Chapter 4C

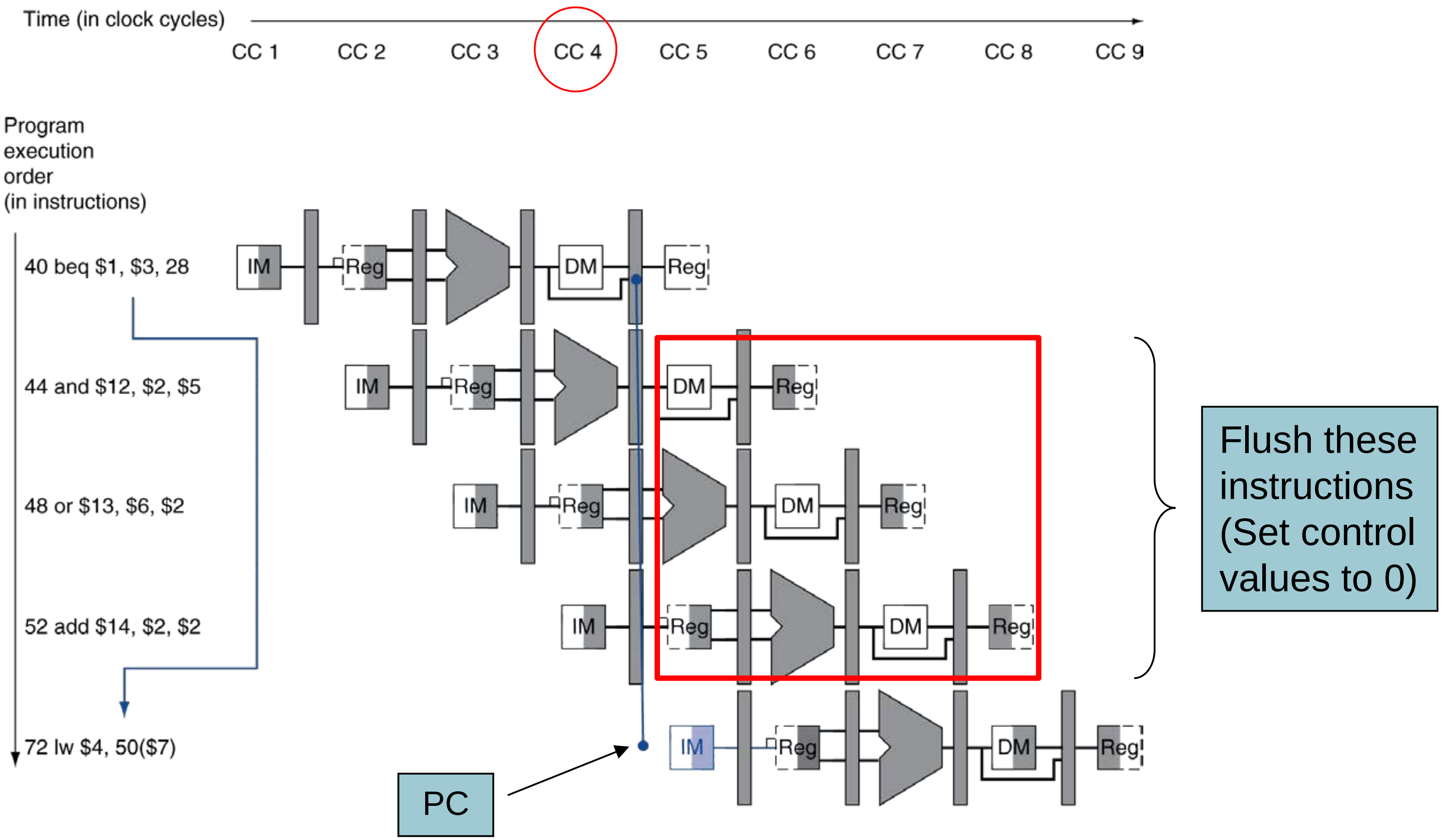
The Processor

Assume Branch Not Taken

- Continue execution down the sequential instruction stream.
- If taken, an instruction in IF stage must be discarded.
  - Execution continues at the branch target.

Branch Hazards

- If branch outcome determined in MEM



Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7 # PC-relative: 40+4+7*4=72
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

Target address calculation

- PC+4 is already available
- Immediate file in in IF/ID register
- Therefore, we just move the branch adder form the EX stage to the ID stage.
- The branch target address calculation will be performed for all instructions
  - But used only when needed.

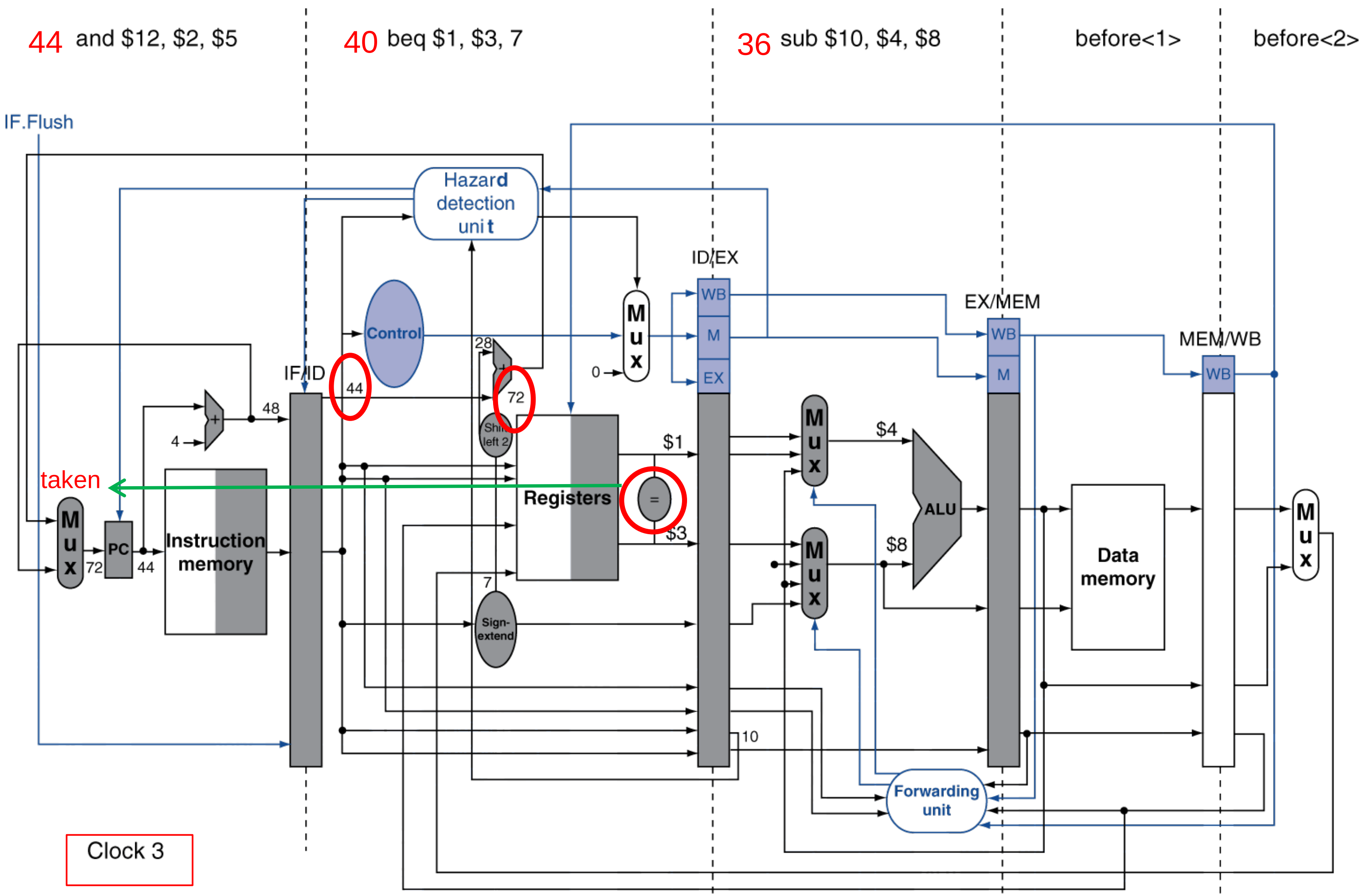
Branch decision

- Moving the branch test to the ID stage
- Additional forwarding and hazard detection hardware are needed.
- The bypass source operands can come form EX/MEM register and MEM/WB register
- If the values in branch comparison is produce later in time, a data hazard can occur and a stall will be needed.

