Chapter 4 – part III

Advanced Pipelining Issues

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Chiao Tung Univ.

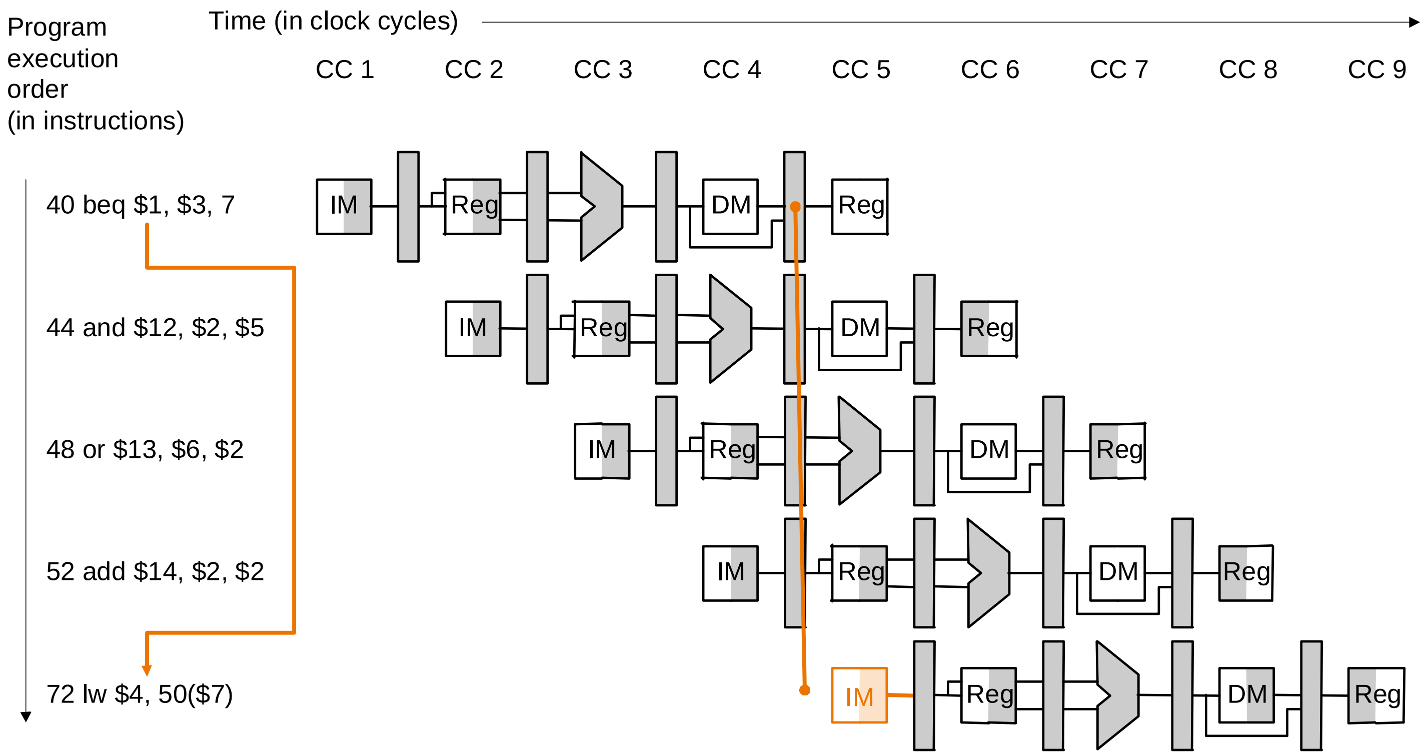
Material source:
RISC-V slides

Deal with Pipelining

- Advanced issues pipelining
  - Instructions may have dependency
  - Dependency may result into hazards
  - More instruction-level parallelism:
    - Pipelining
    - Multiple issues
    - Multithreading
- What makes pipelining hard?
  - structural hazards: resource conflicts, e.g. only one memory
    - sol: adding more resource
  - control hazards: need to worry about branch instructions
    - sol: resolve earlier, predict branch
  - data hazards: an instruction depends on a previous instruction
    - sol: forwarding, bypassing

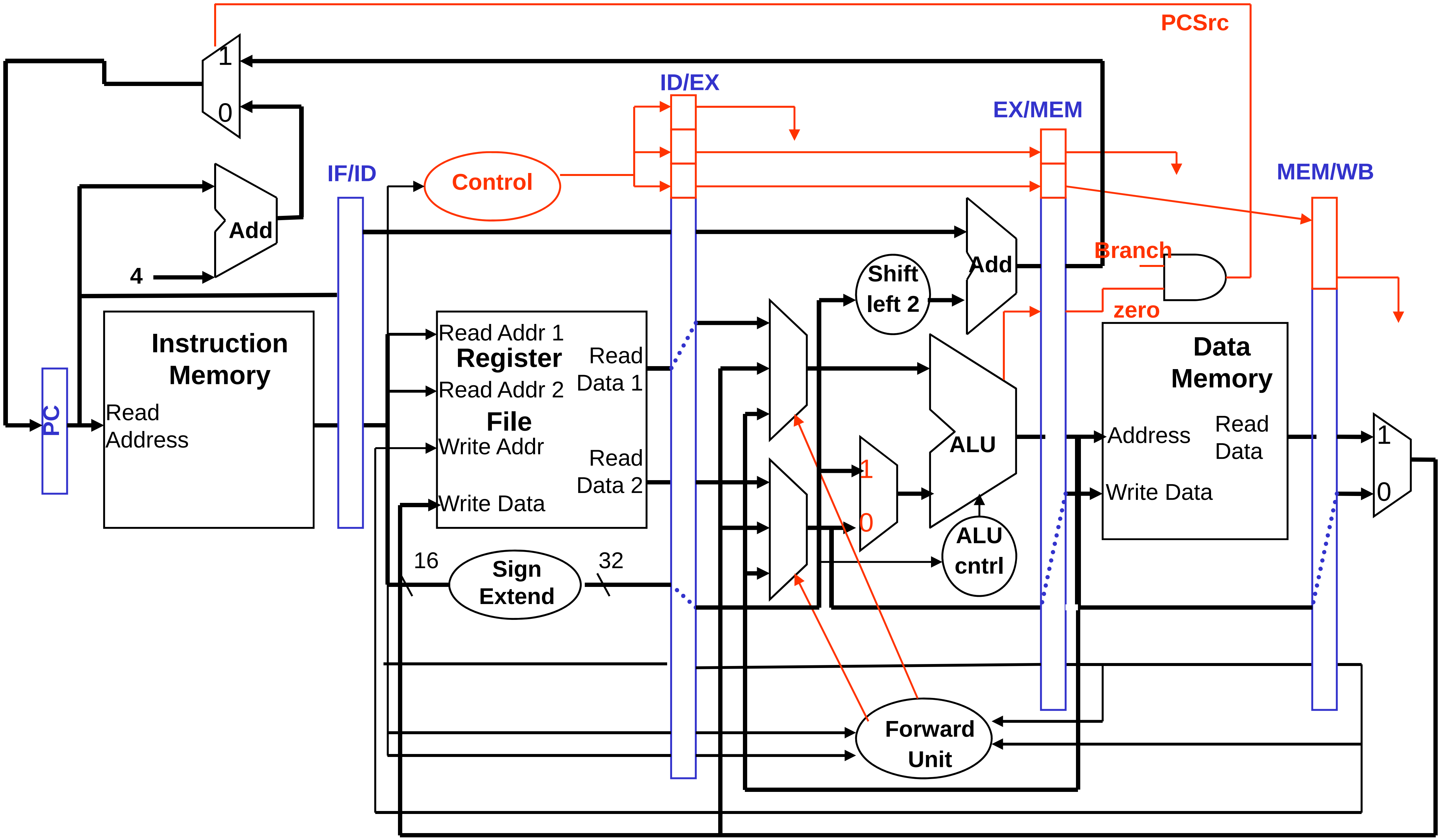
Branch Hazards

- 3 instructions after branch will be incorrectly executed



- Solutions:
  - Stall
  - Reducing branch delay by early determination of branch
  - Instruction scheduling for branch hazard
  - Branch condition prediction
    - 1-b buffer
    - 2-b buffer

