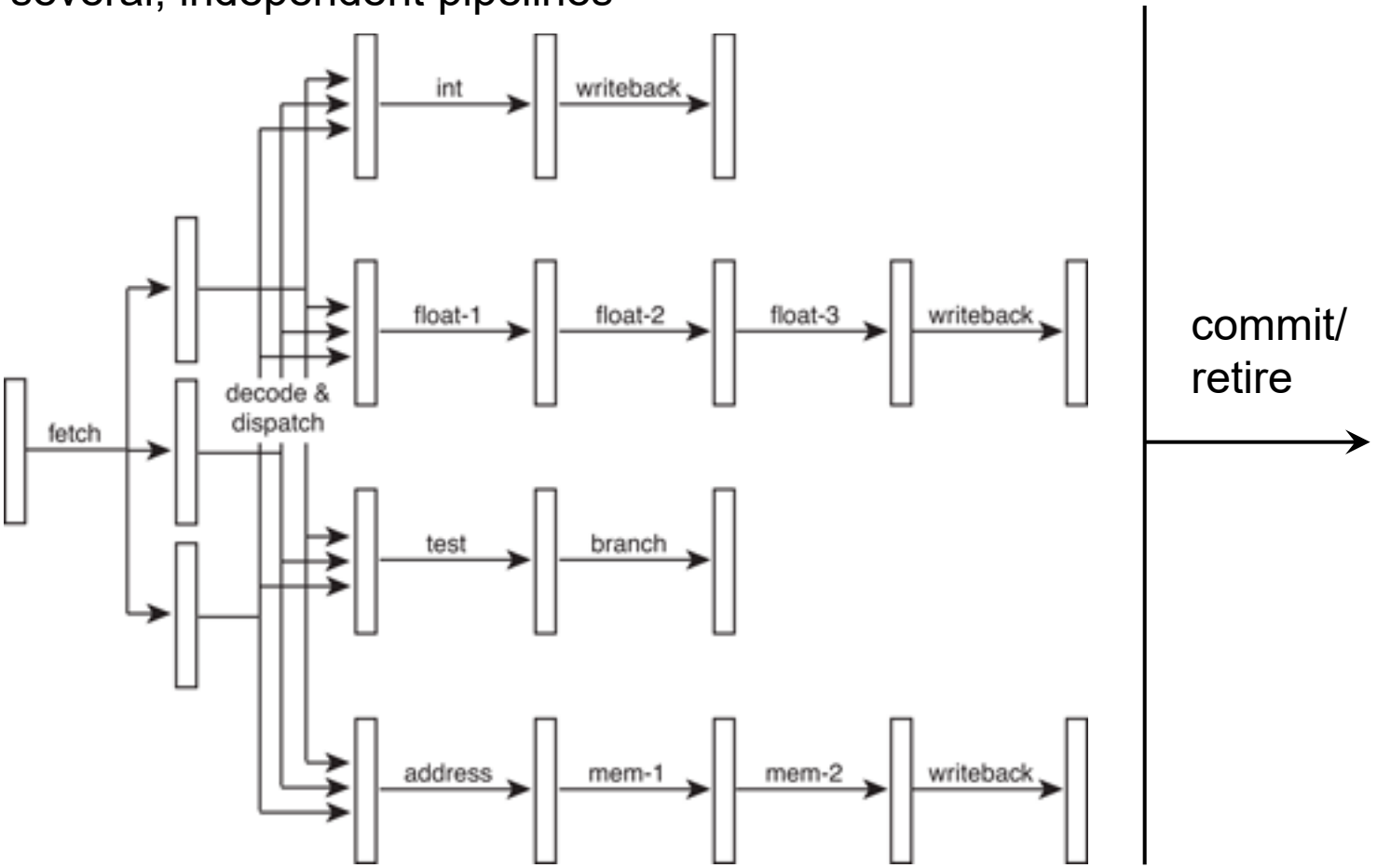
• execution of the single instruction requires more clock cycles
• still 1 CPI (in practice 1.5 CPI)
• shorter stages – **higher maximal clock frequency**
• theoretically **better performance** (e.g. instructions per second)