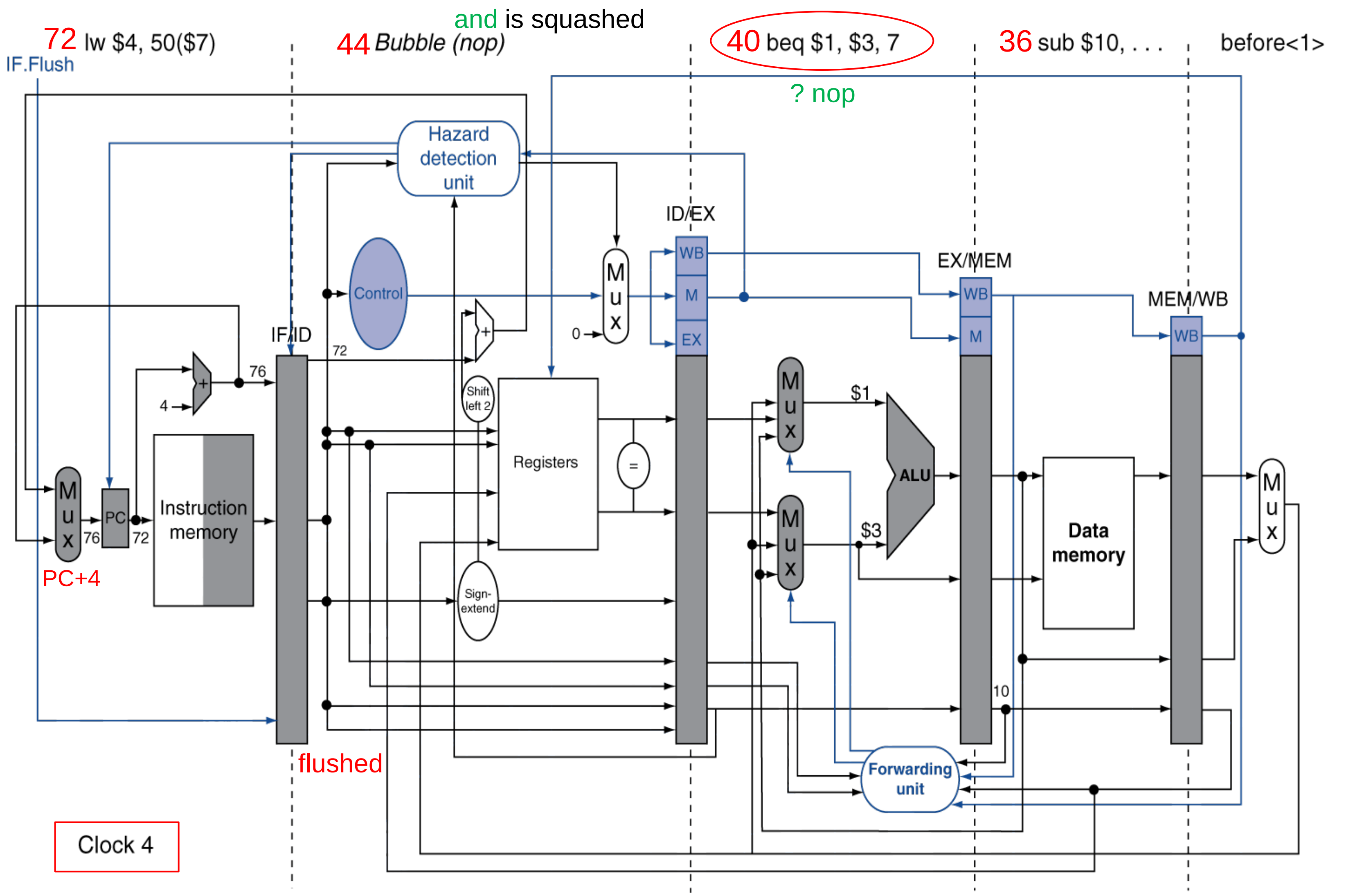
Branch decision

- Moving the branch test to the ID stage
- Additional forwarding and hazard detection hardware are needed.
- The bypass source operands can come form EX/MEM register and MEM/WB register
- If the values in branch comparison is produce later in time, a data hazard can occur and a stall will be needed.

Example: Branch Taken



Example: Branch Taken



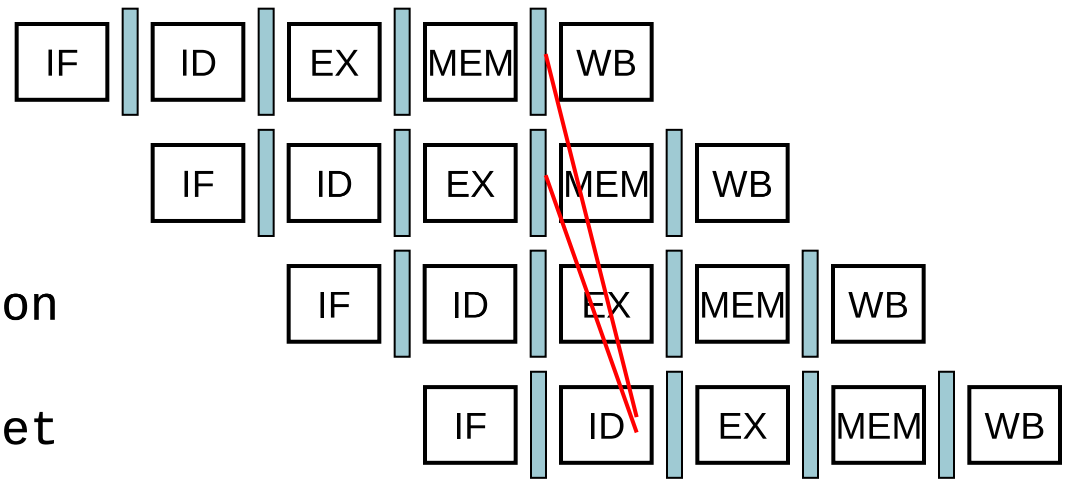
Branch decision in ID

- Equality: 32 2-input XNOR gates and a 32-input AND gate
- Additional forwarding may be required for branch decision in ID.
  - The bypassed source operands from EX/MEM or MEM/WB pipeline registers
- Because the values in a branch comparison are needed during ID but may be produced later in time, data hazard should be detected for stalls
  - A stall cycle will be needed if one of the operands is the result of an immediately preceding ALU instruction.
  - Two stall cycles will be needed if a load is immediately preceding followed by a branch.