3 cycles delay on a branch hazard

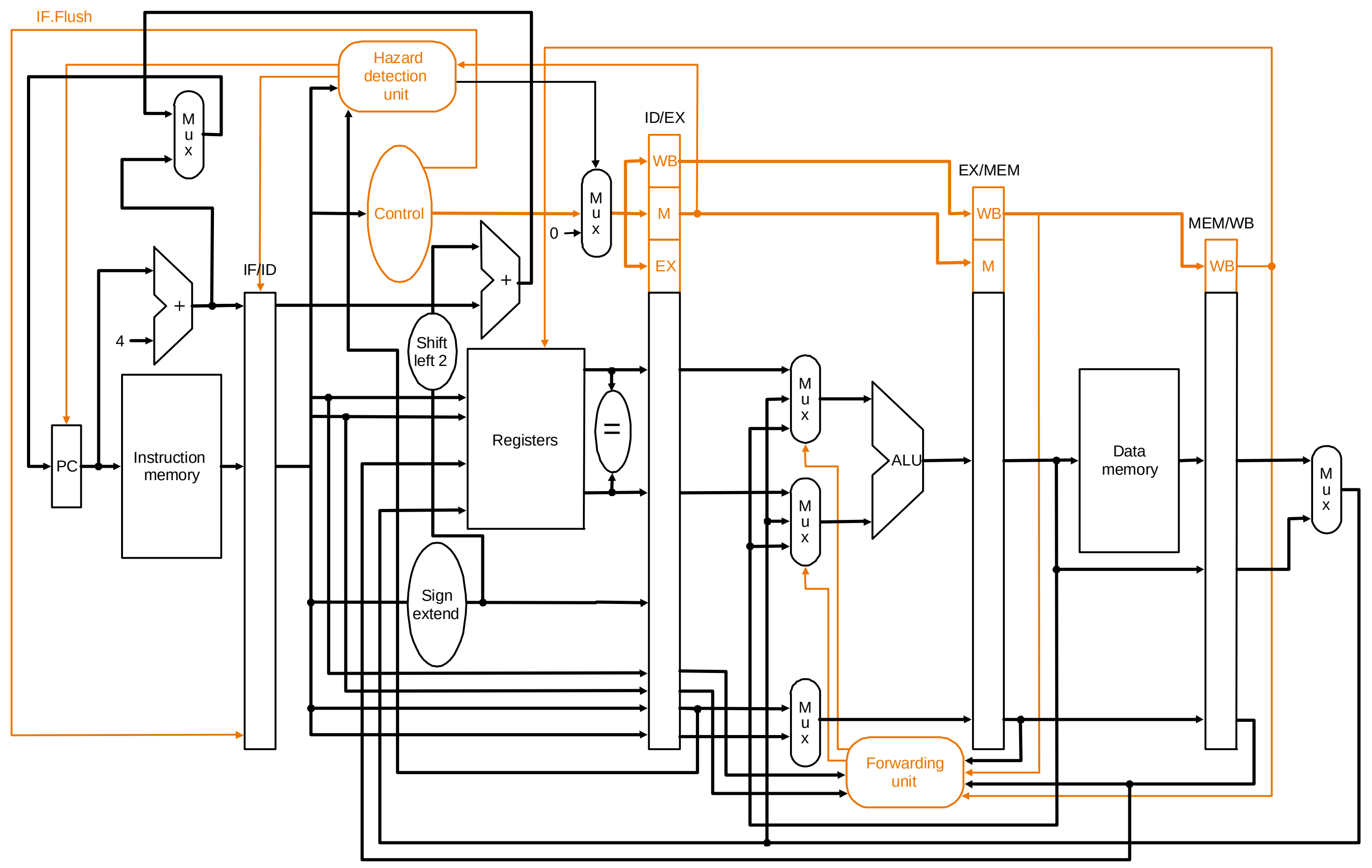


Moving Branch Decisions Earlier in Pipe

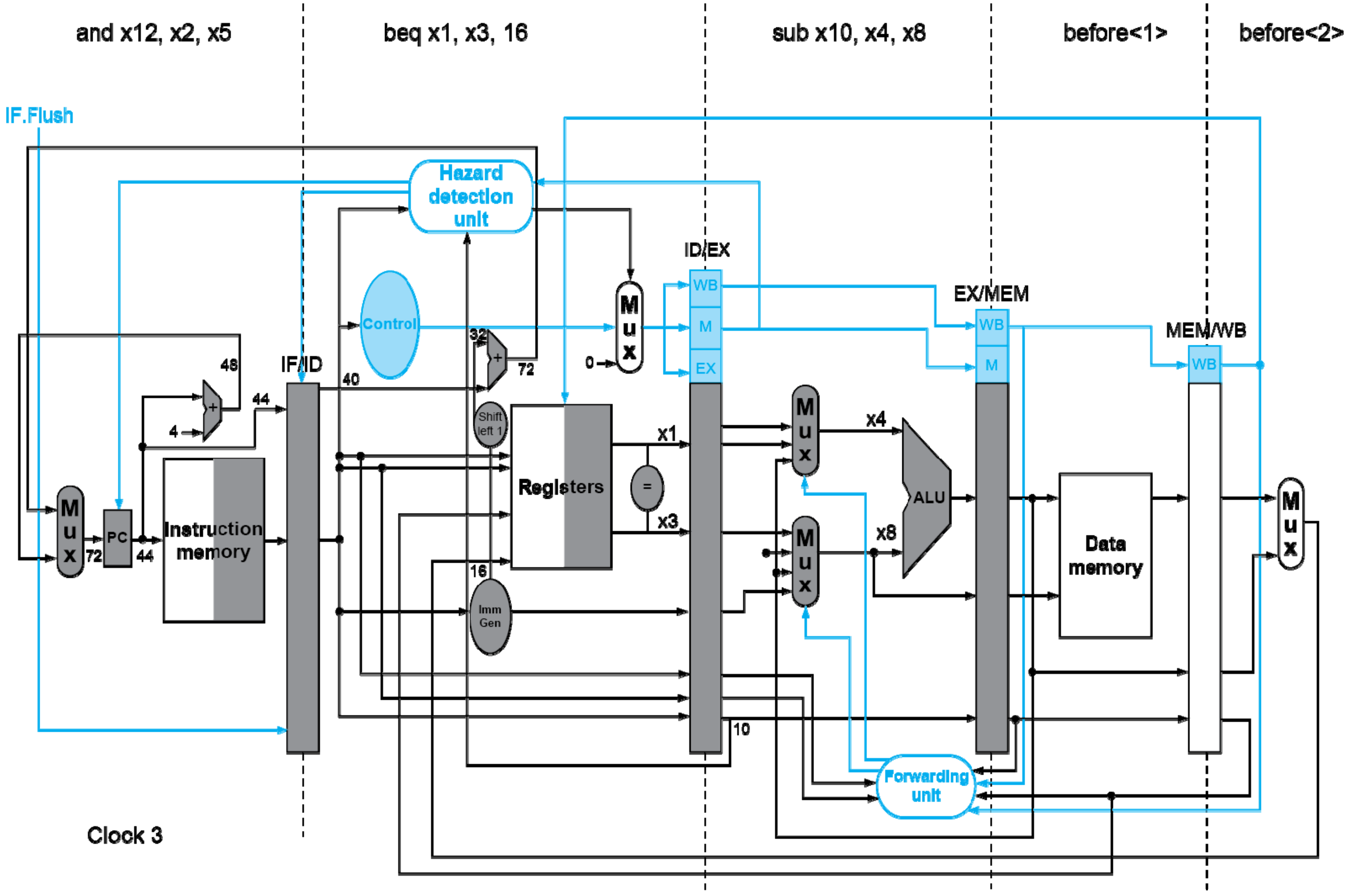
- Two key decisions:
  - Target address adder
  - Register comparator
- Move to the EX stage
  - Reduces the number of stall cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Move to the ID stage
  - Reduces the number of stall cycles to one (like with jumps)
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a comparator, an and gate, and a 3x1 mux to the ID timing path
  - Need forwarding hardware in ID stage

Flush instructions follows branch

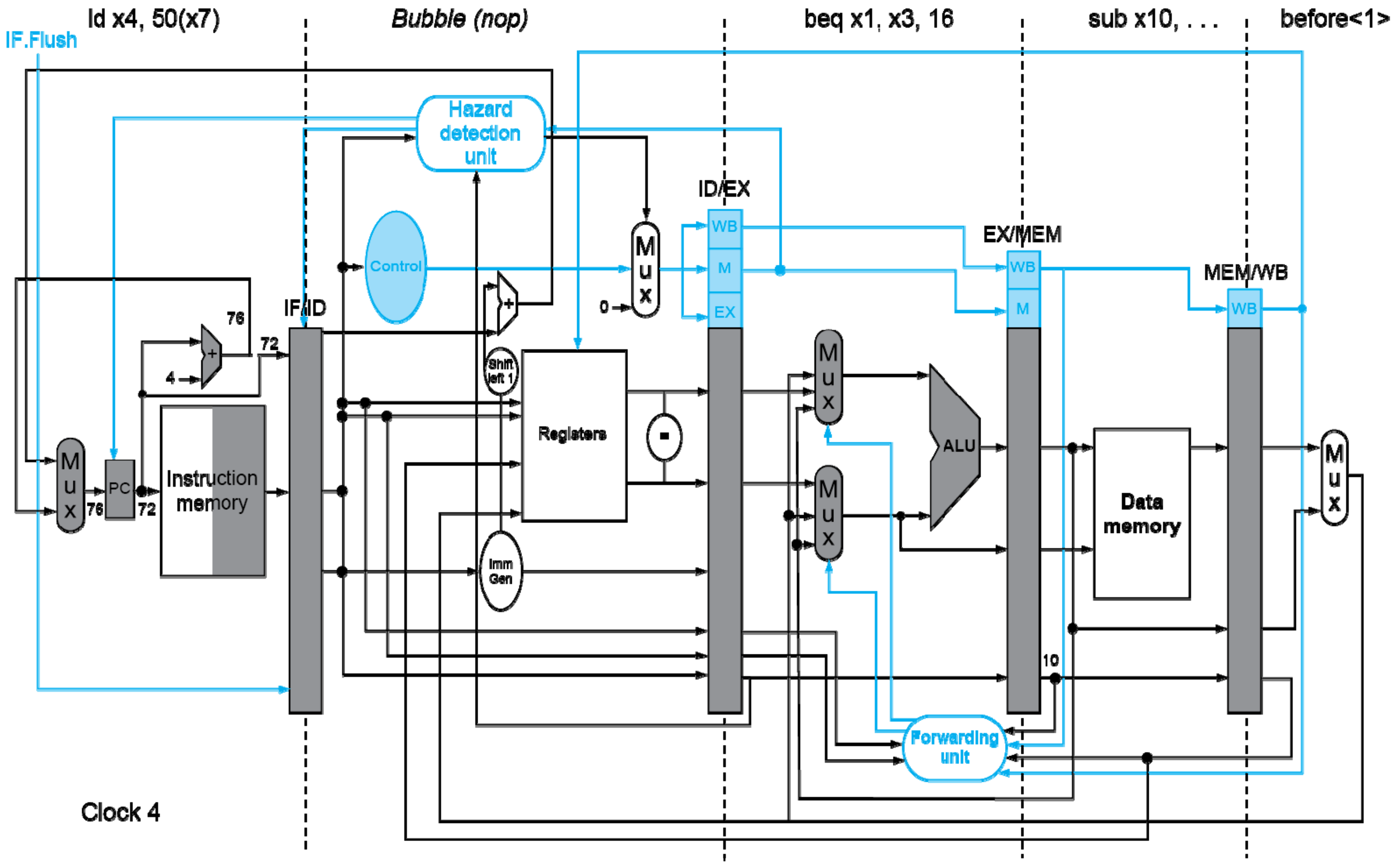
- Move branch decision from Mem to ID
- Zeros the instruction for next cycle