• longer pipeline = higher penalties e.g. due to branch misprediction…

Superscalar processors – Dynamic Multiple Issue

try to **execute more than one instruction at one clock cycle…**

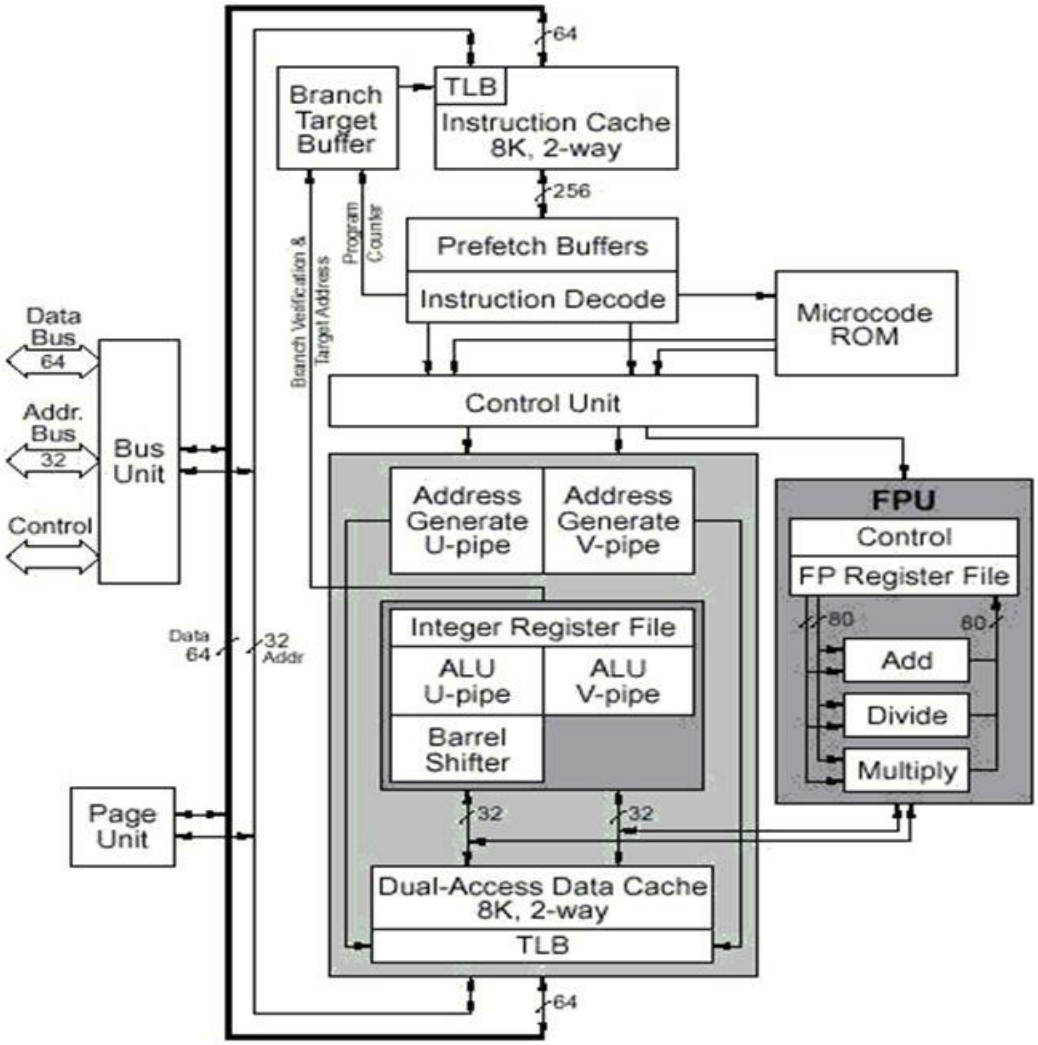
- complex pipeline with multiple, concurrent execution units
- sometimes several, independent pipelines



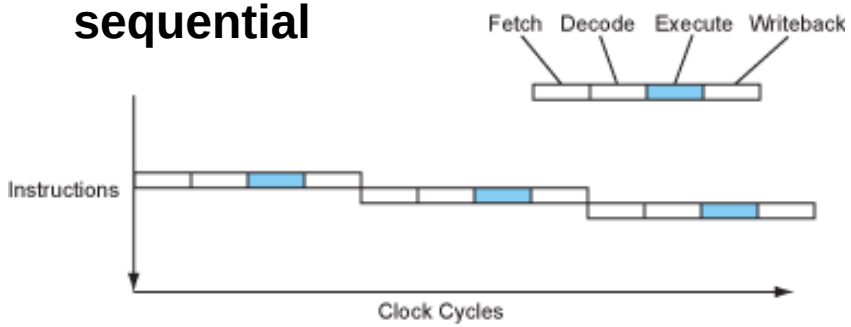
- fetch and decode a packet of instructions
- check the dependencies between them, sort, change order and dispatch to exec. units
- ensure that results are written to memory in the original order

Pentium 1 - microarchitecture

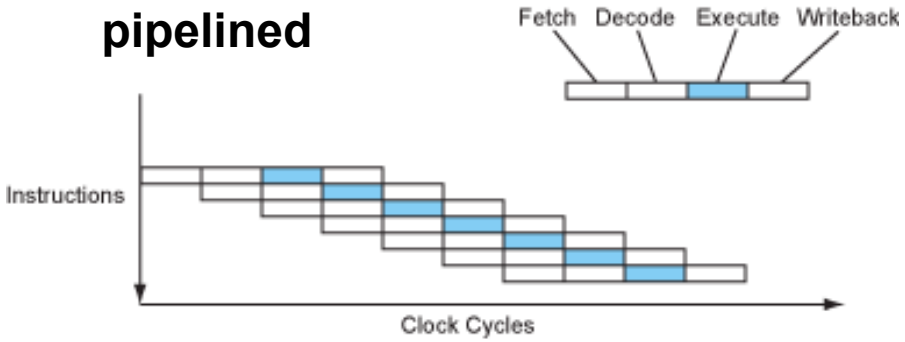
x86-compatible superscalar:
two „pipes”: U – all instructions, V – only certain, simple instructions