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
xxx an instruction
beq $1, $4, target
```

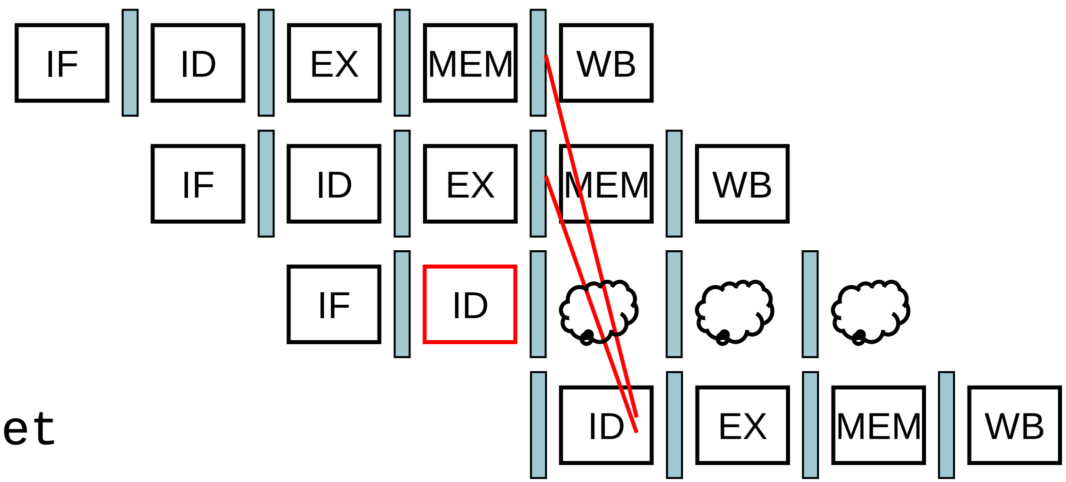


- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

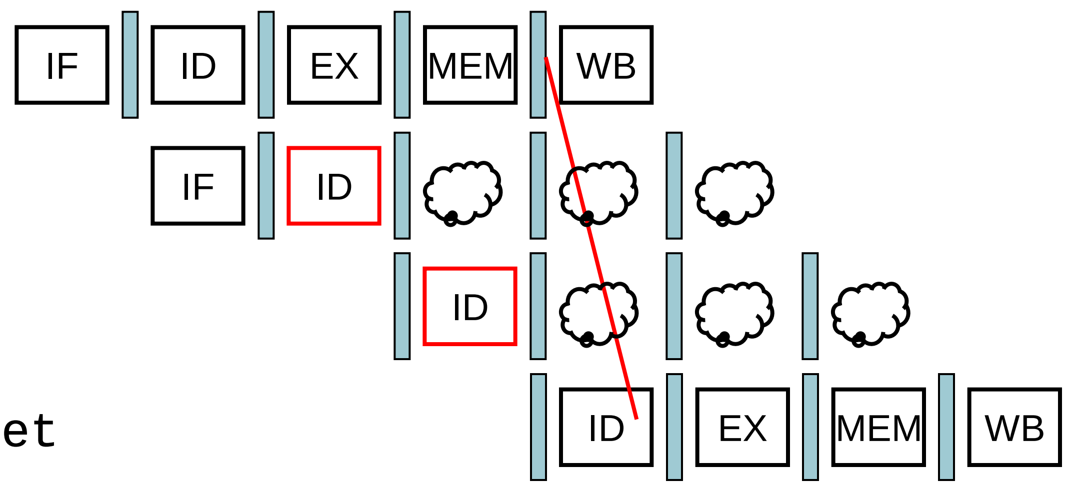
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
    - A small memory indexed by lower portion of the recent branch instruction addresses
    - Stores outcome (taken/not taken)
- To execute a branch with branch prediction table
  - Check table, expect the same outcome (1-bit prediction)
  - Start fetching from fall-through or target according to prediction
  - If wrong, flush pipeline and flip prediction bit

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

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

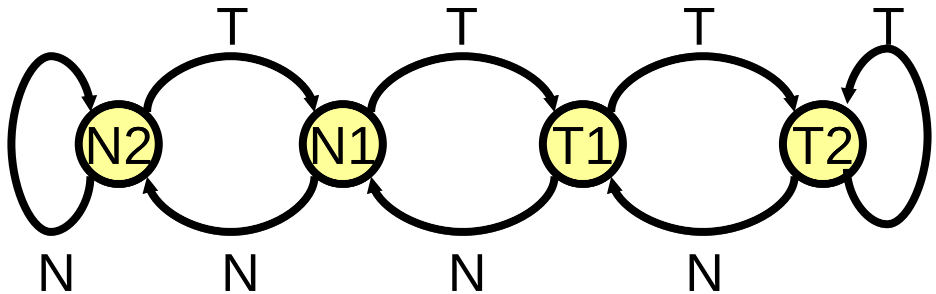
1. Mispredict as taken on last iteration of inner loop → change to not taken
2. Then mispredict as not taken on first iteration of inner loop next time around: wrong!

Even if a branch is almost always taken, we can predict incorrectly twice, rather than one, when it is not taken.

2-bit predictor: better

- A 'predict same as last' strategy gets two mispredicts on each loop
  - Predict NTTT…TTT
  - Actual TTTT…TTN
- Can do much better by adding *inertia* to the predictor
  - e.g., two-bit saturating counter
  - Predict TTTT…TTT
  - Actual TTTT…TTN

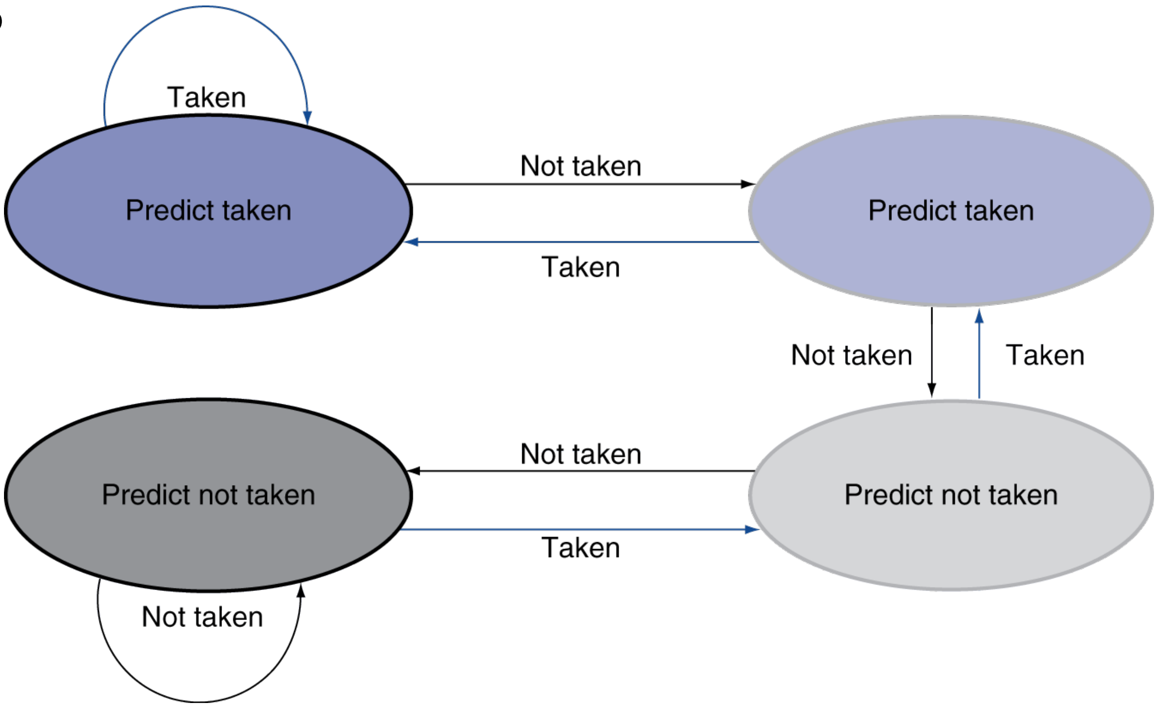
```
for(j=0;j<30;j++) {
  …
}
```



2-Bit Predictor

- Only change prediction on two successive mispredictions

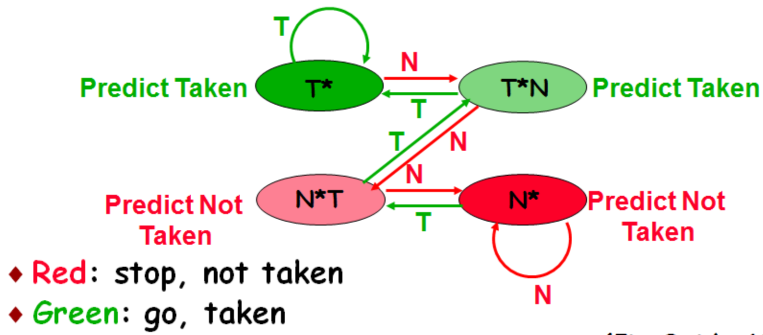
1: 01 (T2)
0: 00 (T1)
-1: 11 (N1)
-2: 10 (N2)



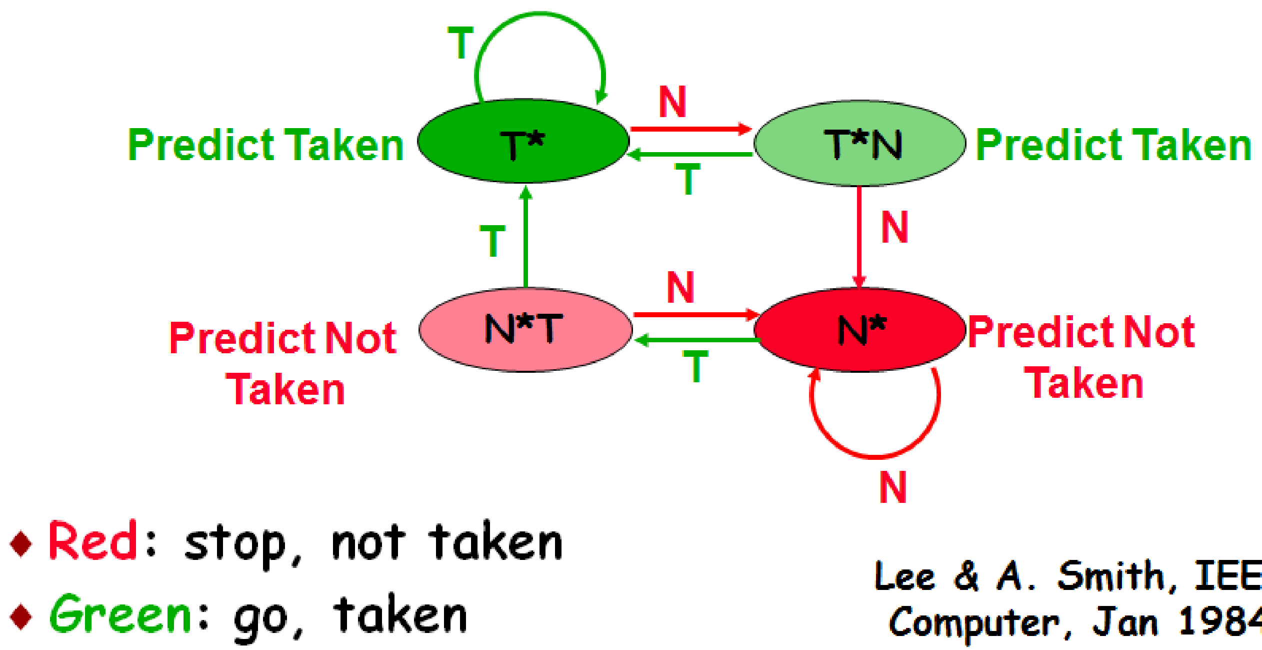
- 2-bit counter:
  - taken: increment,
  - not taken: decrement

Another 2-Bit Predictor

- A little different from the previous version



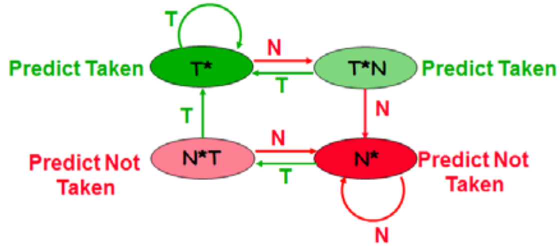
(Jim Smith, 1981)



Lee & A. Smith, IEEE Computer, Jan 1984

Comparison

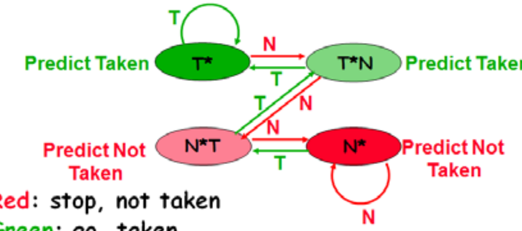
2



- Red: stop, not taken
- Green: go, taken

Lee & A. Smith, IEEE Computer, Jan 1984

1



- Red: stop, not taken
- Green: go, taken

(Jim Smith, 1981)

Actual: T N T T T N T
State: T* T* T*N T* T* T* T*N T*
Predicted: T T T T T T T

Actual: N N T N N T N N
State: N* N* N* N*T N* N* N*T N*
Predicted: N N N N N N N N

For both schemes

Actual: N N T T N N T T N
State: N* N* N* N*T T*N N*T N* N*T T*N
Predicted: N N N N T N N N T

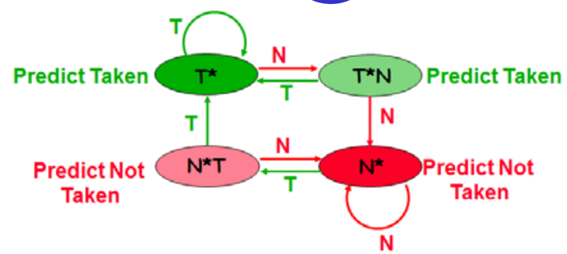
Scheme 1

Actual: N N T T N N T T N
State: N* N* N* N*T T* T*N N* N*T T*N
Predicted: N N N N ? ? ? ? ?

Scheme 2

Comparison

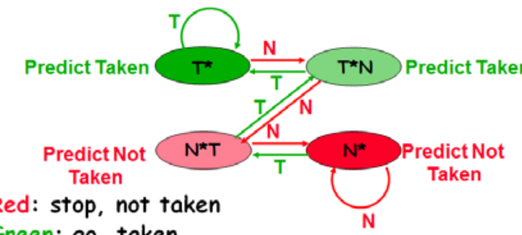
2



- Red: stop, not taken
- Green: go, taken

Lee & A. Smith, IEEE Computer, Jan 1984

1



- Red: stop, not taken
- Green: go, taken

(Jim Smith, 1981)

Actual: T N T T T N T
State: T* T* T*N T* T* T* T*N T*
Predicted: T T T T T T T

Actual: N N T N N T N N
State: N* N* N* N*T N* N* N*T N*
Predicted: N N N N N N N N

For both schemes

Actual: N N T T N N T T N
State: N* N* N* N*T T*N N*T N* N*T T*N
Predicted: N N N N T N N N T

Scheme 1

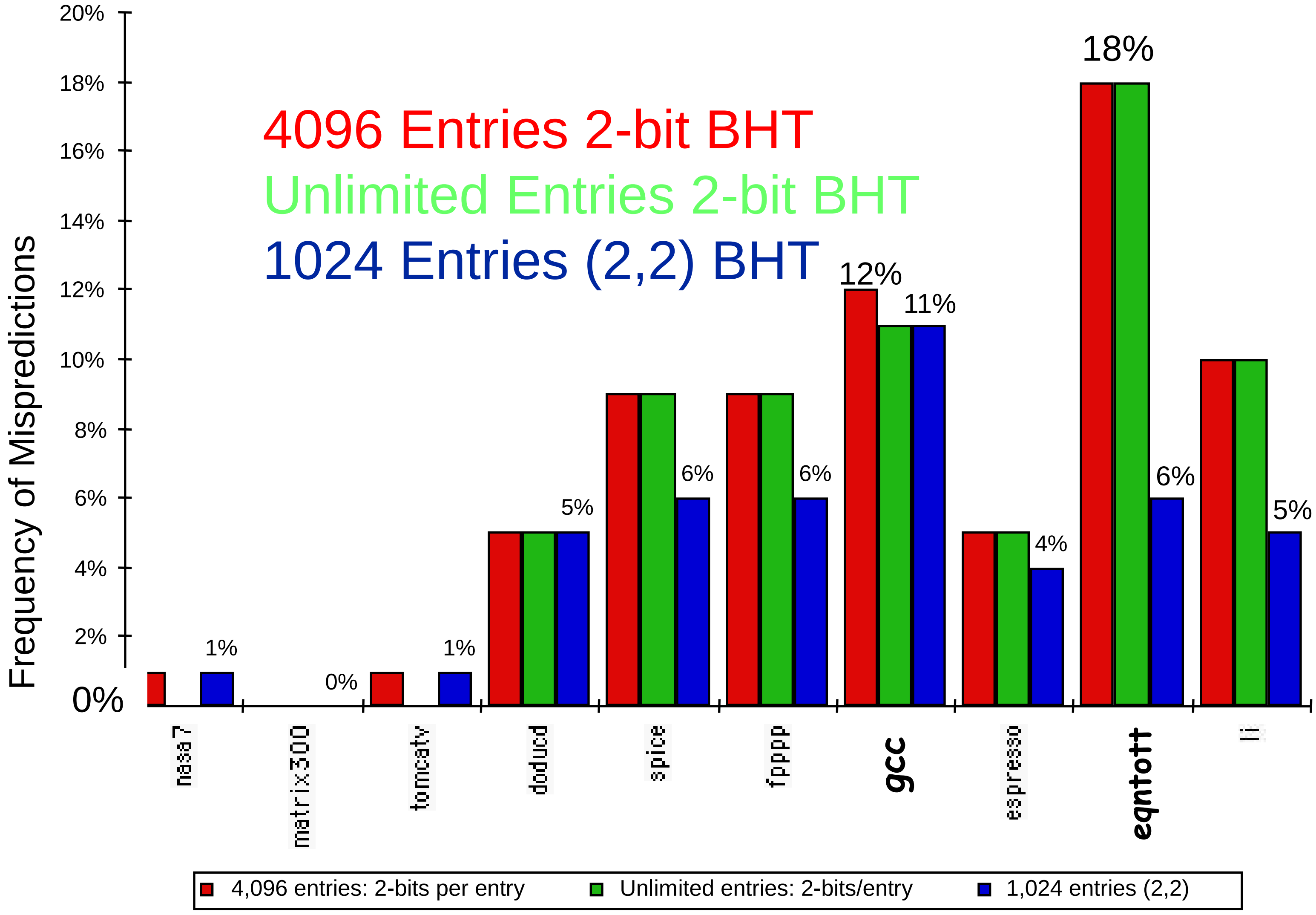
Actual: N N T T N N T T N
State: N* N* N* N*T T* T*N N* N*T T*N
Predicted: N N N N T T N N T

Scheme 2

Dynamic Branch Prediction

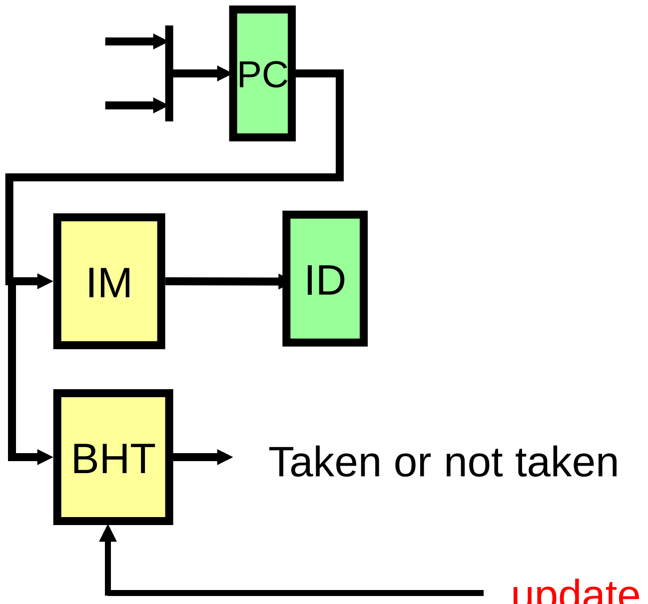
- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains a prediction bit passed to the ID stage through the IF/ID pipeline register
  - Prediction bit may predict incorrectly but the doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Accuracy of Different Schemes



Dynamic Predictor (Local)