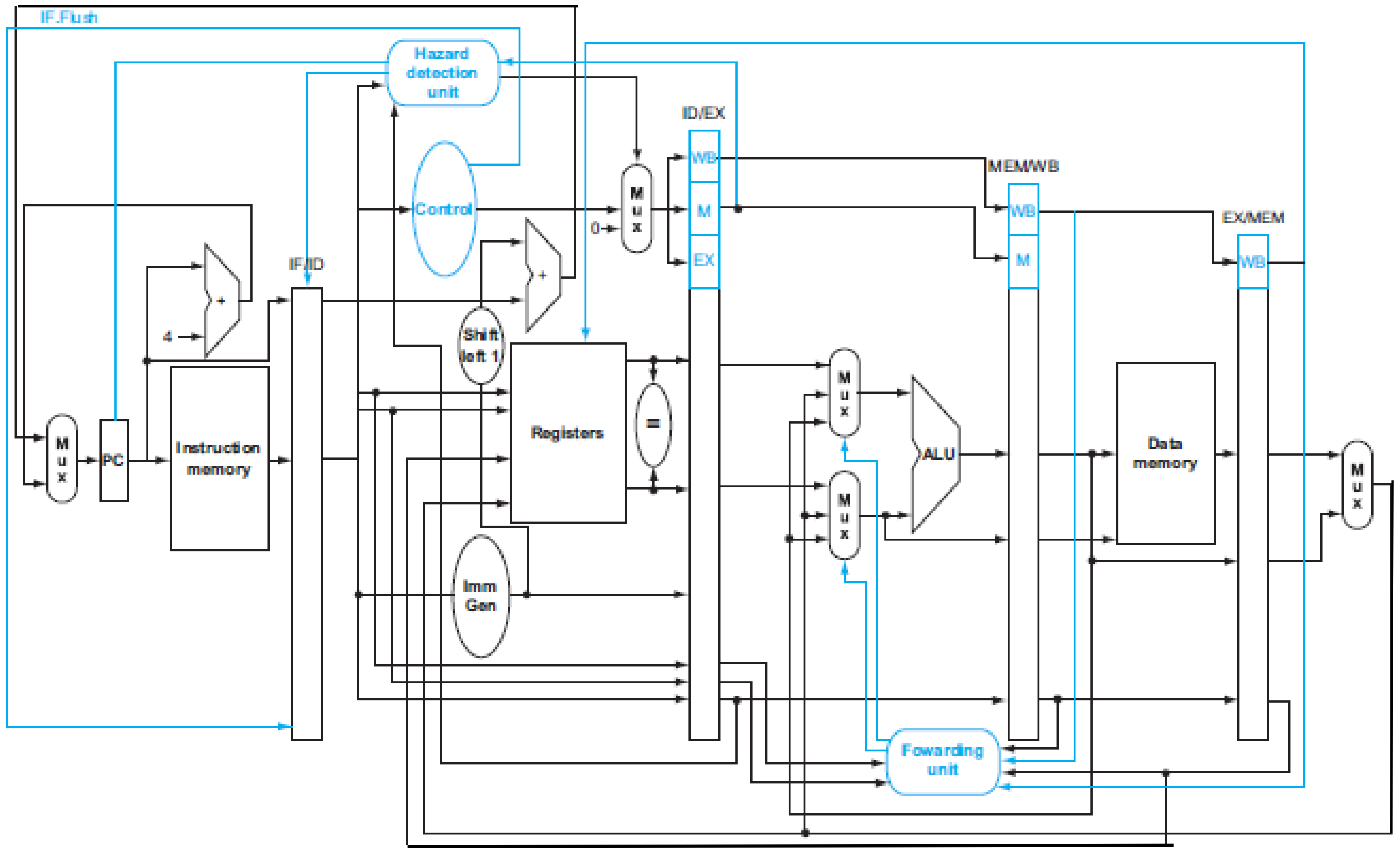
Example: Branch Taken



Example: Branch Taken

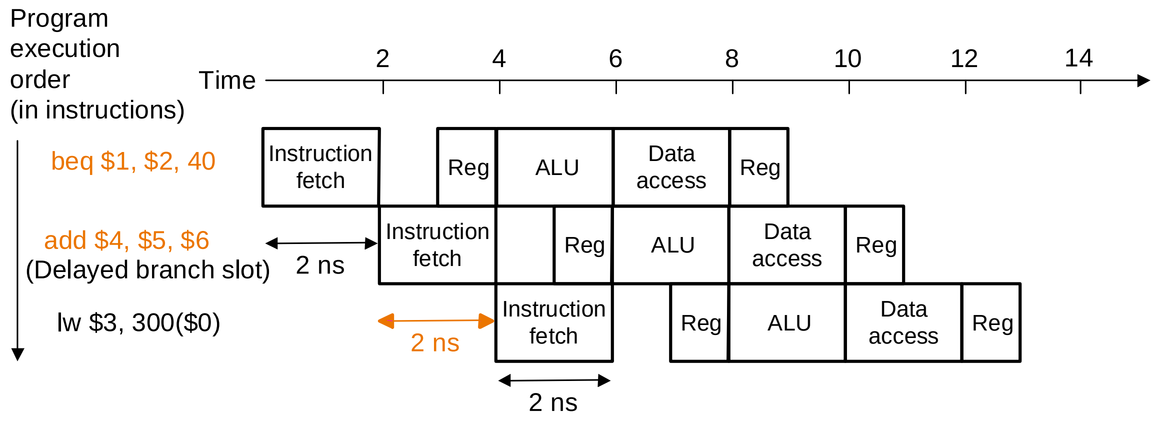


Datapath for branch



More solutions for control hazard

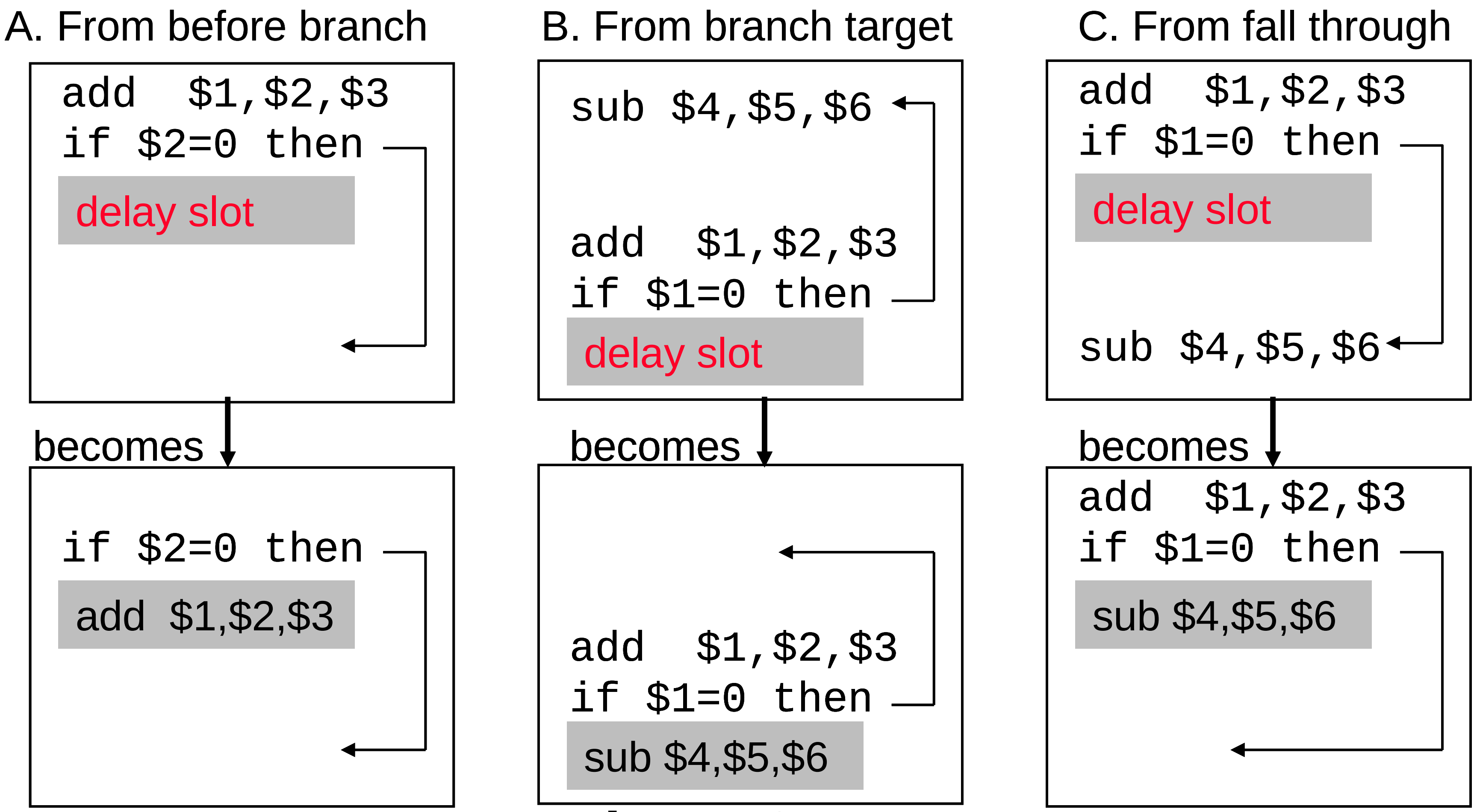
- Software solution by instruction scheduling
  - Three strategies to optimize branch
- Dynamic Branch Prediction
  - dynamic branch prediction
  - predict taken/not taken
  - Predict branch target
- Delayed branch



- Conditional instruction set
  - 4-b condition in 32b ARM instructions

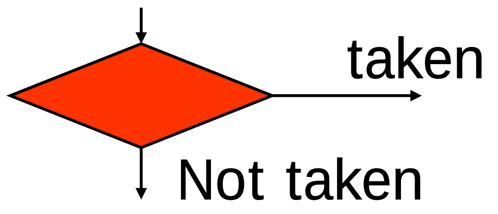
Software solutions for branch hazards

- Software by instruction scheduling
  - branch delay slot
  - moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay



Branch Condition Prediction

- Predict branch direction: taken or not taken (T/NT)



```
    BNE R1, R2, L1
    …
L1: …
```

- Static prediction: compilers decide the direction
- Dynamic prediction: hardware decides the direction using dynamic information
  1. 1-bit Branch-Prediction Buffer
  2. 2-bit Branch-Prediction Buffer
  3. Correlating Branch Prediction Buffer
  4. Tournament Branch Predictor
  5. and more …
- Branch target prediction: branch target buffer (BTB)

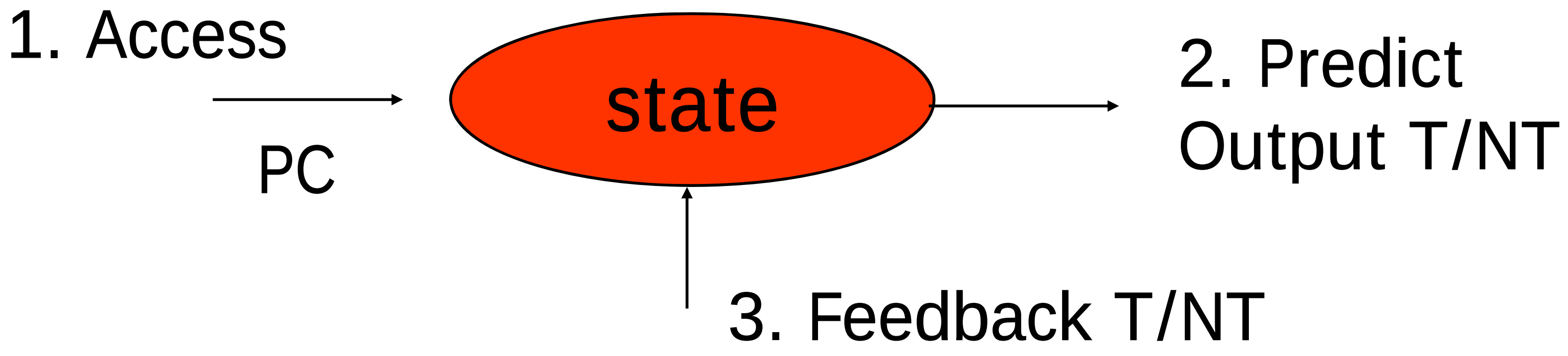
Dynamic Branch Prediction

- Deeper and superscalar pipelines produce larger branch penalty
- A branch prediction buffer (aka branch history table (BHT))

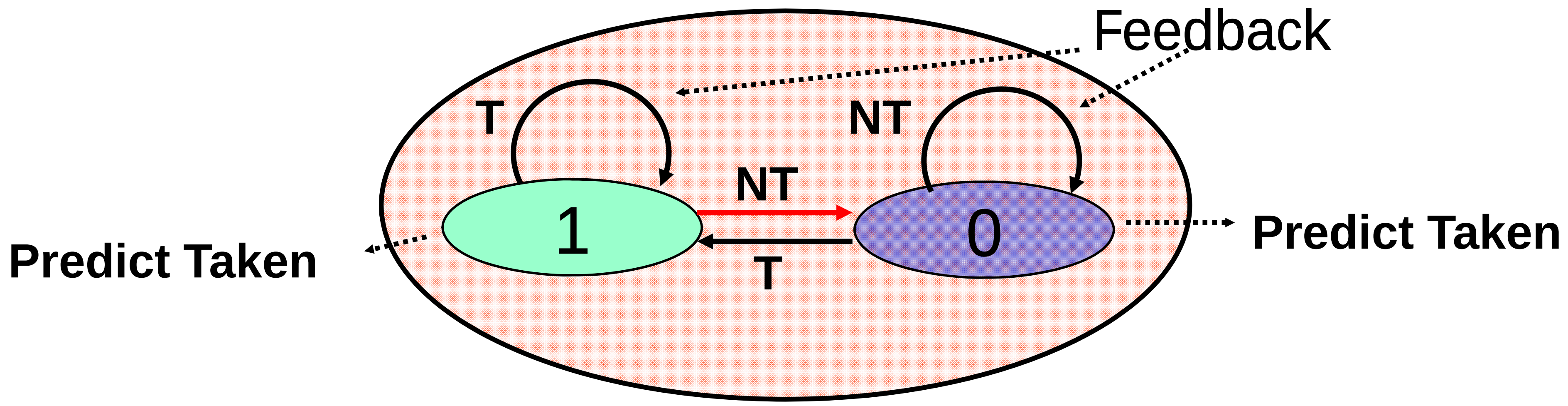
  In the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken last time it was executed
- Use dynamic prediction
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