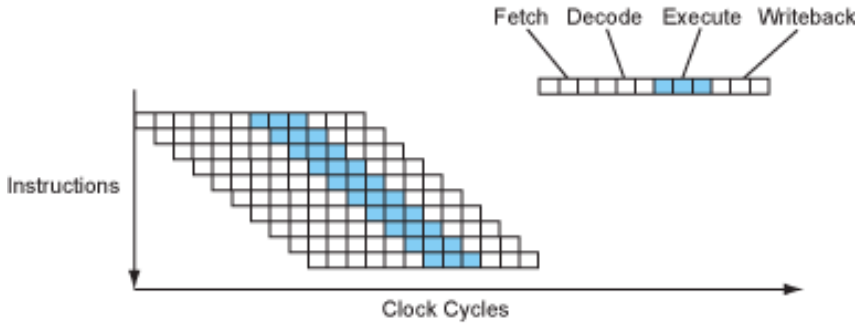


sequential

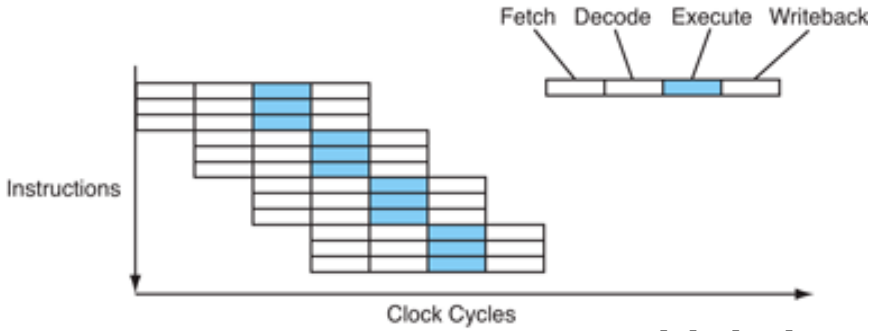


pipelined

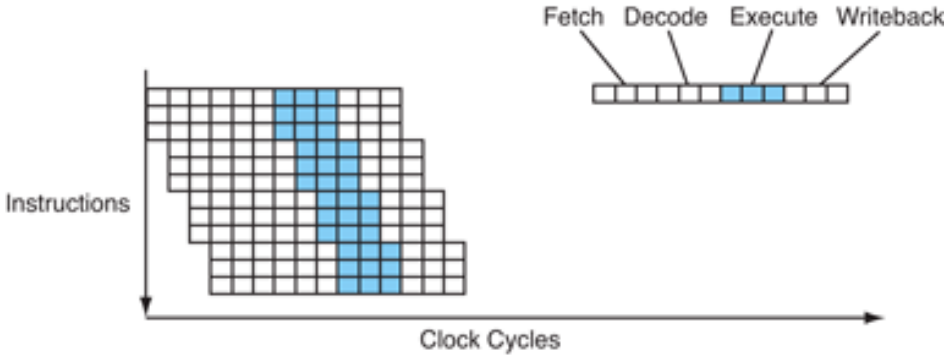




superpipelined



multiple issue superscalar



superpipelined superscalar

most modern general-purpose CPUs

images from: www.lighterra.com/papers/modernmicroprocessors

Multiple-Issue processors

Static Multiple Issue

- compiler assists with matching and packaging instructions into the groups (issue packets) to be executed in parallel

- compiler handles the hazards (e.g. data dependencies)

- compiler can change the order of instructions (can even change the original code!)

- simpler logic design:
  CPU does not contain complex out of order execution and dependency-checking logic circuits

Static Multiple Issue approach is not popular today…

VLIW processors (Very Long Instruction Word)
EPIC (Explicitly Parallel Instruction Computing) - HP & Intel
Intel Itanium

some Digital Signal Processors (DSP), GPU shaders…

Static Multiple Issue - example

issue width = 2, two pipelines

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static Multiple Issue – example

Add a constant value (32 bit, stored in reg. $s2) to each element of the table.
Pointer (address) of the last element is in reg. $s1.

```
for_loop:
1       lw      $t0, 0($s1)          ;load element
2       addu    $t0,$t0,$s2          ;modify: $t0 += $s2
3       sw      $t0,0($s1)           ;write-back
4       addi    $s1,$s1,-4           ;decrement pointer
5       bne     $s1,$zero,for_loop ;repeat until we process the first element (0)
```

- **Read after Write (RaW) data dependencies** (blue, red…)

- **data dependency can be loop-carried…**

Original code:

```
for_loop:
1   lw    $t0, 0($s1)          ;load element
2   addu  $t0,$t0,$s2          ;modify: $t0 += $s2
3   sw    $t0,0($s1)           ;write-back
4   addi  $s1,$s1,-4           ;decrement pointer
5   bne   $s1,$zero,for_loop   ;repeat until we process the first element (0)
```

CPU can perform two independent operations in parallel: data transfer & ALU operation

```
    ALU                        LOAD/STORE
forloop:
1   nop                        lw      $t0, 0($s1)
2   addi  $s1,$s1,-4           nop
3   addu  $t0,$t0,$s2          nop
4   bne   $s1,$zero,forloop    sw      $t0,4($s1)
```

nop - no operation

Very poor result: only one pair of instructions was executed in parallel way…

CPI (clocks per instruction) = 4/5 = 0.8 (the best theoretical result = 0.5)

IPC (instructions per clock) = 5/4 = 1.25 (the best theoretical result = 2.0)

```
for_loop:
1   lw    $t0, 0($s1)           ;load element
2   addu  $t0,$t0,$s2           ;modify: $t0 += $s2
3   sw    $t0,0($s1)            ;write-back
4   addi  $s1,$s1,-4            ;decrement pointer
5   bne   $s1,$zero,for_loop    ;repeat until we process the first element (0)
```