- Predict branch based on past history of branch
- Branch history table
  - indexed by PC (or fraction of it)
  - stores last direction each branch went
  - may indicate if last instruction at this address was a branch
  - table is a cache of recent branches
  - Buffer size of 4096 entries are common
- What happens if:
  - Don't find PC in BHT? update
  - Run out of BHT entries?



Two advanced Branch Prediction

- Correlating predictor: a branch predictor that selects a local behavior of a particular branch among its 2^k choices by using a global (taken or not taken) behavior of most recently executed k branches as an index.
  - Exploits temporal correlation of a specific branch
- Tournament branch predictor: a branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.
  - Three predictors: a selection predictor, a global predictor, and a local predictor

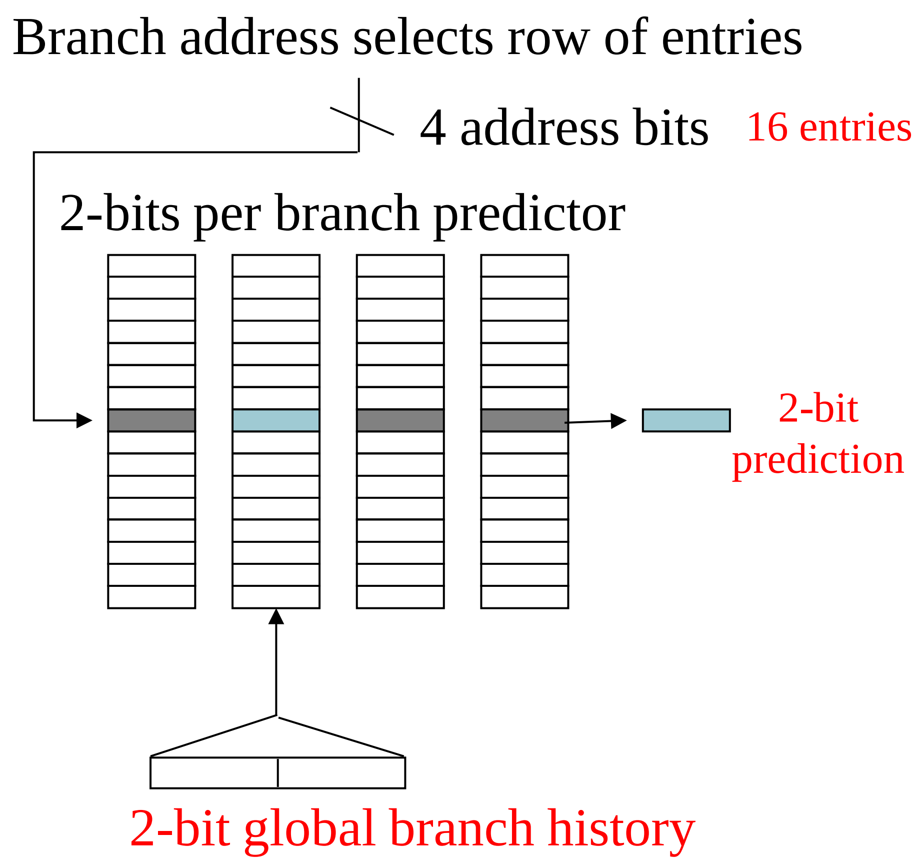
Correlating Branch Prediction

- Idea: record *k* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table
- Global Branch History: *k*-bit shift register keeping Taken/Not_Taken status of last *k* branches anywhere.
- In general, (*k*,*n*) predictor means use record of last *k* global branches to select between 2^k local branch history tables, each with *n*-bit counters
  - Thus, the old 2-bit BHT is a (0,2) predictor
- Each entry (row for given set of PC address bits) in table has 2^k *n*-bit predictors.

Correlating Branch Predictors

A (2,2) predictor with 16 sets of four 2-bit predictions

- Behavior of most recent 2 branches selects between four predictions for next branch, updating just that prediction



Tournament Branch Predictors

- A tournament (selection) predictor using, say, 4K 2-bit counters indexed by the LBS 12 bits of a local branch address, which chooses between a global predictor and a local predictor.
- A global predictor
  - 4K entries index by the history of last 12 branches ($2^{12} = 4K$)
  - Each entry is a standard 2-bit predictor
- A local predictor that has two levels
  - Top level (a local history table): 1K 10-bit entries recording last 10 branch comes, indexed by the LBS 10 bits of a branch address
  - Next level (a local predictor table): The pattern of the last 10 occurrences of that particular branch is used to index a table of 1K entries with 3-bit saturating counters
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits
- (~180K transistors)

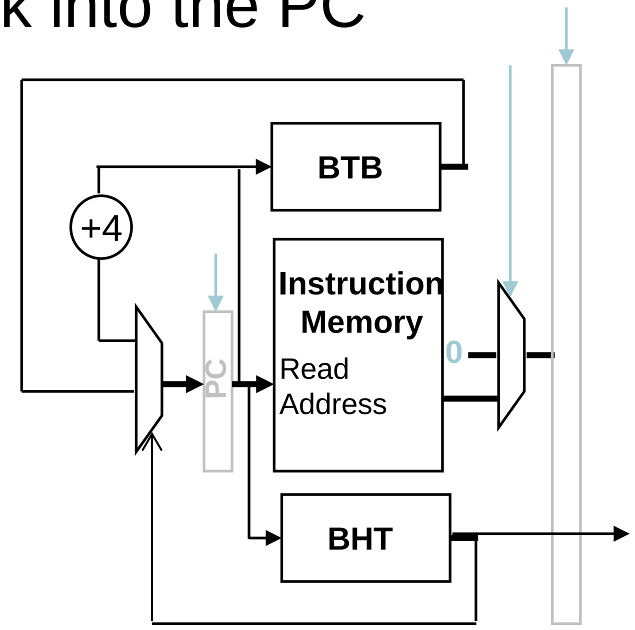
Branch target buffer

- A branch predictor tells us whether a branch is taken or not
  - Still requires the calculation of the branch target
  - 1-cycle penalty for a taken branch
- In a high-performance pipeline, especially one with multiple-issue, predicting branch well is not enough
- Branch target cache
  - Cache of branch target addresses
  - Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately with zero penalty

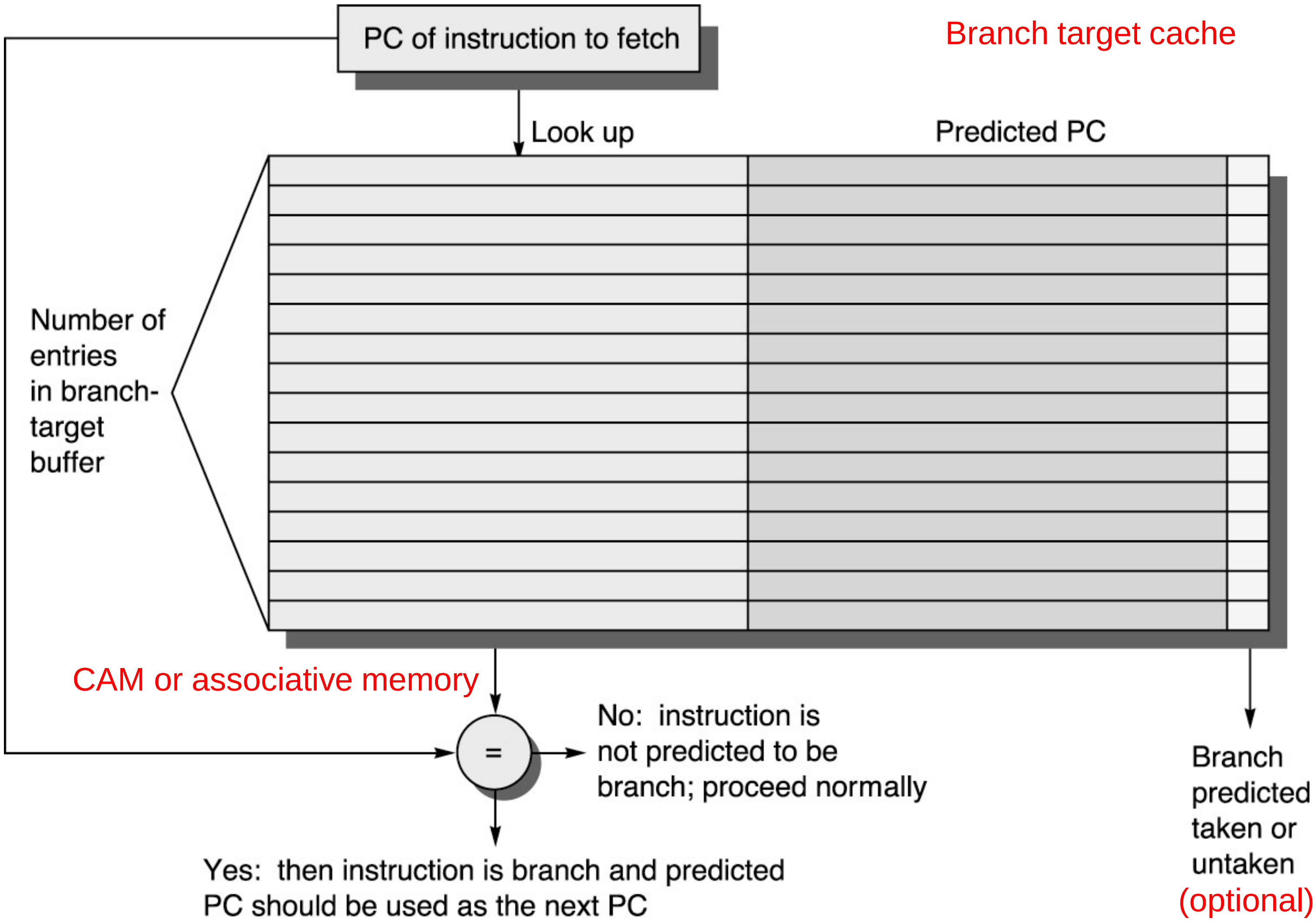
Branch Target Buffer

- The predictor predicts *whether* a branch is taken, but does not tell *where* it is taken to!
- A branch target buffer (BTB) in the IF stage can cache the branch target address
  - The branch predictor controls whether the BTB address or PC+4 is loaded back into the PC

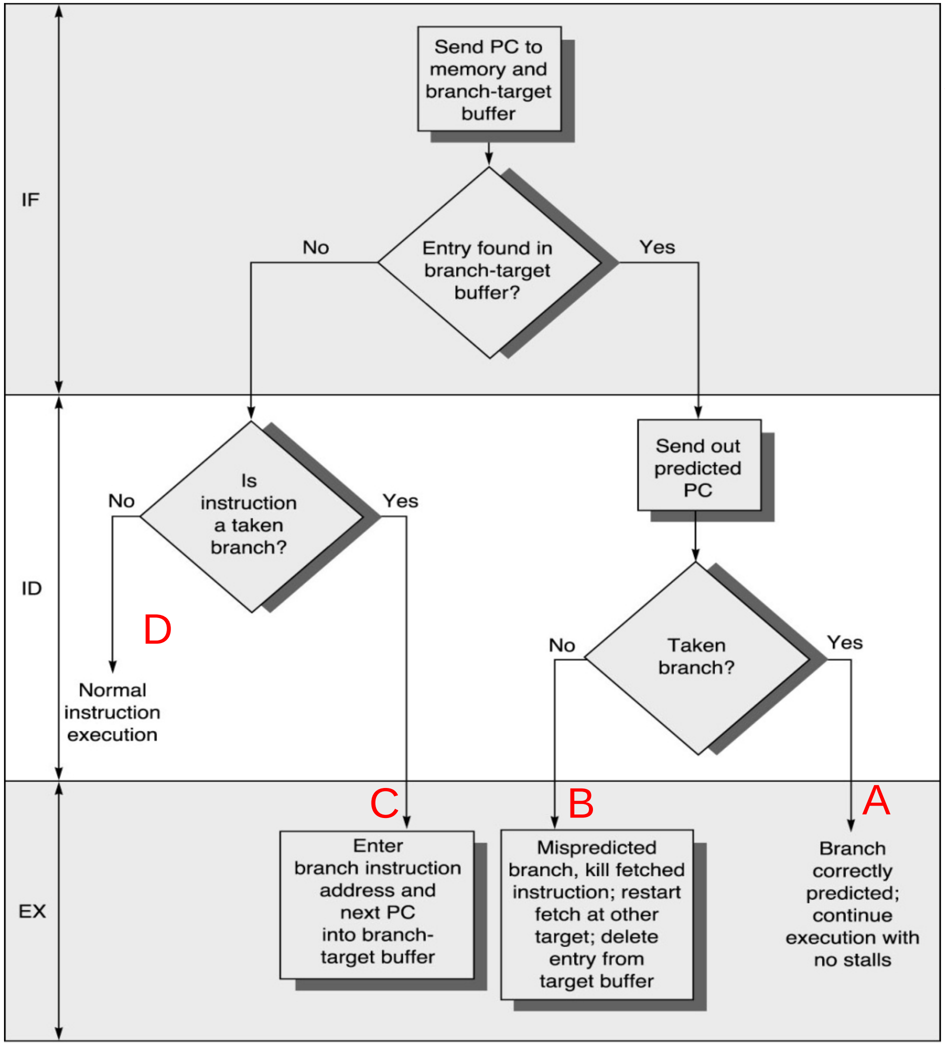
- If the prediction is correct, stalls can be avoided no matter which direction they go



BTB: store only taken branches



BTB operations



A. BTB hit, branch taken→ no penalty