Predictor for a Single Branch

General Concept

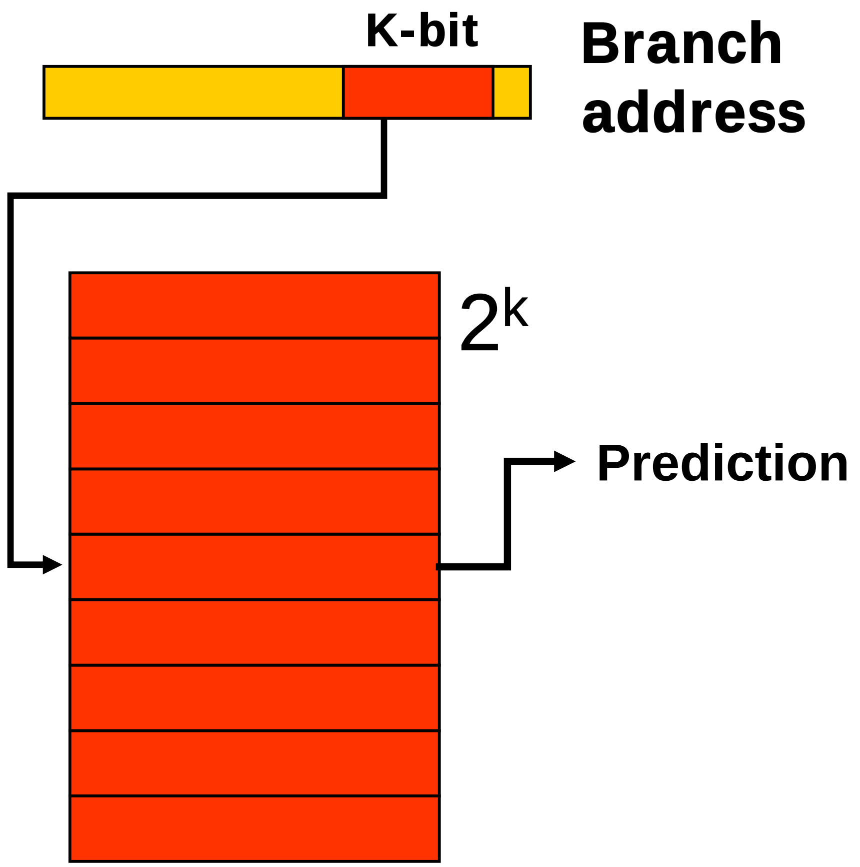


1-bit prediction



Branch History Table BHT of 1-bit Predictor

- Can use only one 1-bit predictor, but accuracy is low
- BHT: use a table of simple predictors, indexed by bits from PC
- Similar to direct mapped cache
- More entries, more cost, but less conflicts, higher accuracy
- BHT can contain complex predictors



1-bit BHT Weakness

- Example: in a loop, 1-bit BHT will cause 2 mispredictions
- Consider a loop of 9 iterations before exit:

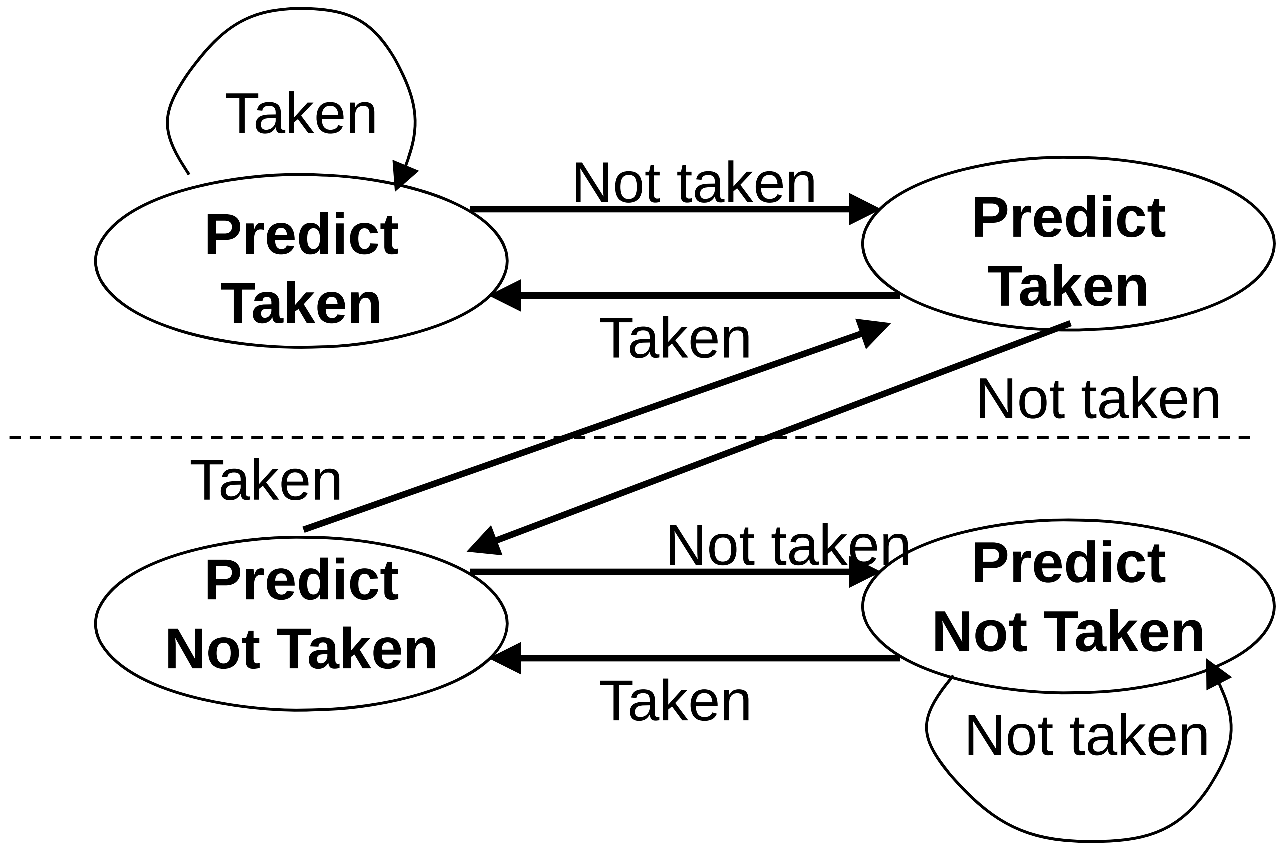
```
for (…){
   for (i=0; i<9; i++)
      a[i] = a[i] * 2.0;
}
```

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Only 80% accuracy even if loop 90% of the time

2-bit Predictors

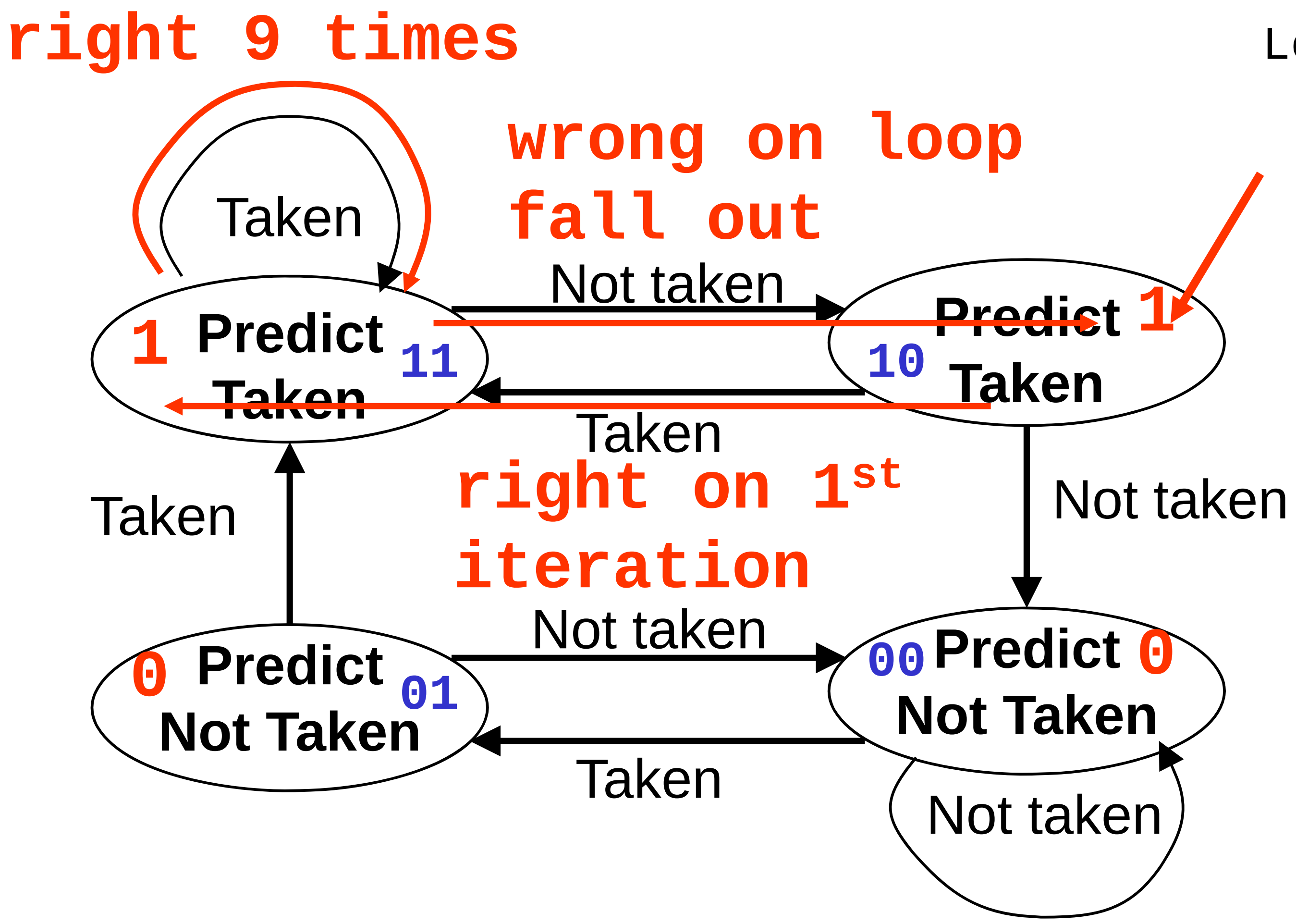
- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed



```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed

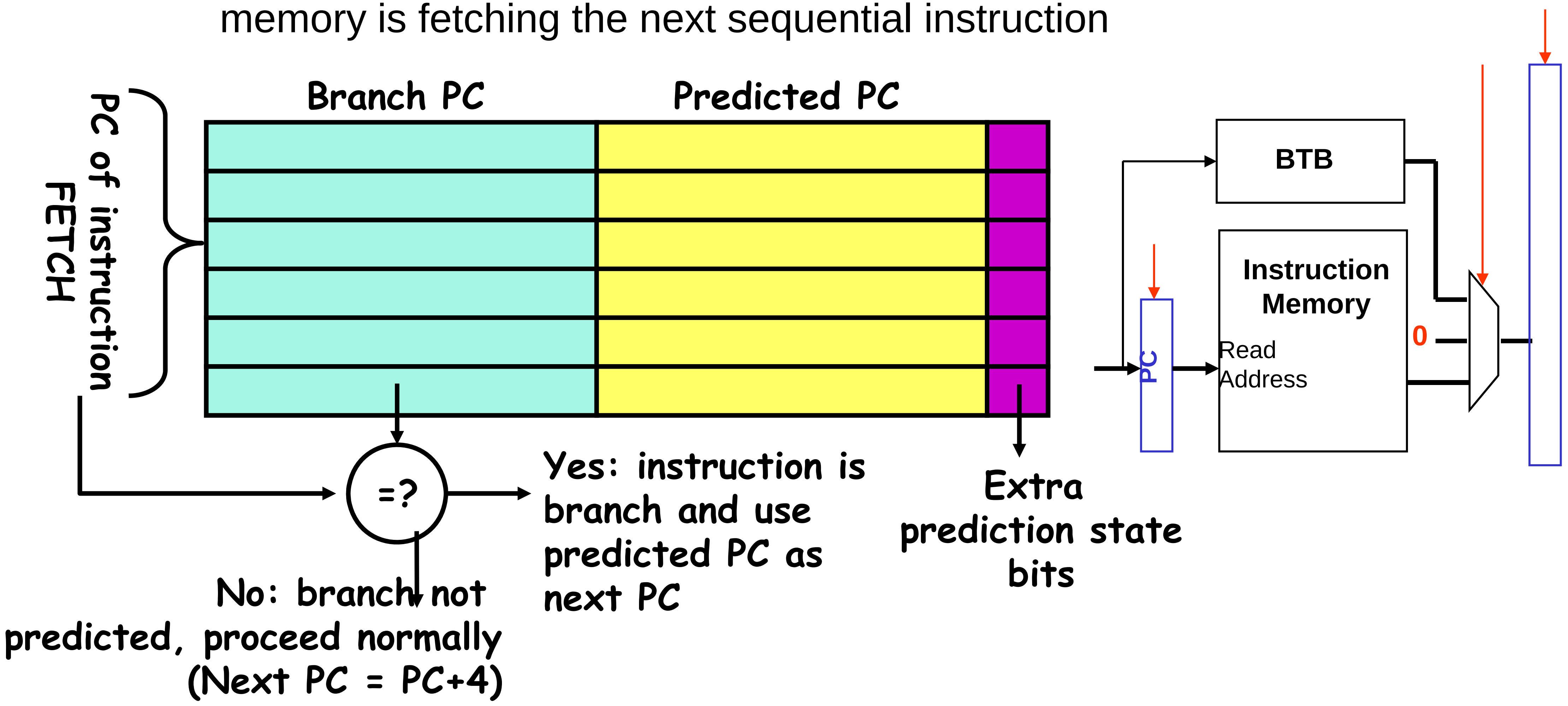