Loop Unrolling! (mod 4)

```
forloop:
1   addi  $s1,$s1,-16
2                               lw    $t0,0($s1)
3                               lw    $t1,4($s1)
4   addu  $t0,$t0,$s2           lw    $t2,8($s1)
5   addu  $t1,$t1,$s2           lw    $t3,12($s1)
6   addu  $t2,$t2,$s2           sw    $t0,0($s1)
7   addu  $t3,$t3,$s2           sw    $t1,4($s1)
8                               sw    $t2,8($s1)
9   bne   $s1,$zero,forloop     sw    $t3,12($s1)
```

10 instructions from 14 were processed in parallel (9 clocks)
CPI = 9/14 = 0.64 ; IPC = 14/9 = 1.56
additionally: 4x less number of iterations (conditional branches)

Dynamic Multiple Issue – Superscalar processors

- **CPU chooses which (and how many) instructions to execute in parallel**, in a given clock cycle

- **CPU can dynamically reorder instructions to reduce the stalls**

- CPU have to analyze dependencies in a group of instructions to avoid hazards!

- **A good compiler can optimize the code**:
  e.g. try to schedule instructions to move the dependencies apart

  additionally – compiler has access to whole code… and a lot of time…
  but can not predict everything: exceptions, i/o handling, cache misses

Finally: all the results must be written back to memory in the order of the original program!

Dynamic Multiple Issue – Superscalar processors

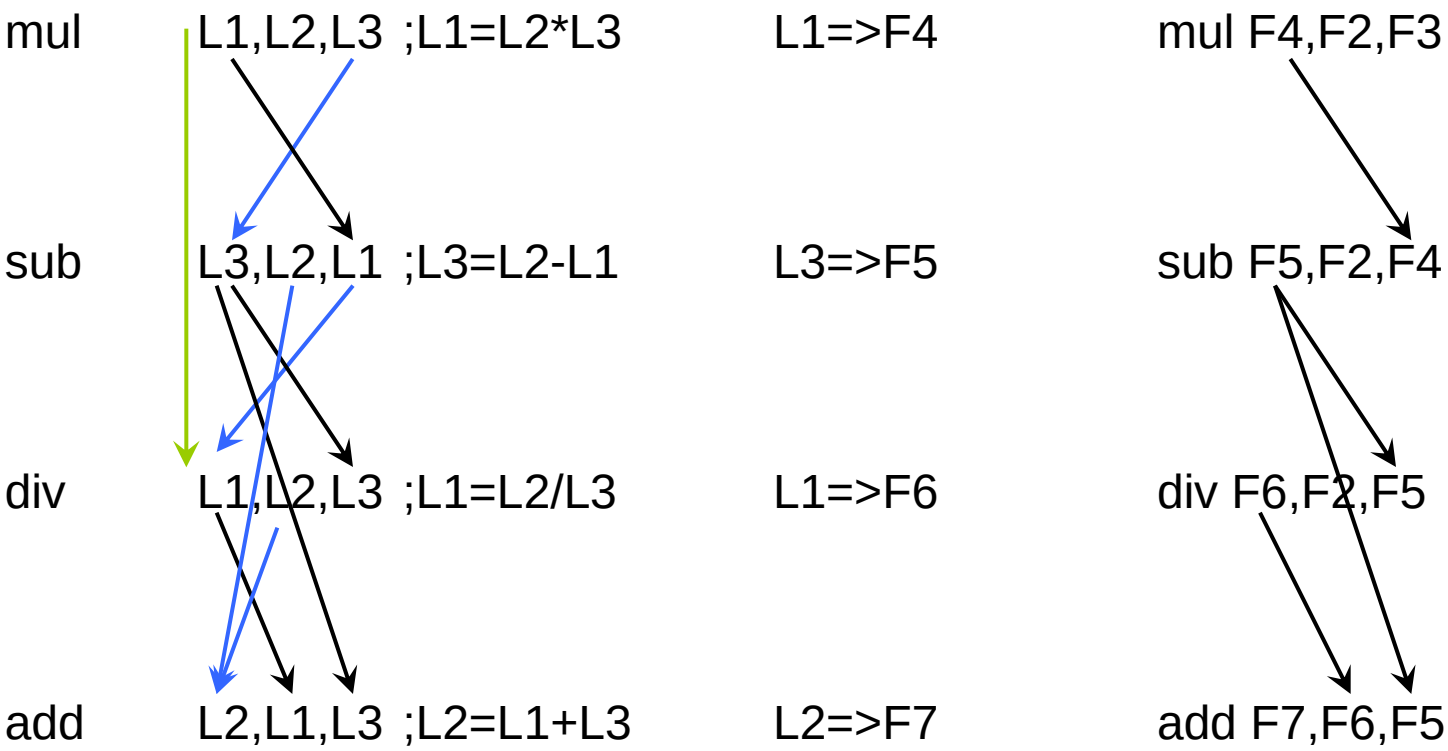
- **Out Of Order execution (OOO)**
  - VERY complex logic design, sometimes high power consumption…
- **Register renaming**
  - remove the Write after Read and Write after Write dependencies…
- **Branch prediction**
- **Conditional execution**

^^^ Just to achieve a better instruction level parallelism and to avoid stalls in the pipelines…

Again: all the results must be written back to memory in the order of the original program!