B. BTB hit, misprediction → branch penalty exists (?) (flush the fetched instruction, restart fetch not taken instruction, delete the entry from BTB)

C. BTB miss, branch taken → branch penalty exists (?) (detect target at the ID stage and update BTB by entering the branch instruction address and its target address into BTB after the ID stage)

D. BTB miss, not taken → 0 cycle penalty

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception: arises within the CPU
  - e.g., undefined opcode, overflow, syscall, …
- Interrupt: from an external I/O controller
- Hardware malfunctions: either interrupt or exception
- It is hard to deal with them without sacrificing performance.

Causes of Exceptions

- Asynchronous: an *external interrupt*
  - input/output device service request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal exception (traps)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults (not in main memory), TLB misses, protection violations
  - *software exceptions:* system calls (jumps into kernel)

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
  - To help OS take the appropriate action
- Save indication of the problem
  - In MIPS: Cause register (a status register)
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Transfer control to the OS
  - In MIPS: Jump to handler at 8000 00180
  - A single entry point for all exceptions
  - The OS decodes the status register to find the cause

OS needs to know two things

- Instruction that caused the exception
  - In MIPS: Exception Program Counter (EPC)
- Reason for the exception
  - In MIPS: Cause register (a status register)
- Two main methods used to communicated the reason for an exception
  - Using a status register, like in MIPS, which holds a field that indicate the reason for the exception
  - Using a distinct address for a cause of the exception.

    (vectored interrupts)

Vectored Interrupts

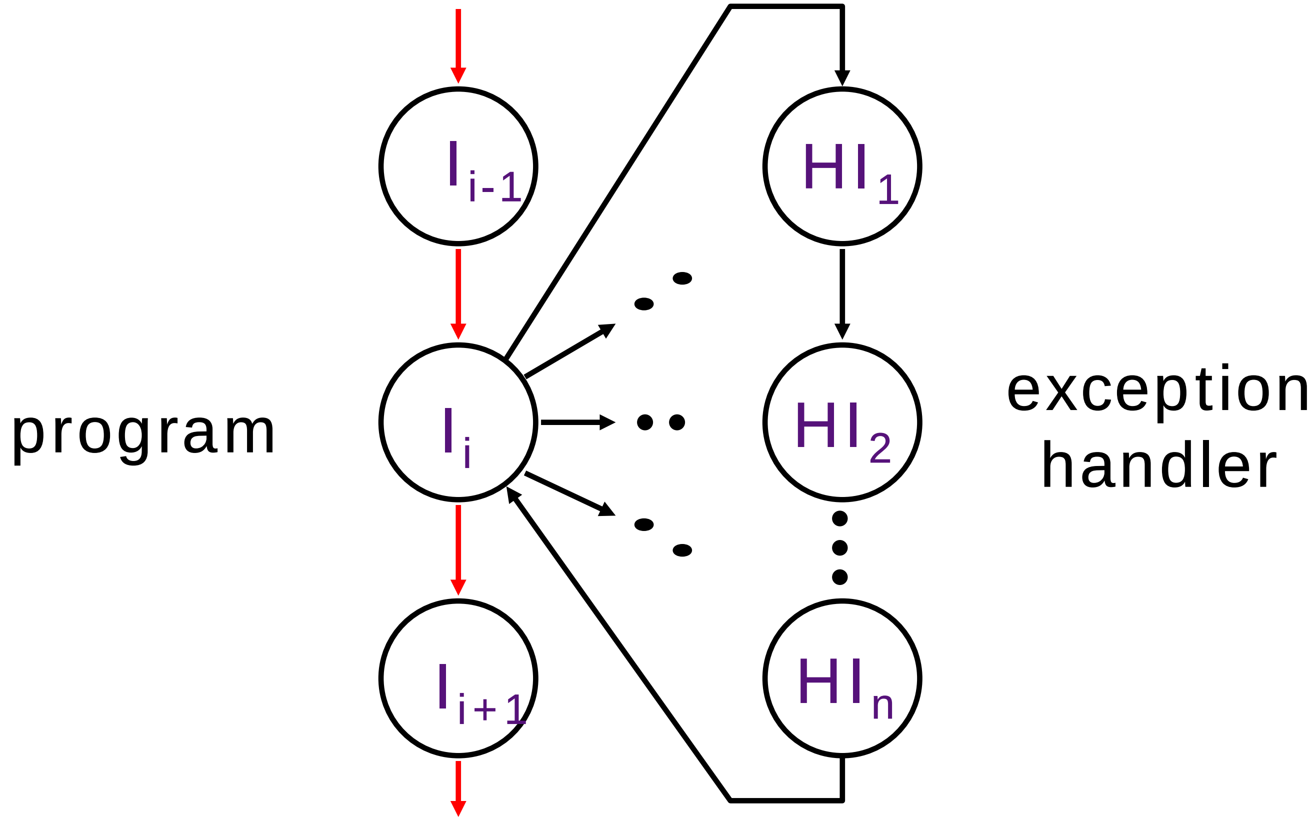
- The address to which the control is transferred is determined by the cause of the exception
  - The addresses are separated by 8 instructions (32 bytes)
- Example:
  - Undefined opcode: 8000 0000
  - Overflow: 8000 0180
- Instructions either
  - Deal with the interrupt, or
  - Jump to a real handler
- When the exception is not vectored, a single entry point for all exception is used, and the OS decodes the status register to find the cause.

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
  - If restartable
    - Take corrective action
    - use EPC to return to program
  - Otherwise
    - Terminate program
    - Report error using EPC, cause, …

Exceptions: altering the normal flow of control

An *exception* transfers control to a special handler code run in privileged mode. Exceptions are usually unexpected or rare from program's point of view.



Interrupts: invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} (a precise interrupt)
  - It saves the PC of instruction I_i in the EPC
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

A MIPS Interrupt Handler Code

- Saves EPC before re-enabling interrupts, if needed, to allow nested interrupts $\Rightarrow$
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) to resume user code, this:
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Exception

- A synchronous exception is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - syscall is a special jump instruction involving a change to privileged kernel mode
  - Handler resumes at instruction after system call

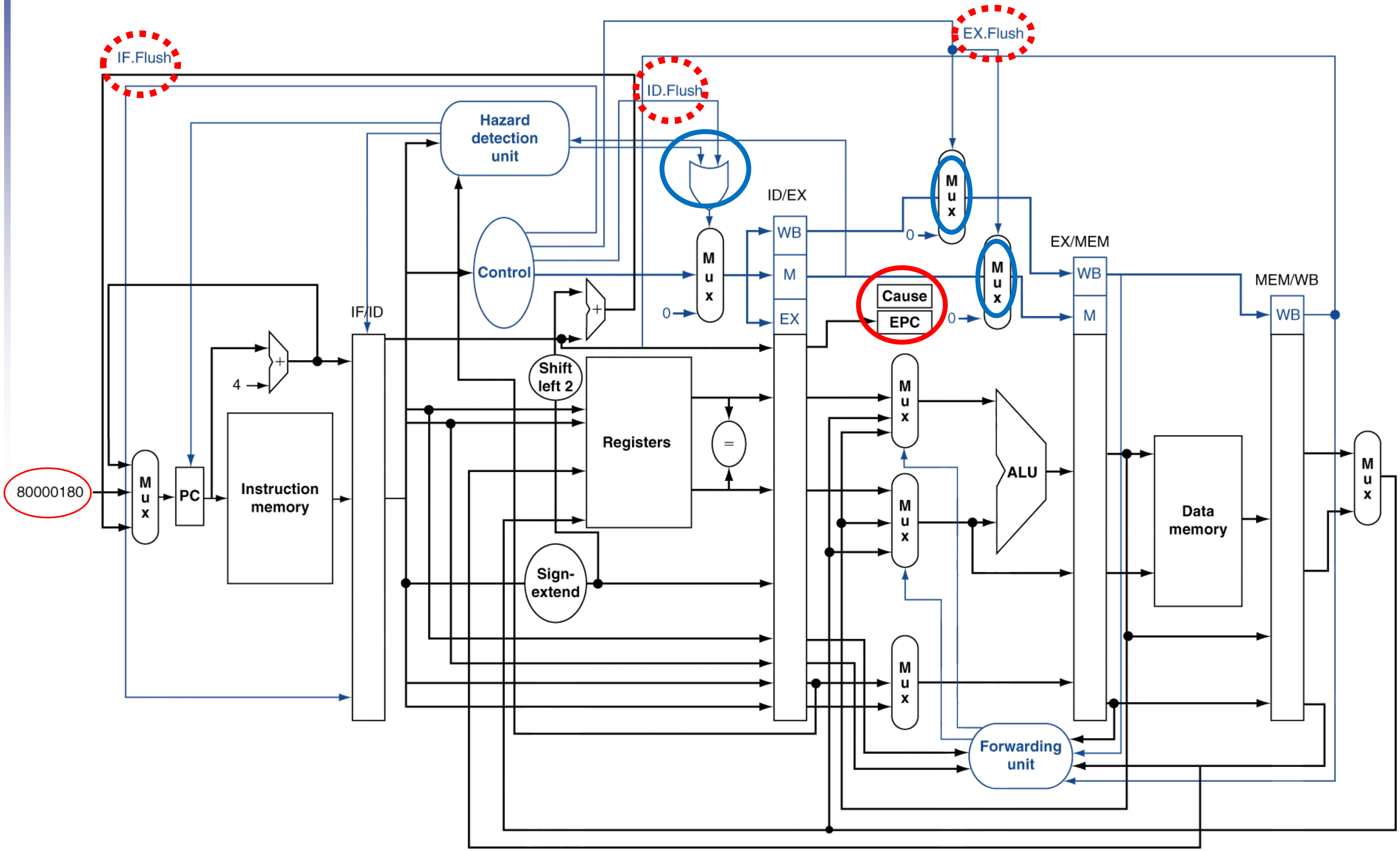
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