```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT also stores the initial FSM state

Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction

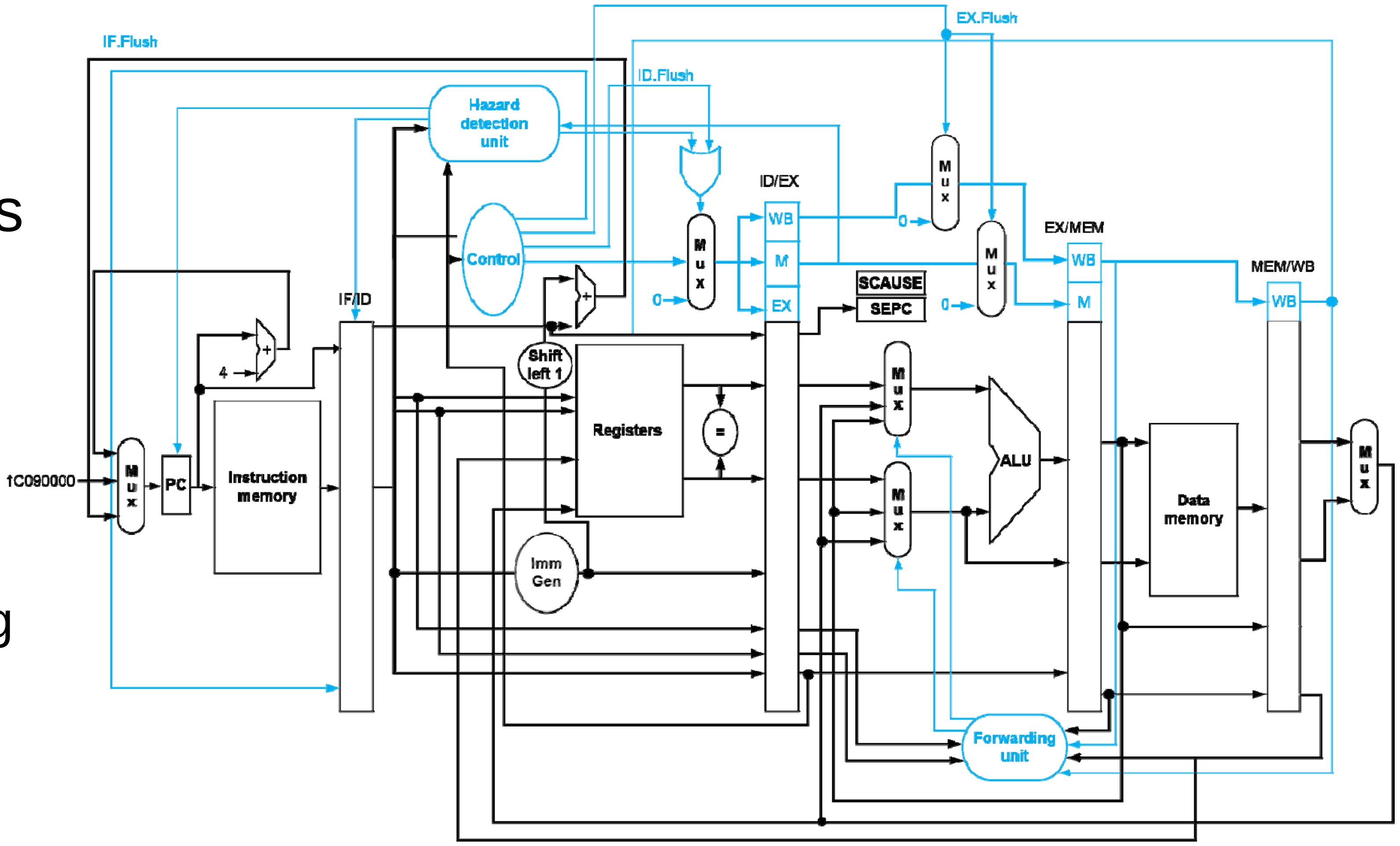


Branch PC
Predicted PC
PC of instruction FETCH
=?
Yes: instruction is branch and use predicted PC as next PC
Extra prediction state bits
No: branch not predicted, proceed normally (Next PC = PC+4)
BTB
Instruction Memory
Read Address
PC
0

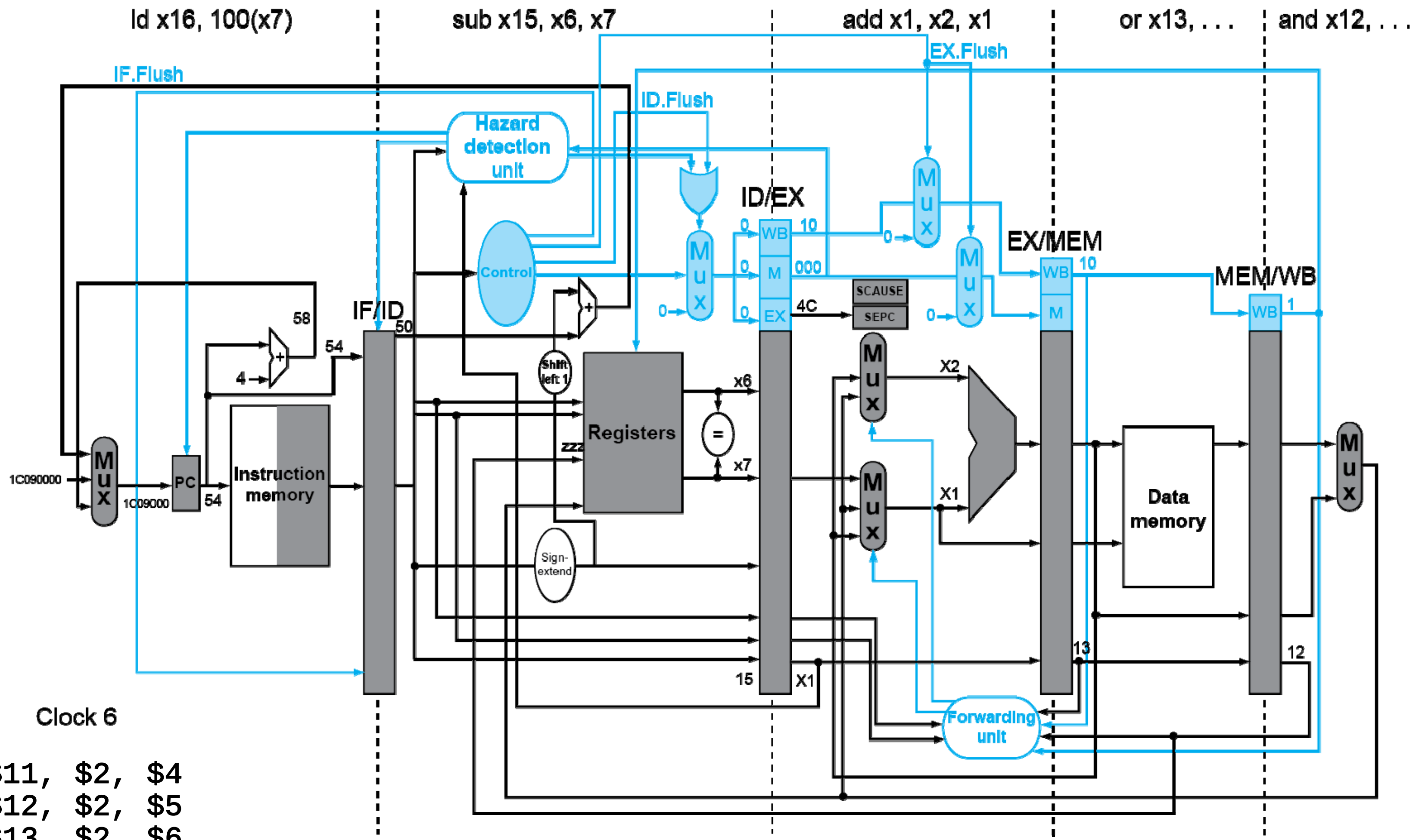
Exceptions on pipeline

- Flush instructions that follow faulted instruction from pipeline
- Save address of offending instruction

- Precise interrupts
  - complete prior instruction
  - stop offending instruction in midstream
  - flush all following instructions