Register renaming: Lx – logical registers Fx – large set of physical registers

e.g. L1=F1, L2=F2, L3=F3, F4…F32 – unused registers



Data dependencies (only some are shown)

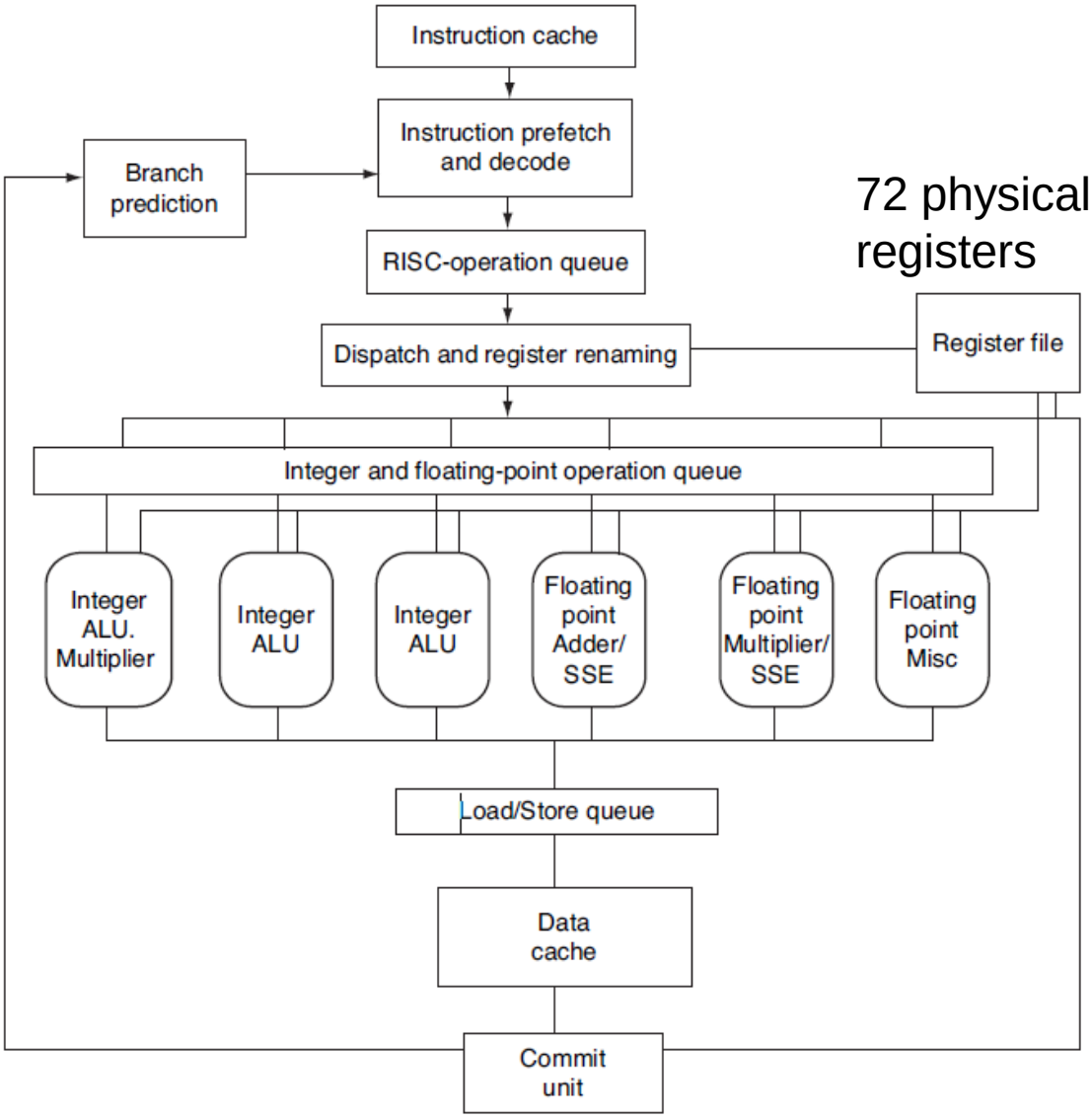
RaW Read after write – true/flow dependency