```
add $1, $2, $1
```

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust by subtracting 4

Exception Example

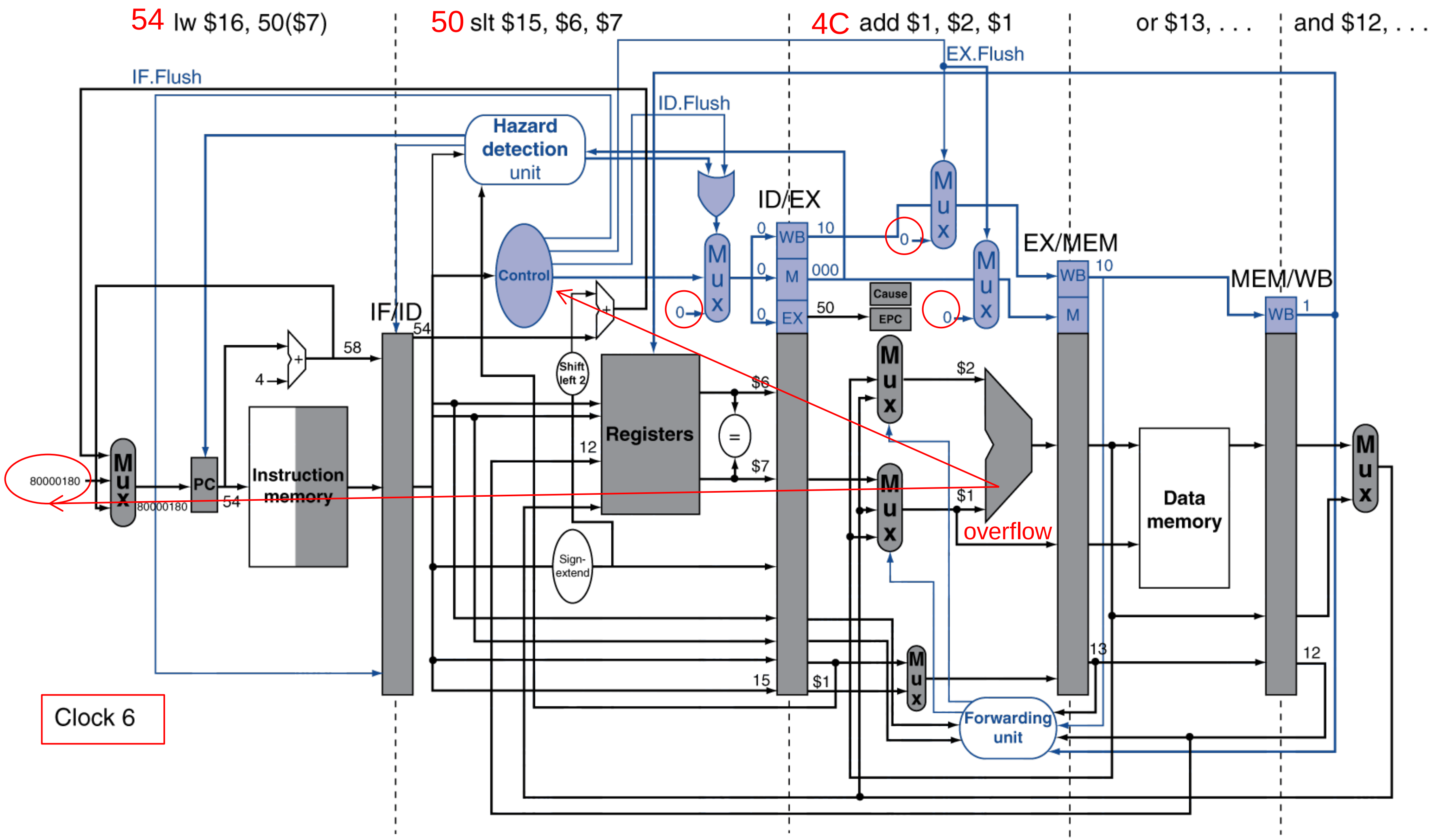
- Overflow exception on add in