Exception Example @ 6th cycle



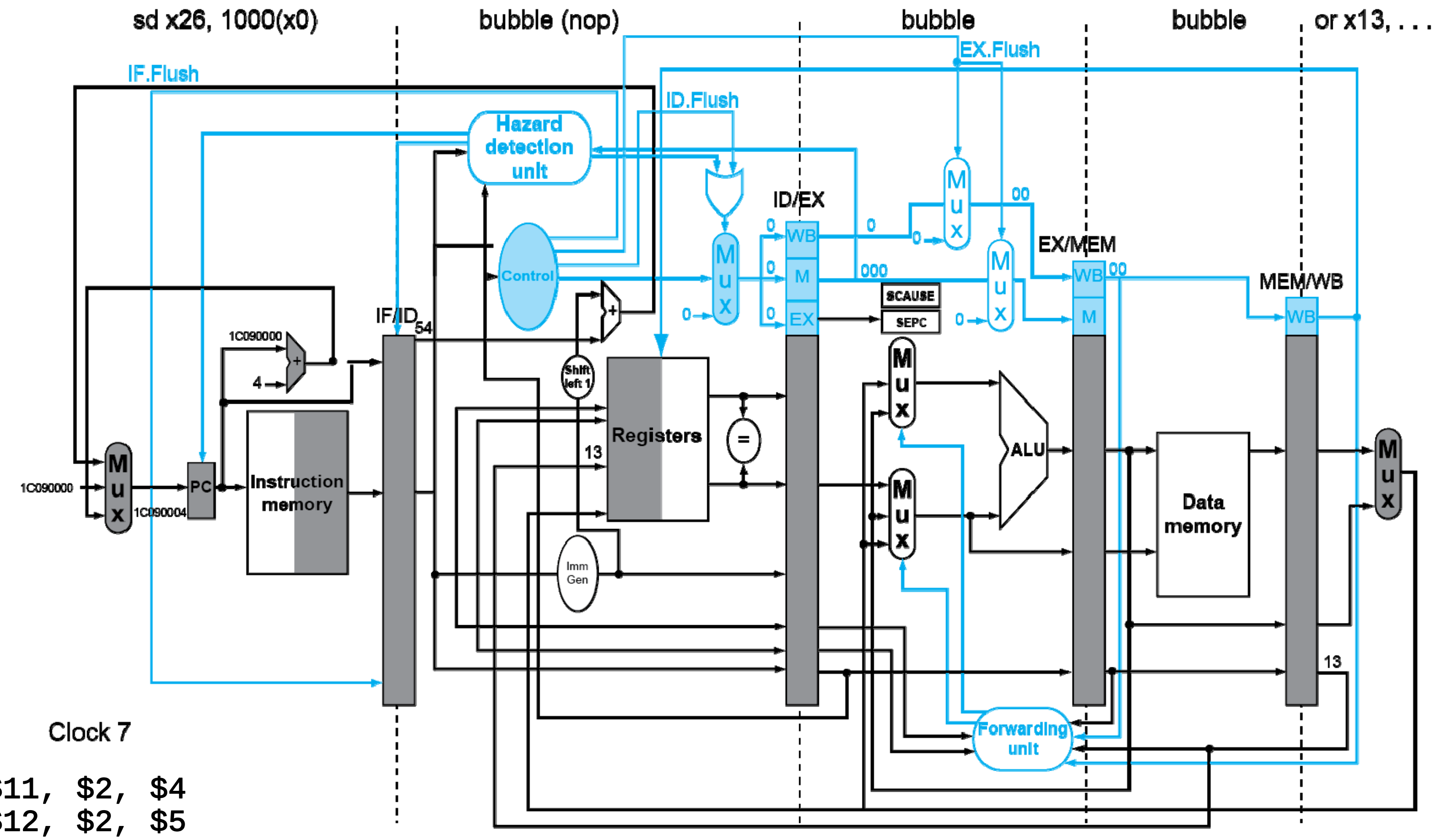
Exception on add in

```
40          sub  $11, $2, $4
44          and  $12, $2, $5
48          or   $13, $2, $6
4C          add  $1,  $2, $1
50          slt  $15, $6, $7
54          lw   $16, 50($7)
…
```

Handler

```
80000180    sw   $25, 1000($0)
80000184    sw   $26, 1004($0)
…
```

Exception Example @ 7th cycle



Exception on add **in**

```
40        sub  $11, $2, $4
44        and  $12, $2, $5
48        or   $13, $2, $6
4C        add  $1,  $2, $1
50        slt  $15, $6, $7
54        lw   $16, 50($7)
…
```

Handler

```
80000180  sw   $25, 1000($0)
80000184  sw   $26, 1004($0)
…
```

Types of Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - – Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in terms of instructions
- Data-level parallelism (DLP)

```
DO  I = 1  TO  100
   A[I] = A[I] + 1
CONTINUE
```

- **To increase ILP**
  - Deeper pipeline

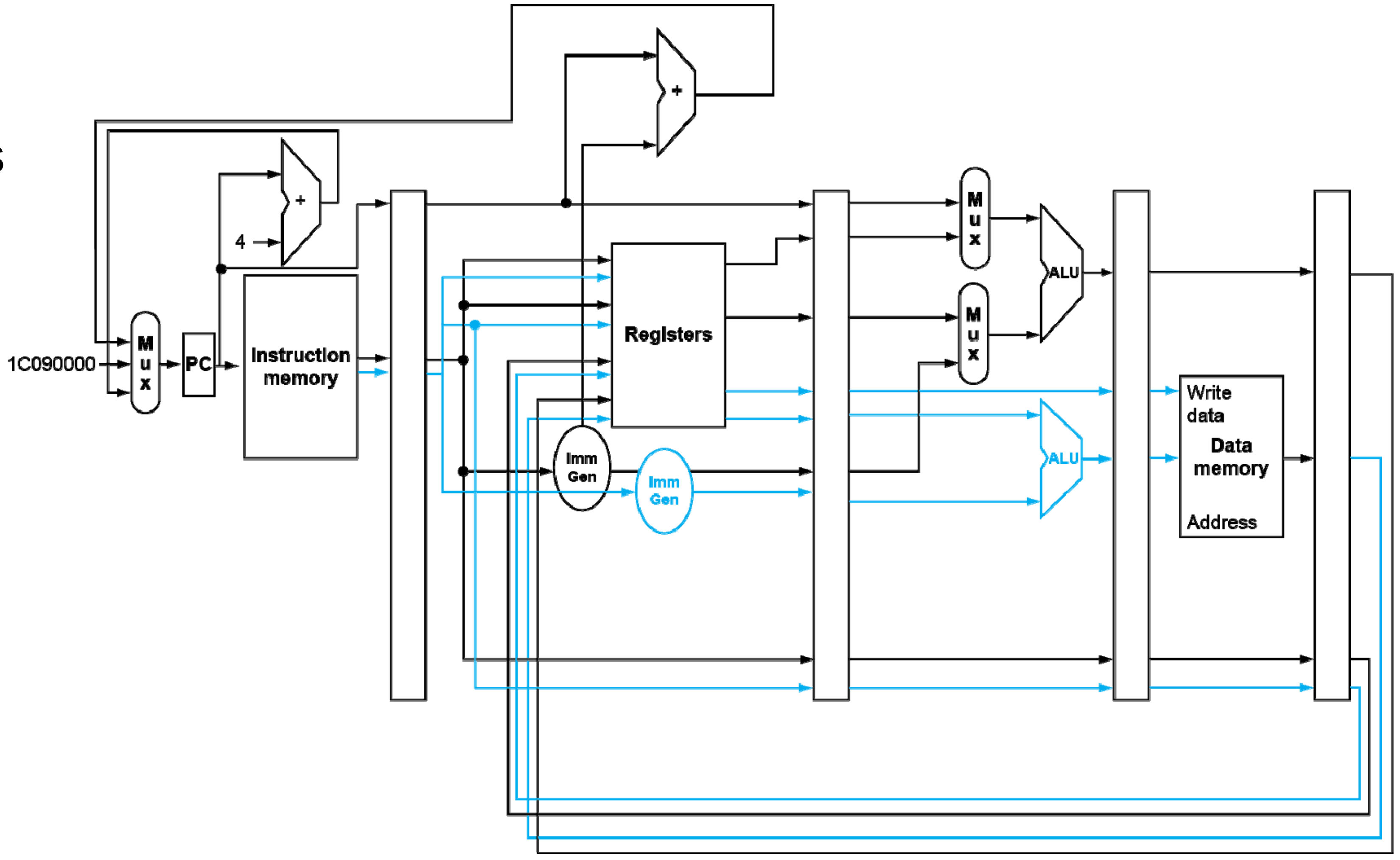
    Less work per stage shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages   multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128-bit "bundles" containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for speculation and predication
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power series, Pentium 4, MIPS R10K, AMD Barcelona

A static two-issue datapath

- What we add:
  - –two 32b instructions from Imem
  - –four read ports
  - –Two write ports
  - –Additional ALU



RISC-V with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Scheduling Example

- Schedule this for dual-issue RISC-V

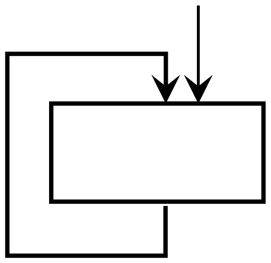
```
Loop: ld   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sd   x31,0(x20)      // store result
      addi x20,x20,-8      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

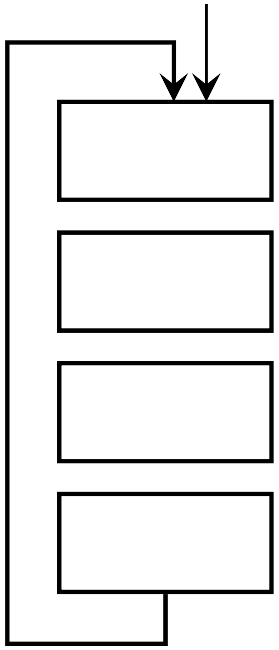
	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called "register renaming"
  - Avoid loop-carried "anti-dependencies"
    - Store followed by a load of the same register
    - Aka "name dependence"
      - Reuse of a register name





```
DO   I = 1   TO   100
   A[I] = A[I] + 1
CONTINUE
```

```
DO   I = 1   TO   100 by 4
   A[I] = A[I] + 1
   A[I+1] = A[I+1] + 1
   A[I+2] = A[I+2] + 1
   A[I+3] = A[I+3] + 1
CONTINUE
```

Loop Unrolling Example