WaR Write after Read – anti dependency

WaW Write after Write – output dependency

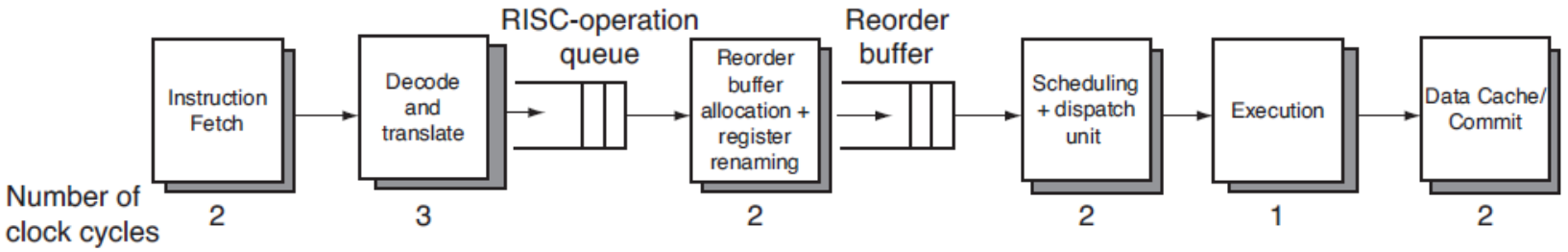
} removed by register renaming

Microarchitecture
AMD Opteron X4



72 physical registers

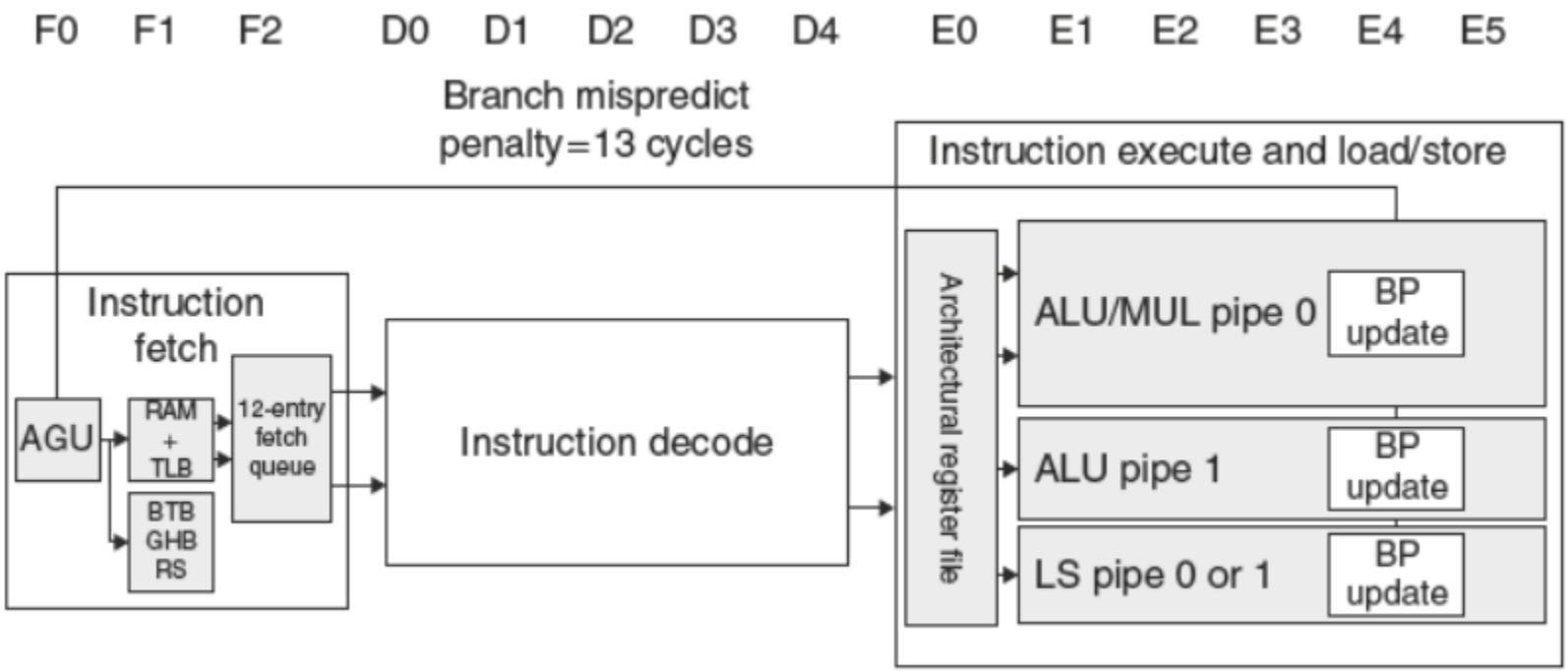
Modern x86 compatible CPUs translate legacy CISC instructions into the sequences of the simple RISC-like microoperations



Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	–	2 - 8 MiB

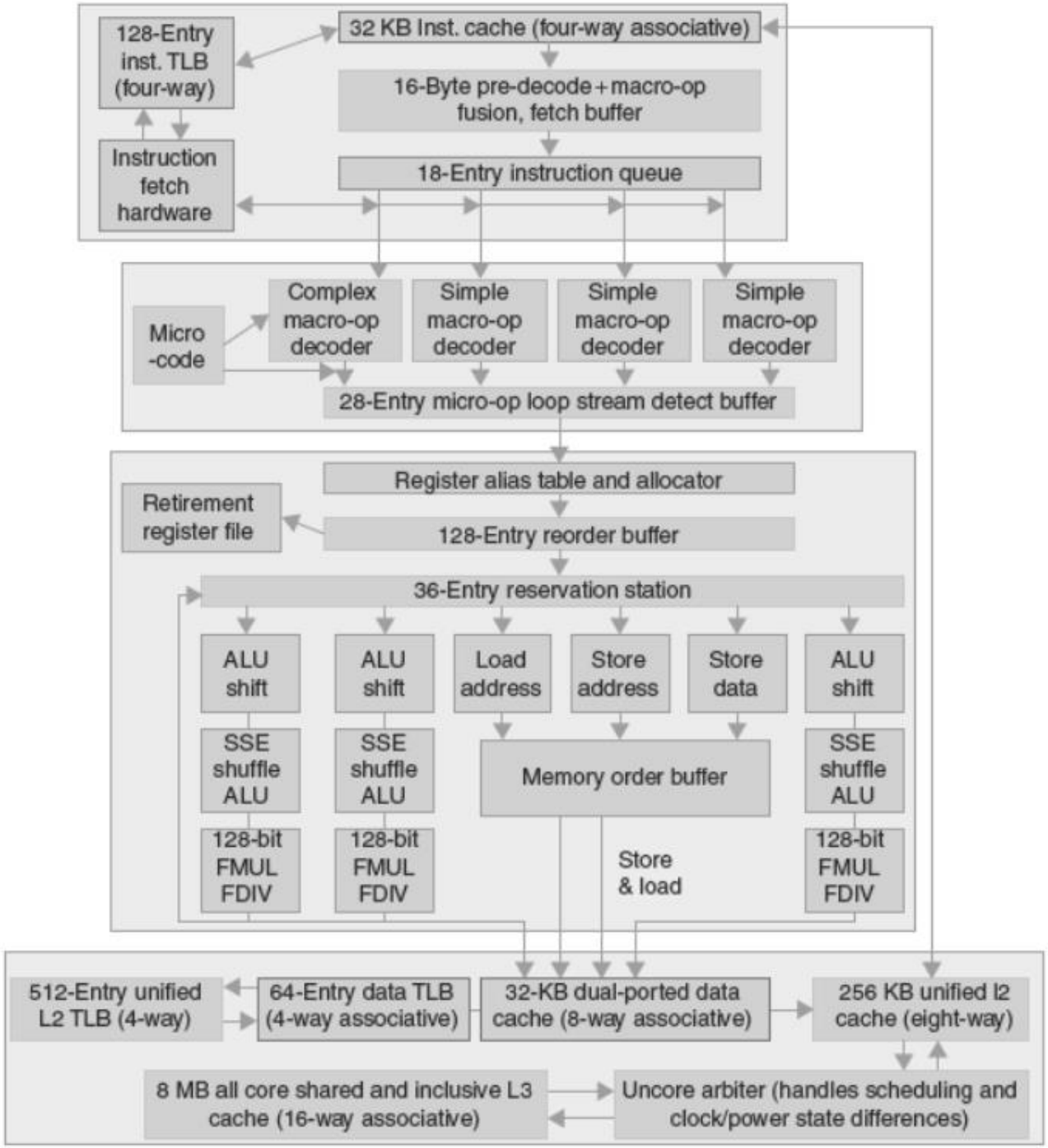
Microarchitecture: **ARM A8** core (portable devices)