```
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)
…
```

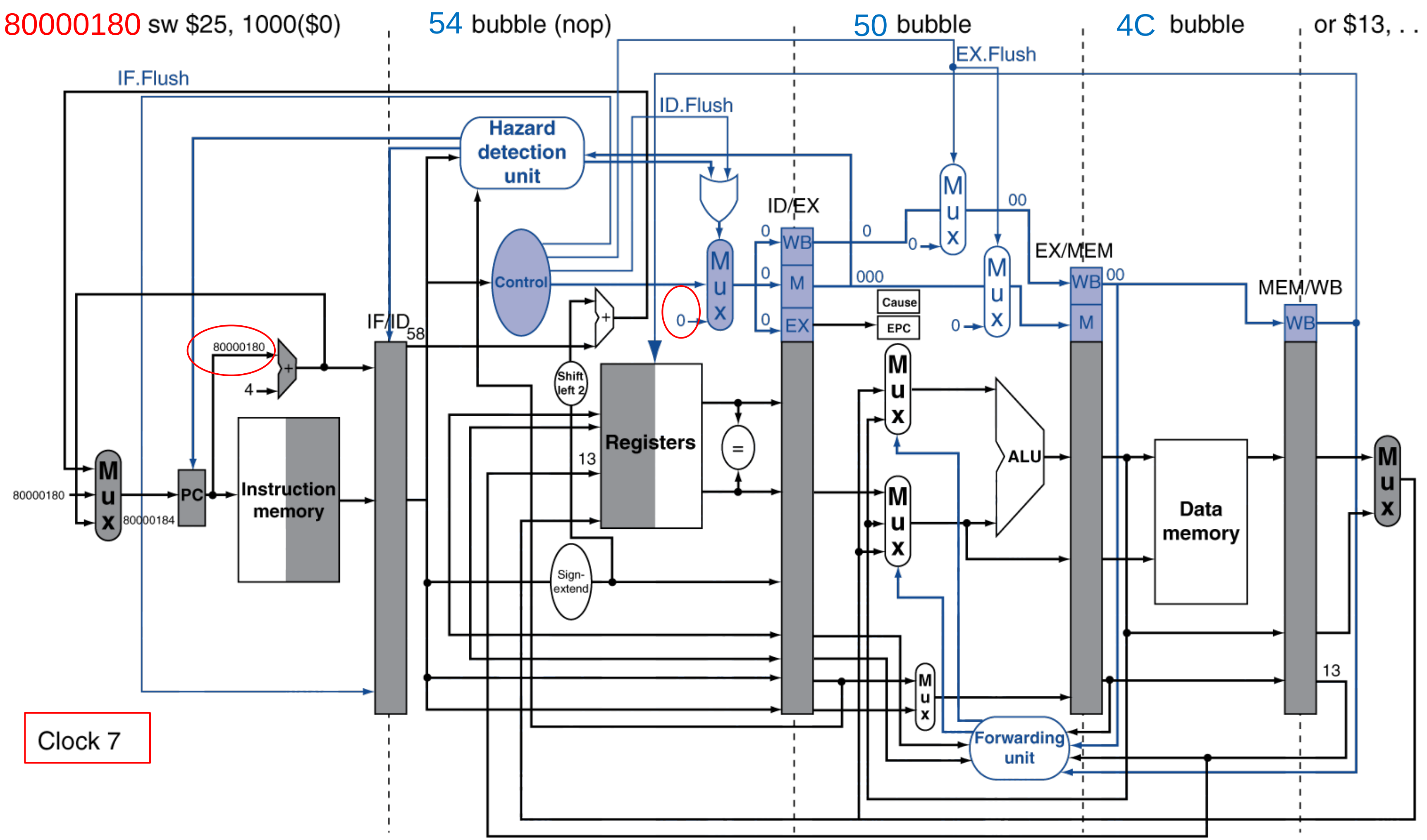
- Handler // overflow

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

Exception Example



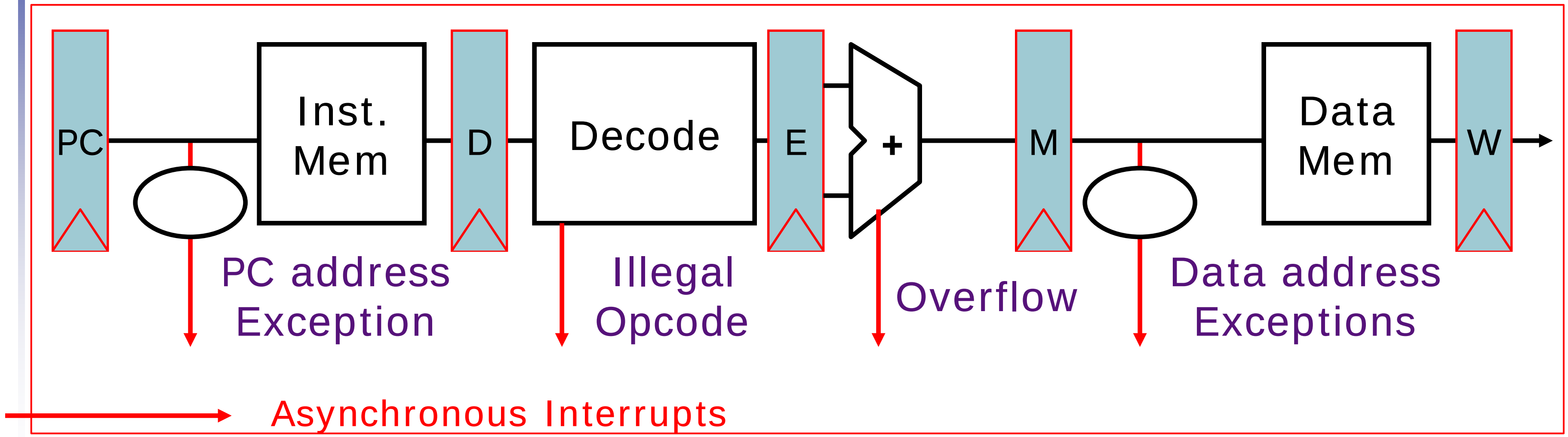
Exception Example



Multiple Exceptions

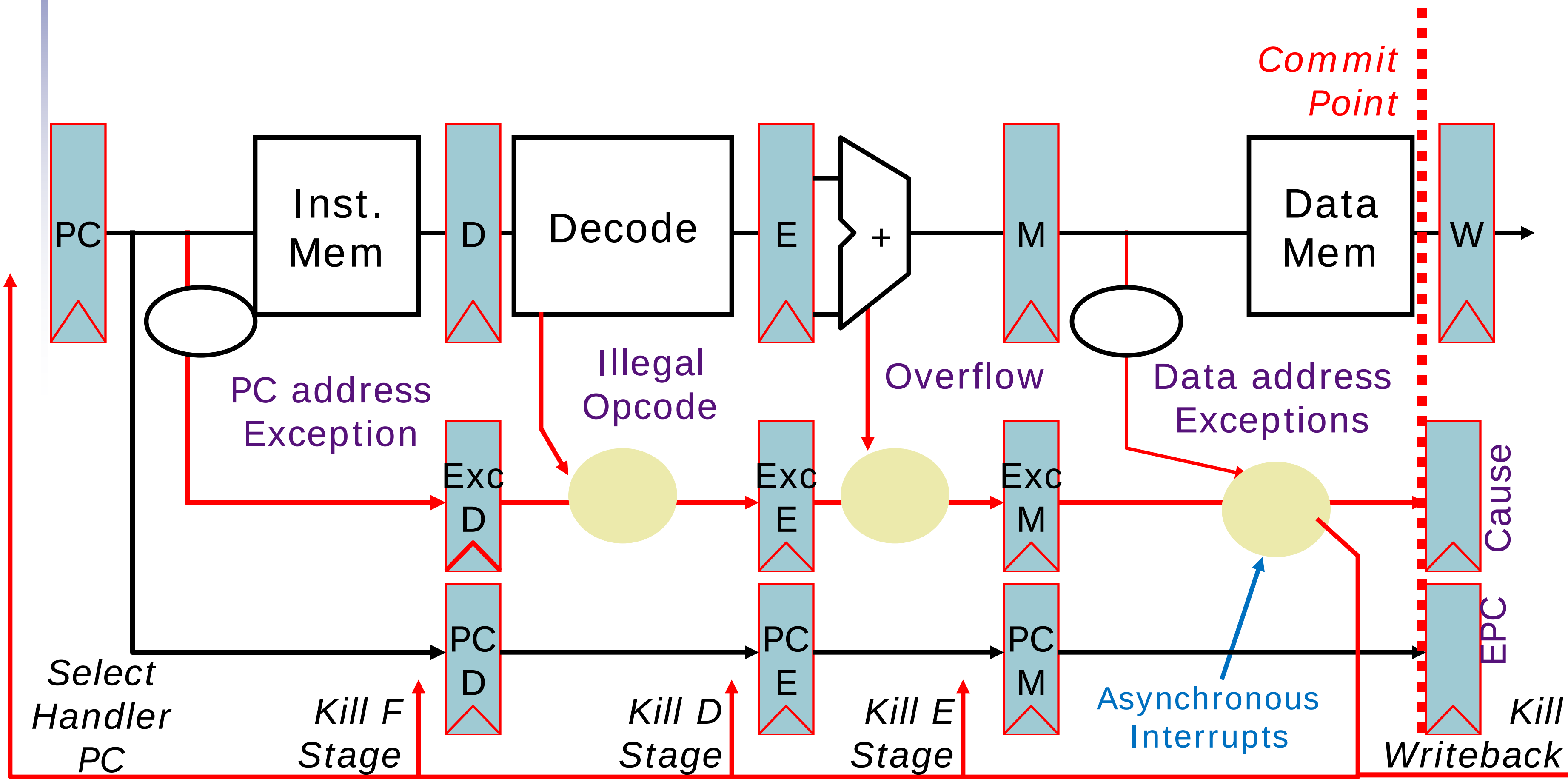
- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Exception Handling



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?
- Exceptions raised from memory access account for 6 out of 8 cases

Exception Handling



Exception Handling

- Hold exception flags in pipeline until commit point (MEM stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Exception Pipeline Diagram

Overflow! Commit point

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	nop				
(I_2) 100: XOR		IF_2	ID_2	EX_2	nop	nop			
(I_3) 104: SUB			IF_3	ID_3	nop	nop	nop		
(I_4) 108: ADD				IF_4	nop	nop	nop	nop	
(I_5) Exc. Handler code					IF_5	ID_5	EX_5	MA_5	WB_5

Resource Usage

	time t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	I_3	nop	I_5			
EX			I_1	I_2	nop	nop	I_5		
MEM				I_1	nop	nop	nop	I_5	
WB					nop	nop	nop	nop	I_5

Speculating on Exceptions

- Prediction mechanism
  - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline
- Flushing is required because bypassing allows use of uncommitted instruction results by following instructions

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- It is not feasible for complex multiple-issue out-of-order pipelines
- Precise exceptions is required to support virtual memory (in chapter 5)

MIPS 4K Processor

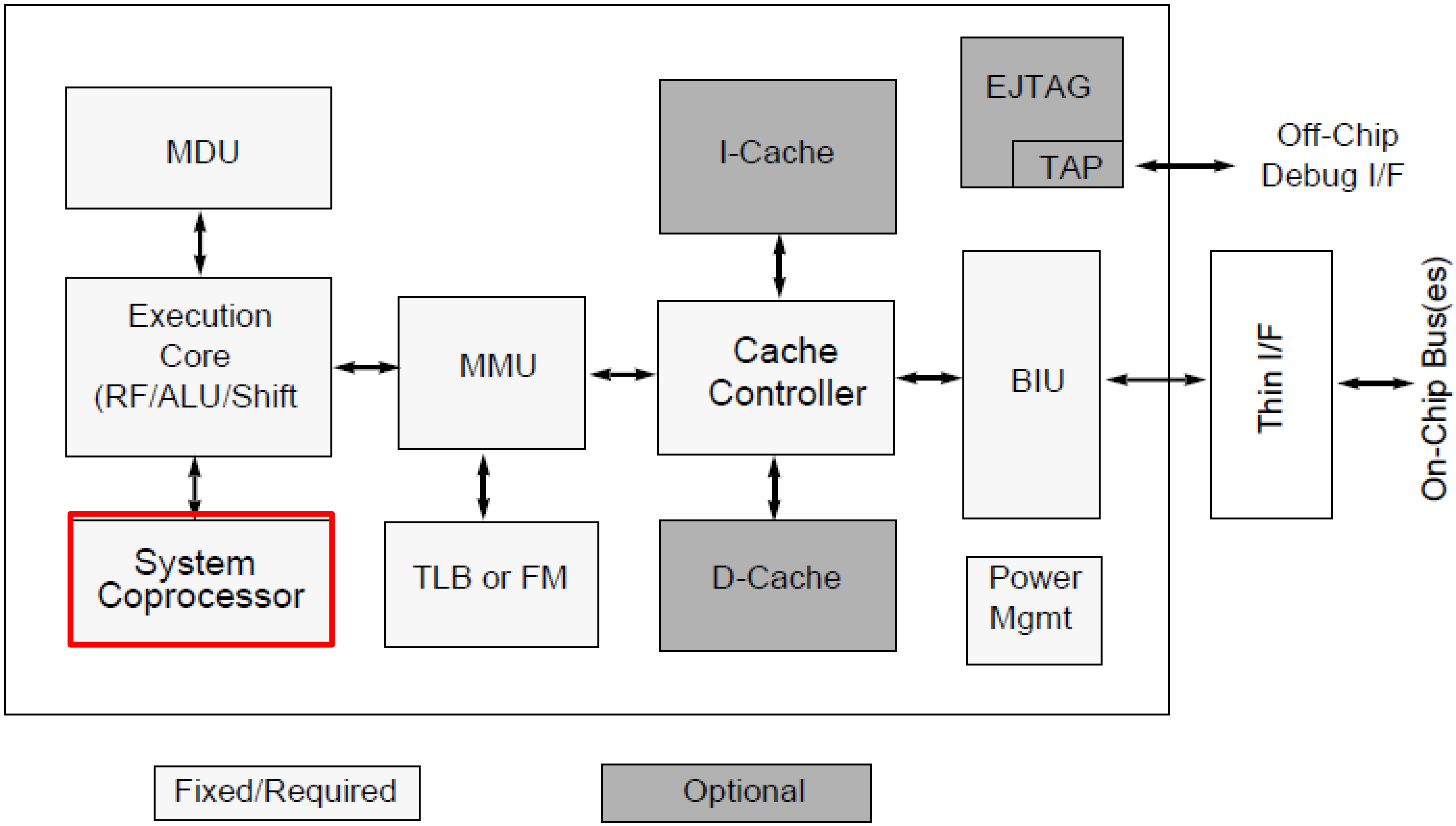
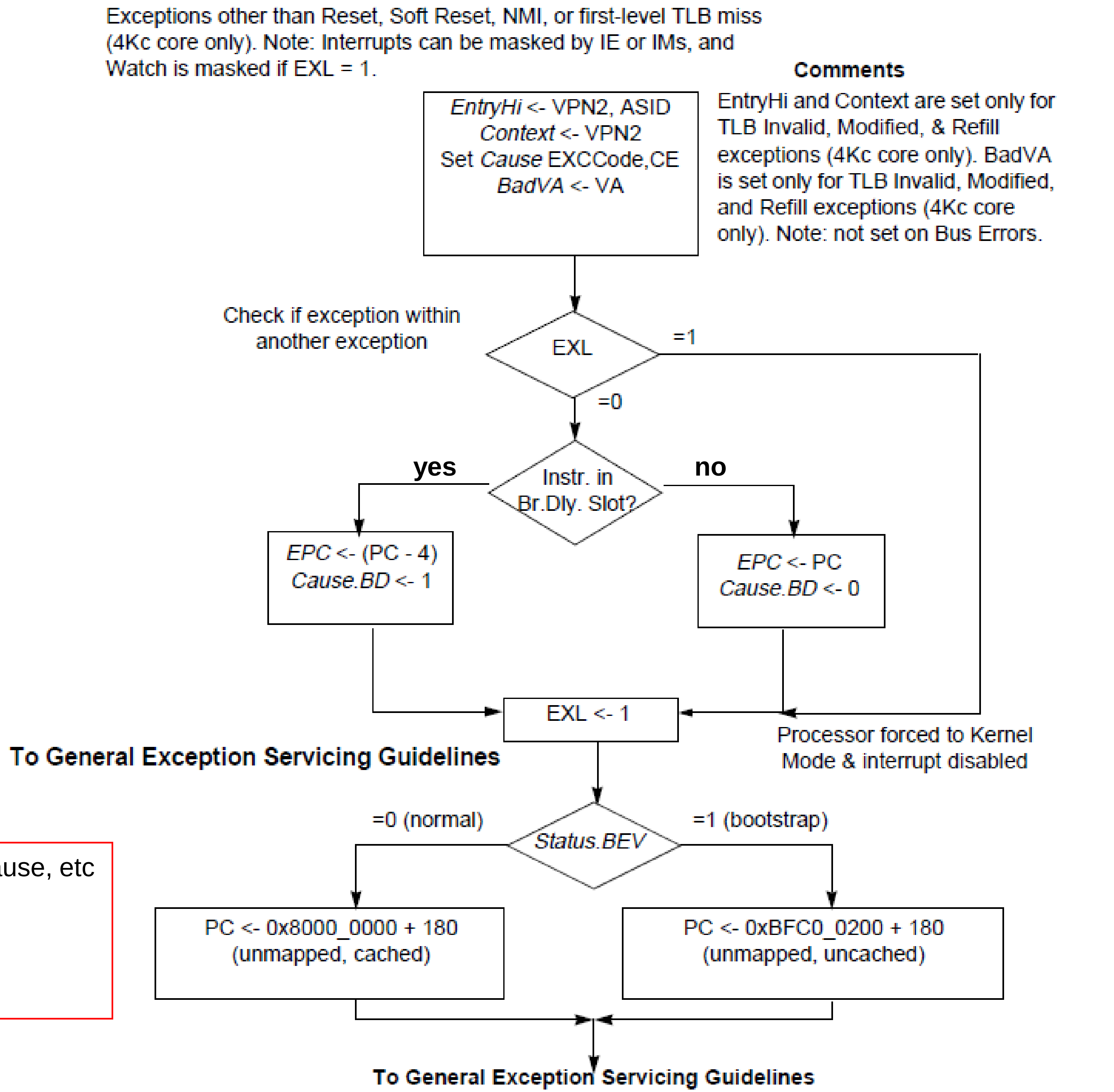


Figure 1-1 4K Processor Core Block Diagram

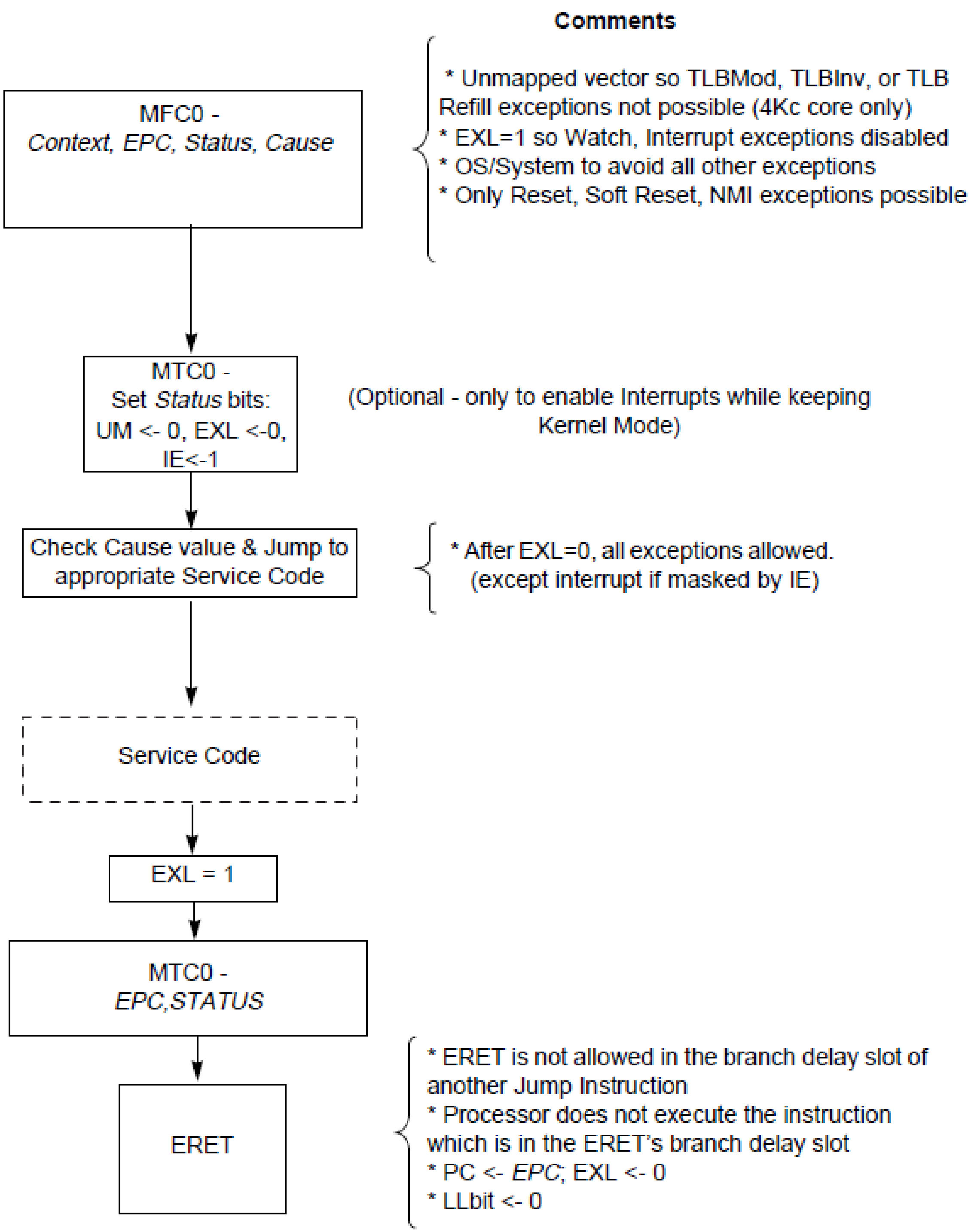
System Control Coprocessor

- CP0 is responsible for
  - virtual-to-physical address translation (TLB),
  - cache control,
  - the exception control system,
  - diagnostic capability,
  - operating mode selection (kernel vs. user mode),
  - enabling/disabling interrupts,
  - configuration such as cache size and associativity
- 32 32-bit CP0 registers

General Exception Handler (HW)



1. The processor records EPC, Cause, etc
2. Change to the kernel mode.
3. Interrupt disable
4. Jump to the exception handler



General Exception Handler (SW)

1. Save some registers on the system stack
2. Check the exception type (ExcCode)
3. Use ExcCode with a jump table to go to a correct service location
4. Execute the service code
5. Restore registers saved in step 1
6. Atomically
   * restore previous kernel/user mode
   * re-enable interrupts
   * jump back to the user program