```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```

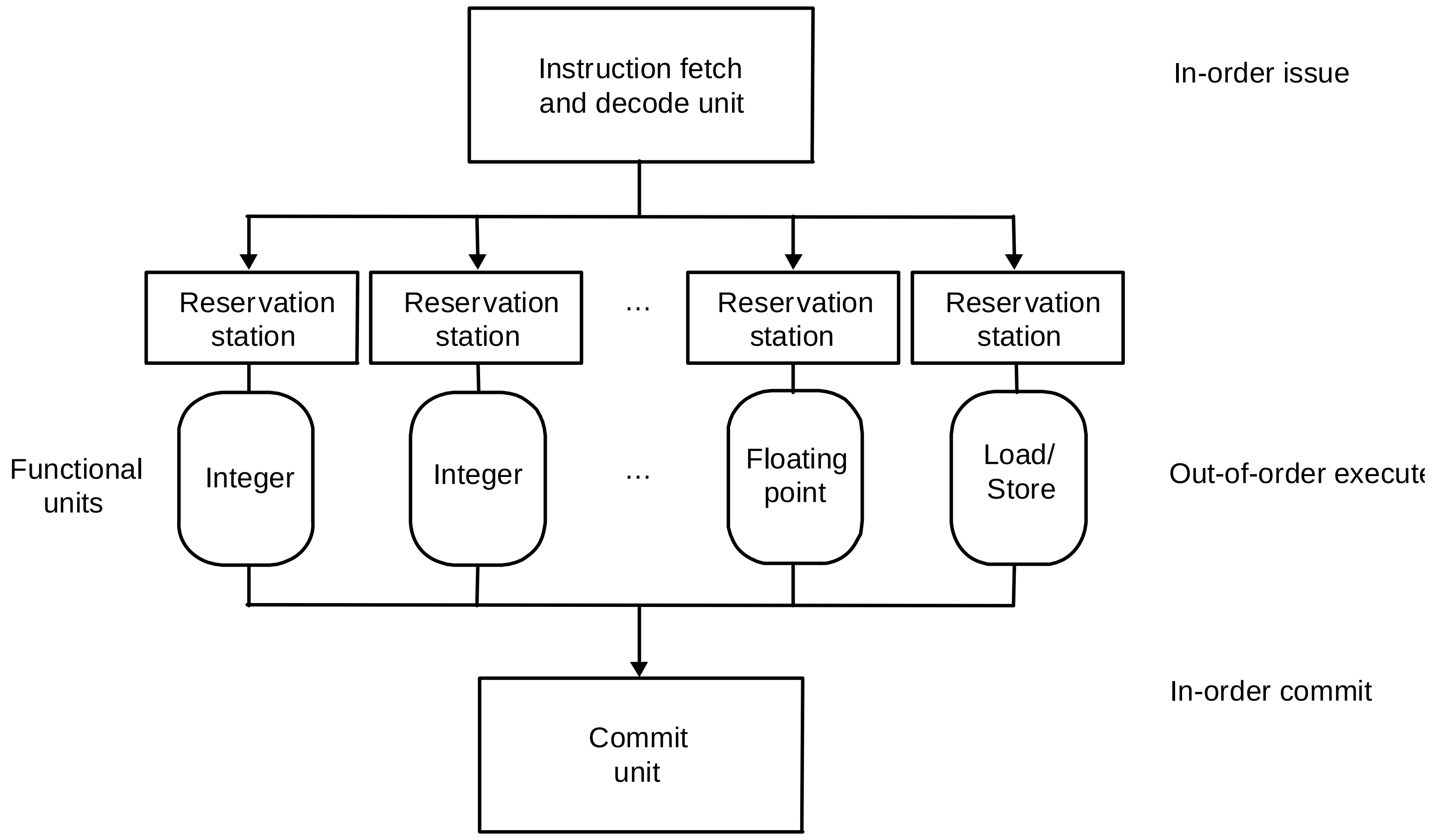
```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```

	ALU/branch	Load/store	cycle
`Loop:`	`addi x20,x20,-32`	`ld  x28, 0(x20)`	1
	`nop`	`ld  x29, 24(x20)`	2
	`add x28,x28,x21`	`ld  x30, 16(x20)`	3
	`add x29,x29,x21`	`ld  x31, 8(x20)`	4
	`add x30,x30,x21`	`sd  x28, 32(x20)`	5
	`add x31,x31,x21`	`sd  x29, 24(x20)`	6
	`nop`	`sd  x30, 16(x20)`	7
	`blt x22,x20,Loop`	`sd  x31, 8(x20)`	8

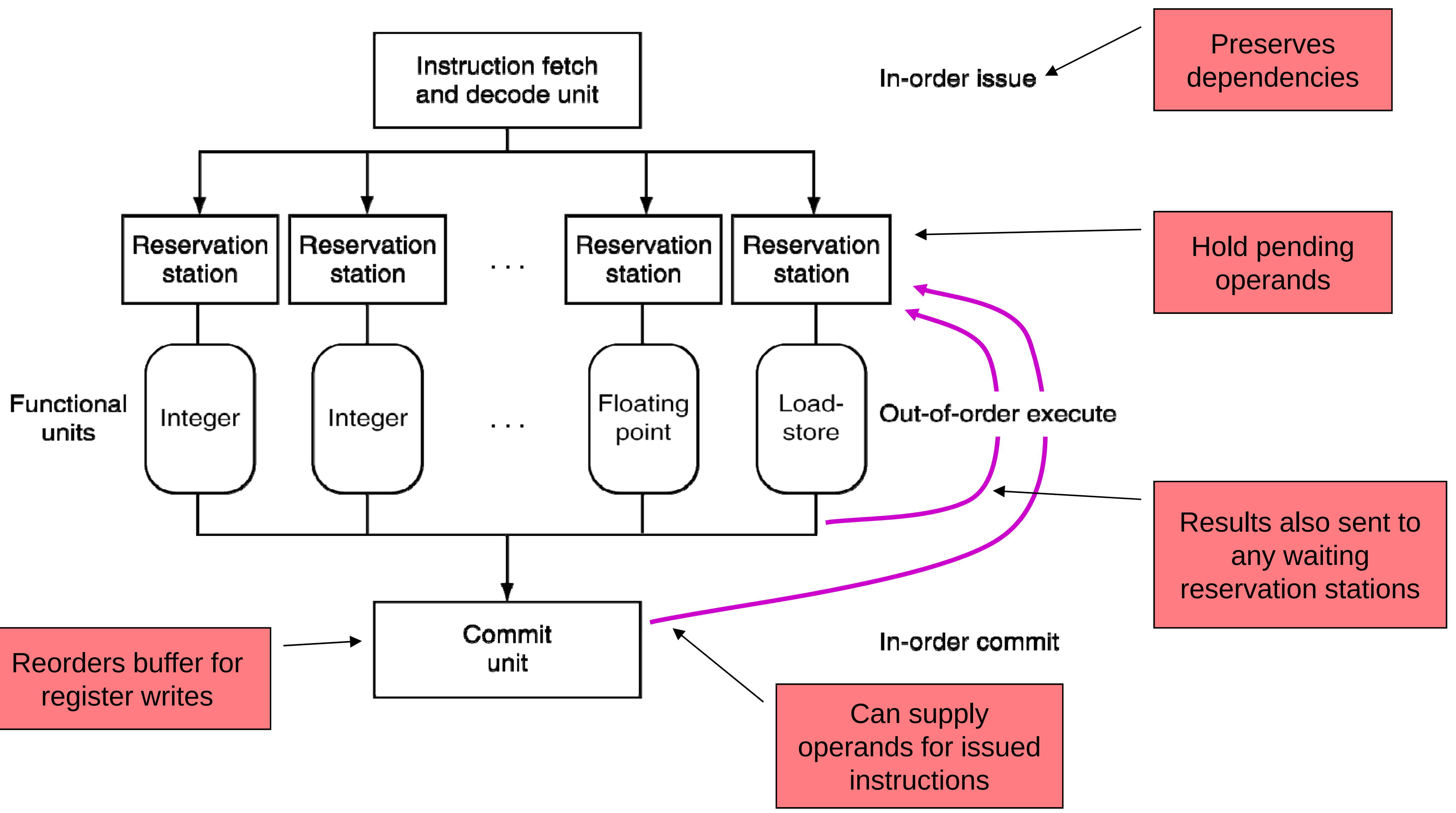
- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Pipeline Scheduling

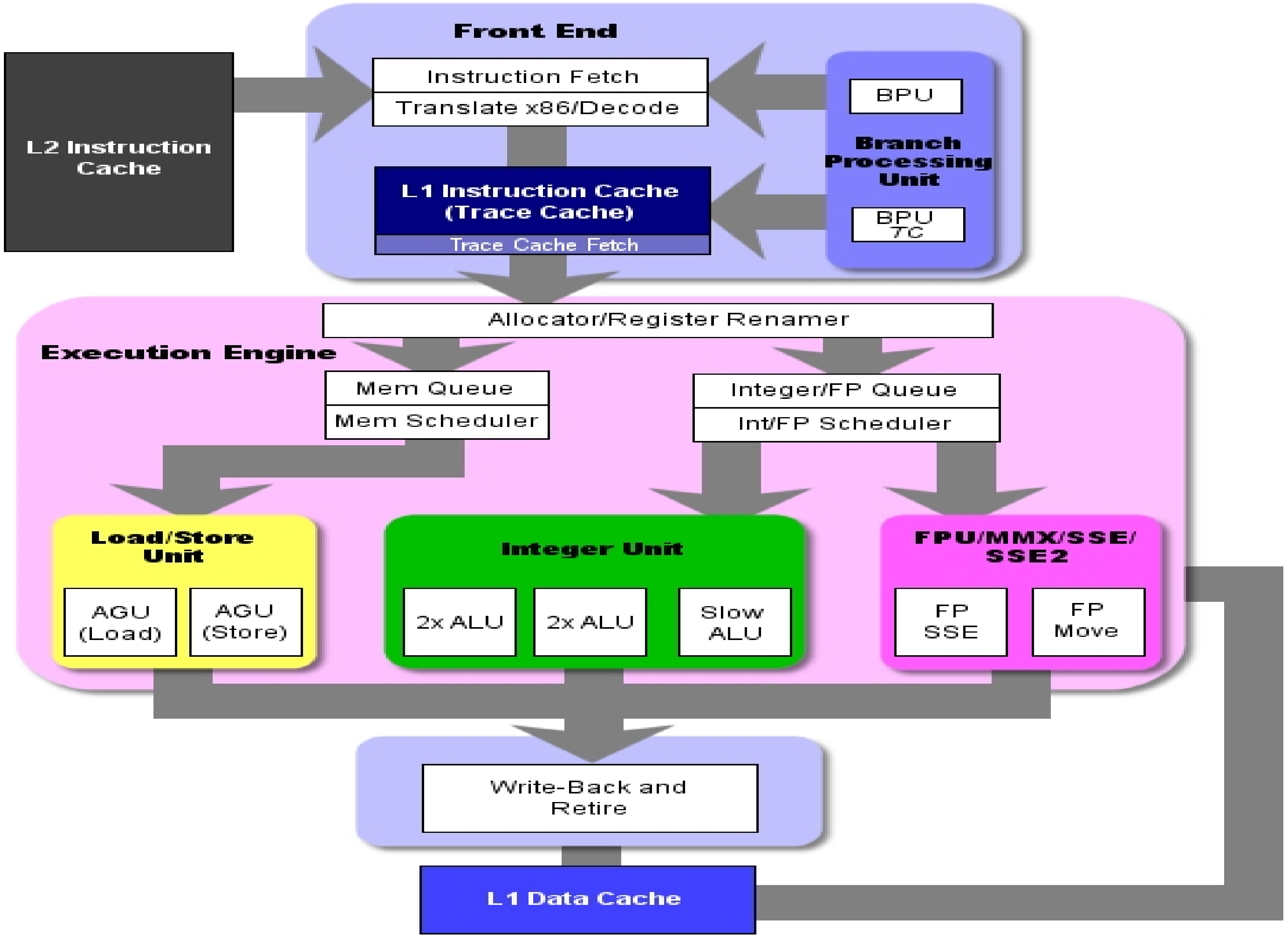
- hardware tries to find instructions to execute
- out of order execution is possible