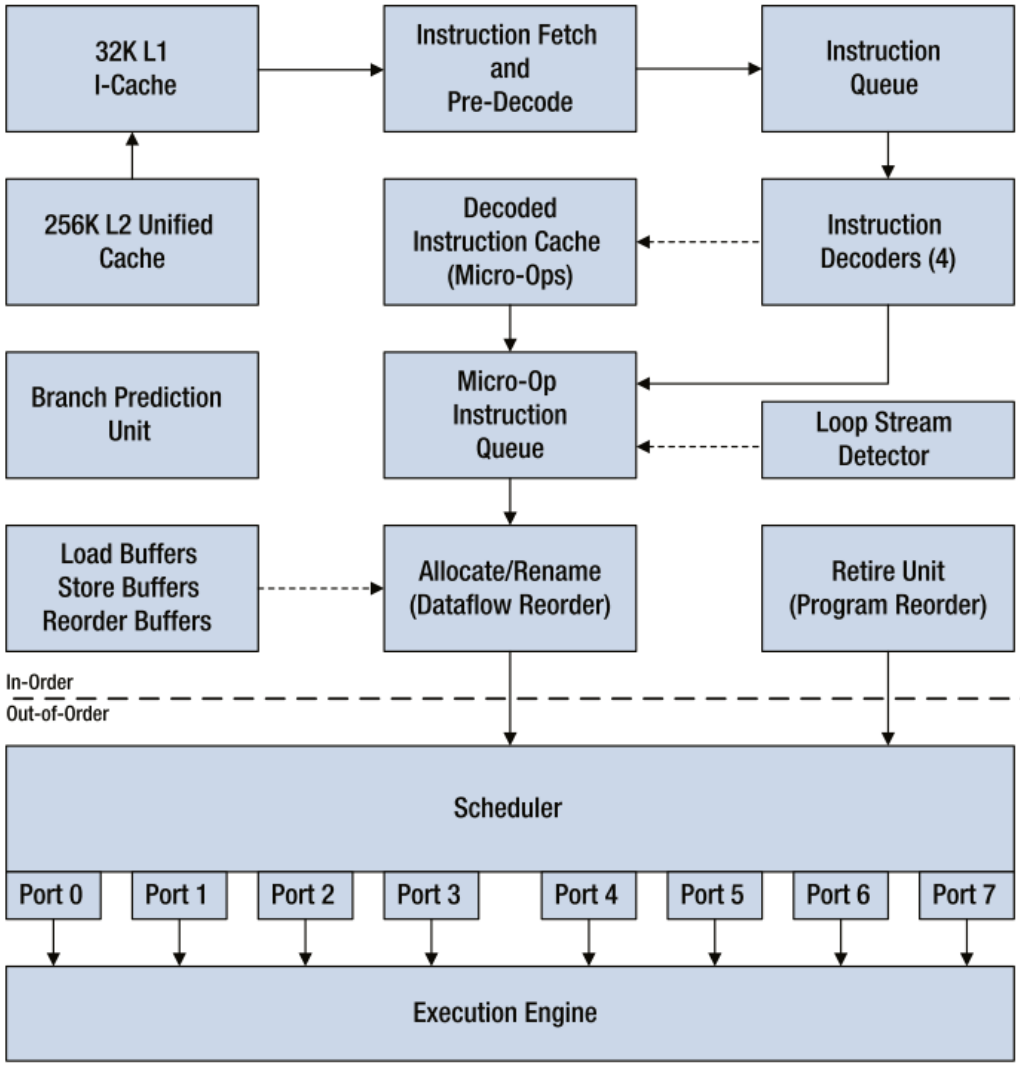
Microarchitecture

Intel i7 920

six pors /functional units
issue width=4



Microarchitecture Intel Haswell



micro fusion:

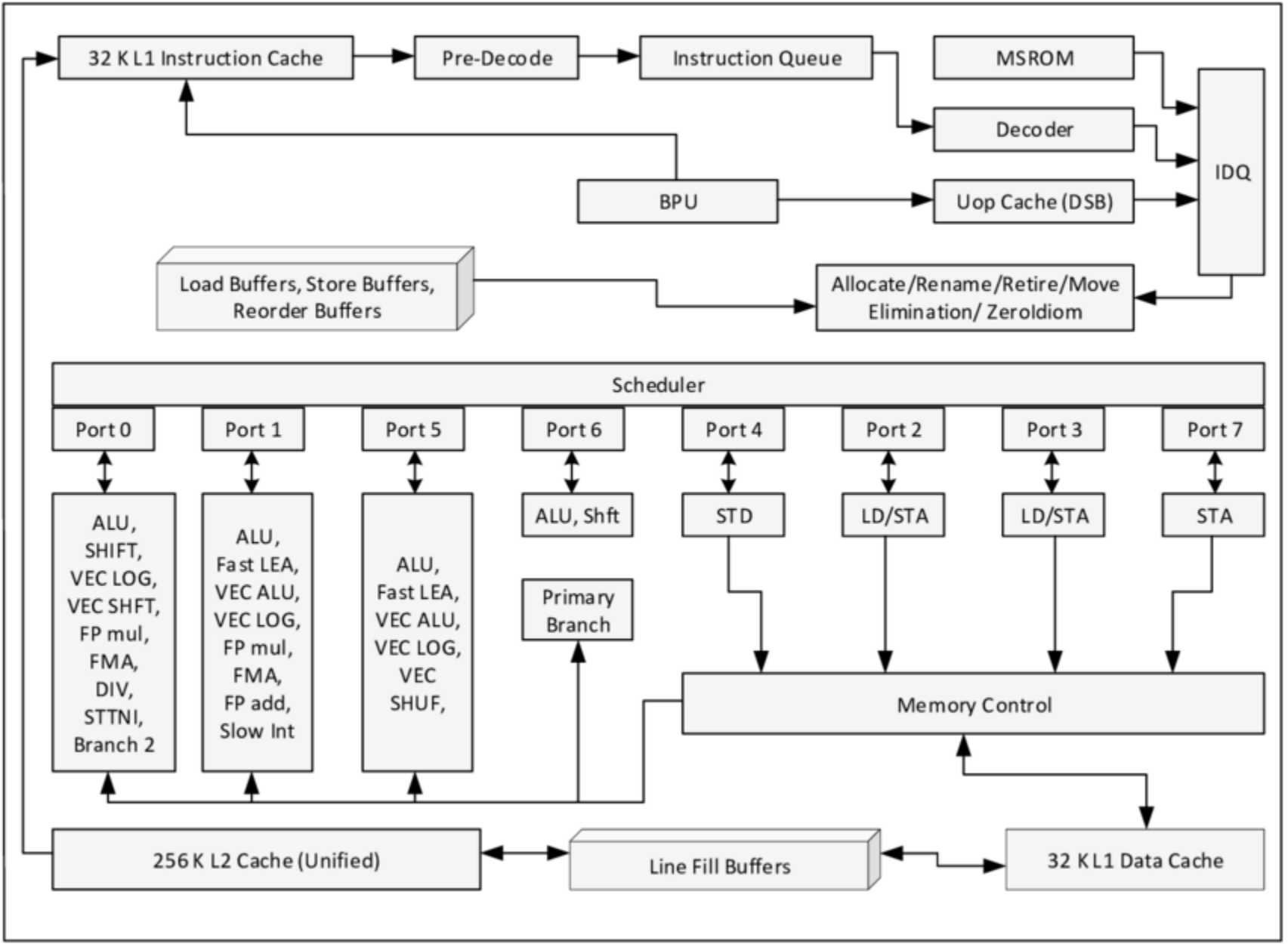
mov %eax,table(,%edi,4)

add mem,%eax

macro fusion:

dec %ecx
jnz _label

Microarchitecture Intel Haswell



in order

out of order

instruction lists of the x86-compatible (and some newer) CPUs:

http://www.agner.org/optimize/instruction_tables.pdf