- speculative execution and dynamic branch prediction



Superscalar by dynamically scheduling

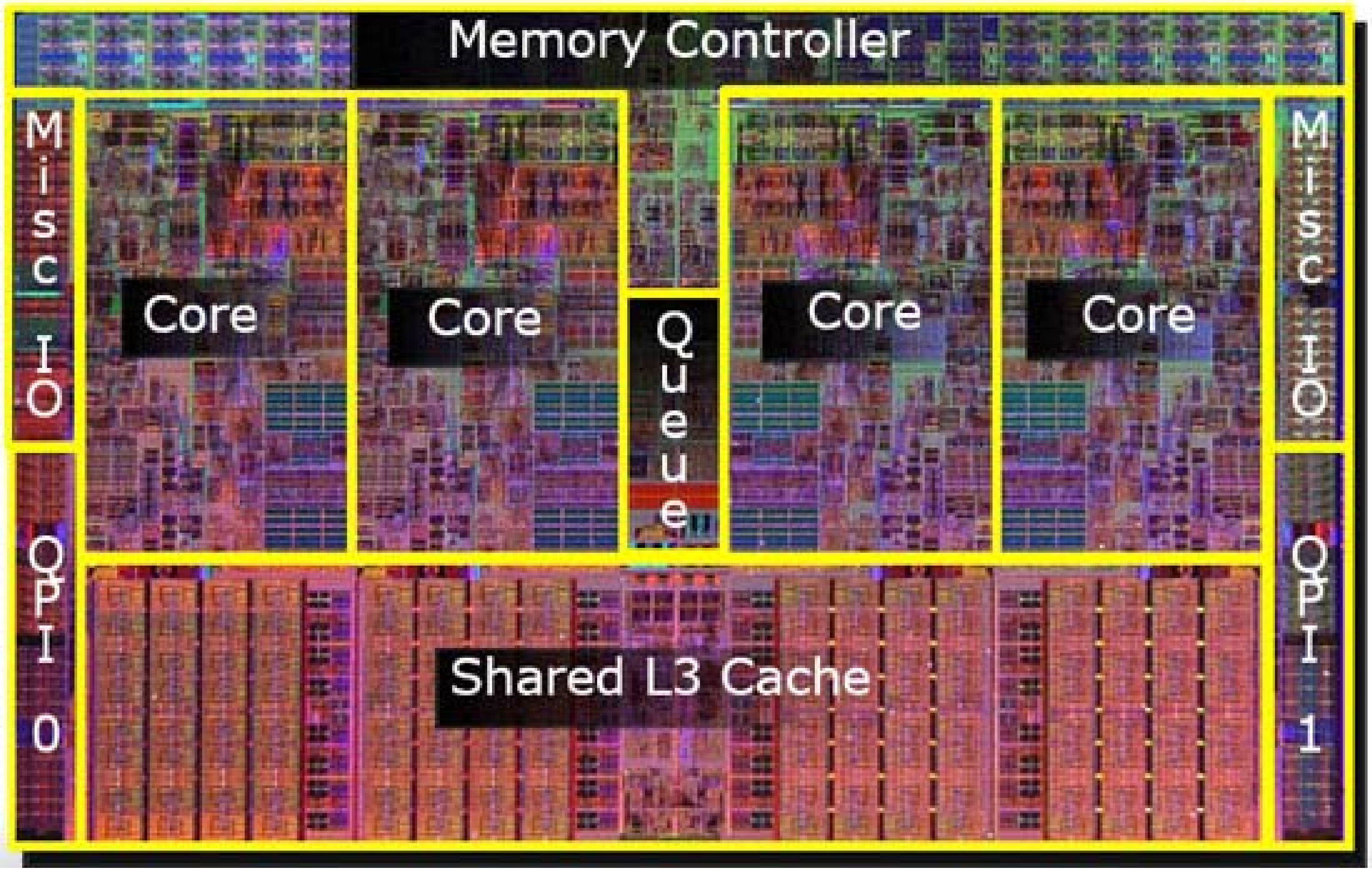


Intel Pentium IV

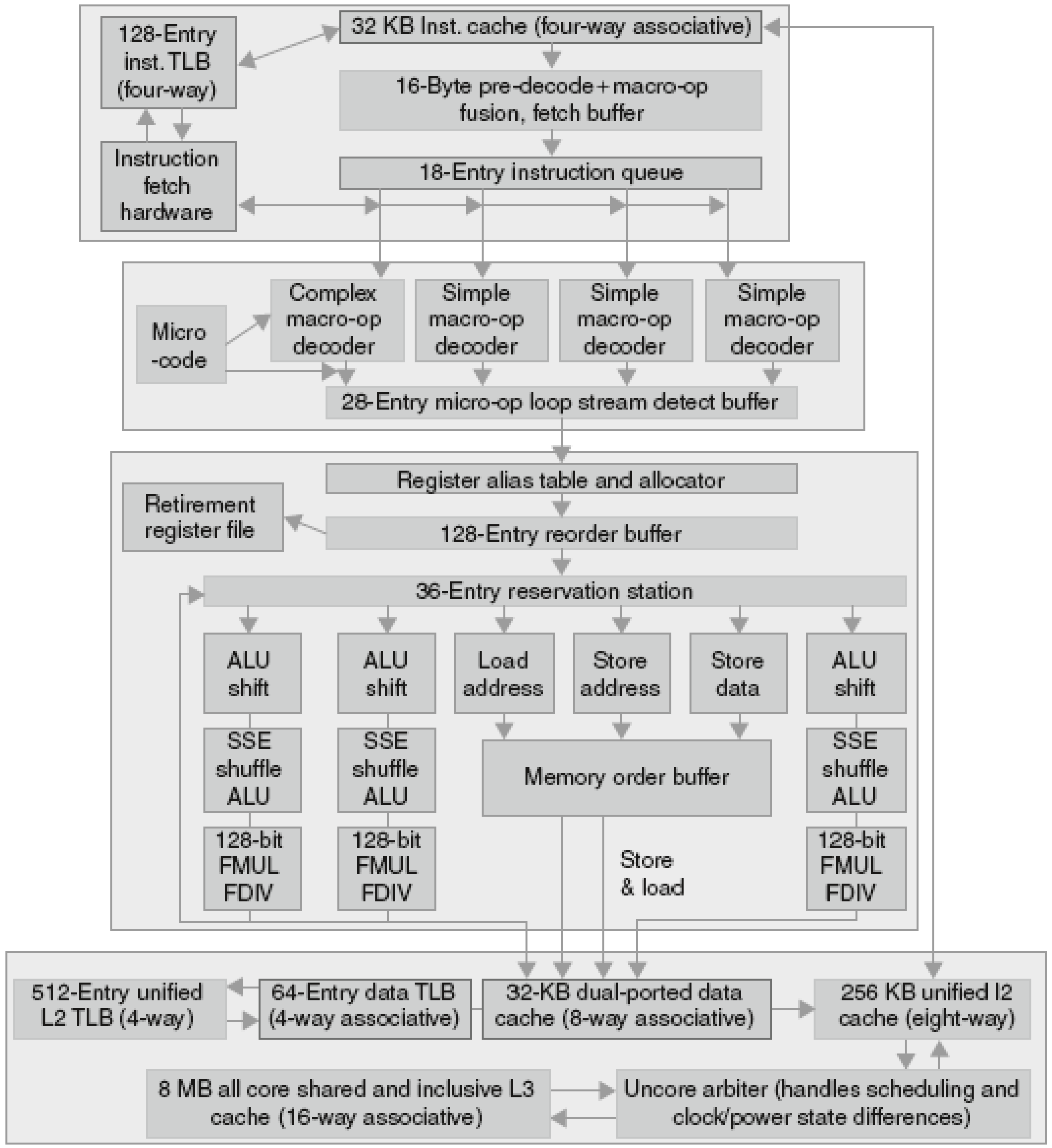


How about the most recent Intel i7?

- The new Core i7 processor
  - 1366 pins
  - 263 mm2
  - 731 Million transistors



Core i7 Pipeline


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte 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)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Power Efficiency

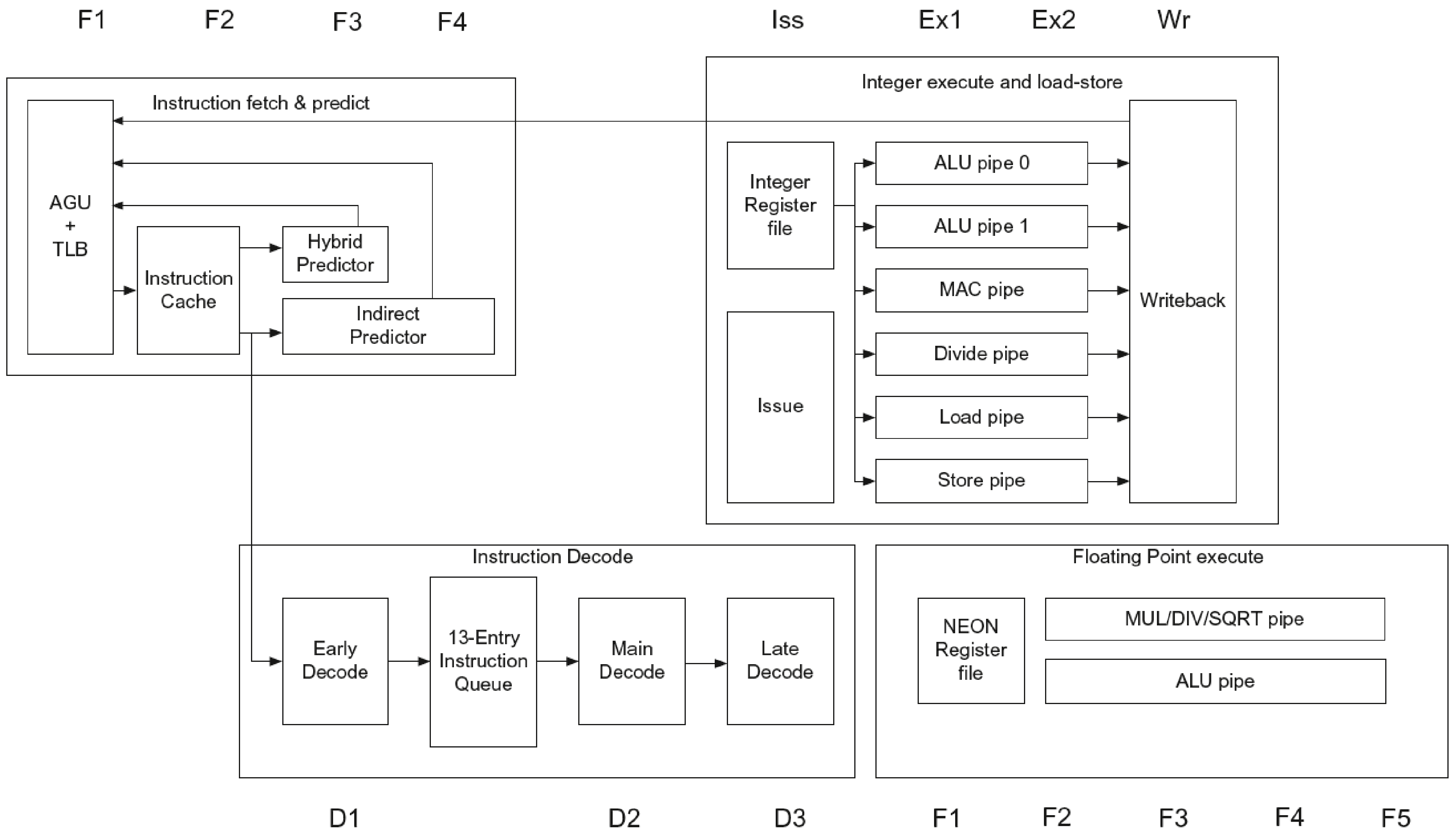
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

ARM Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

ARM Cortex-A53 Pipeline



Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots