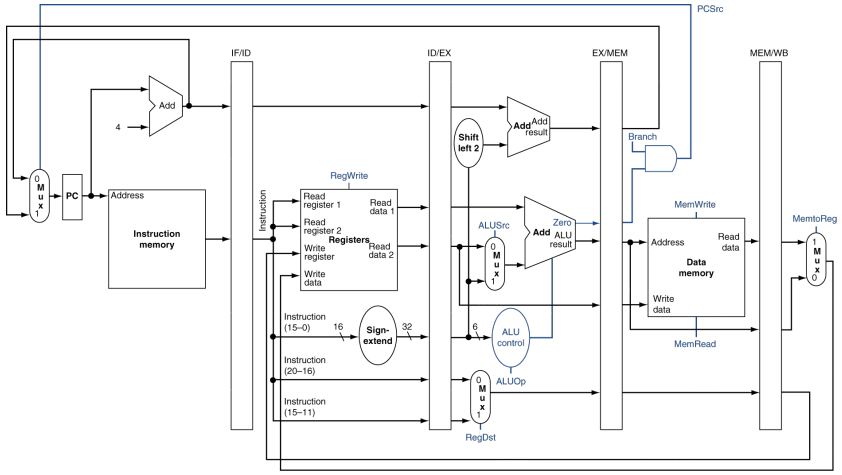
Computer Organization & Architecture
计算机组成与结构
Chapter 4.8 Branch Prediction

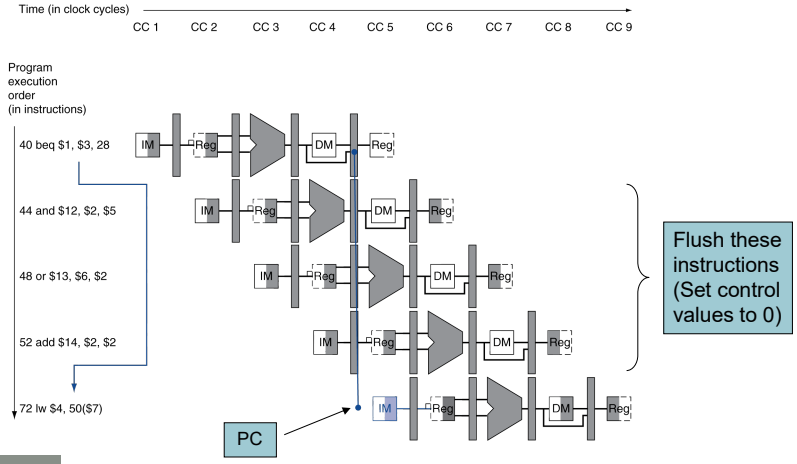
Dr. Qingfeng (Karen) Zhuge, Professor
College of Computer Science and Software Engineering
East China Normal University

Pipelined Control (Simplified)



Branch Hazards

- If branch outcome determined in MEM

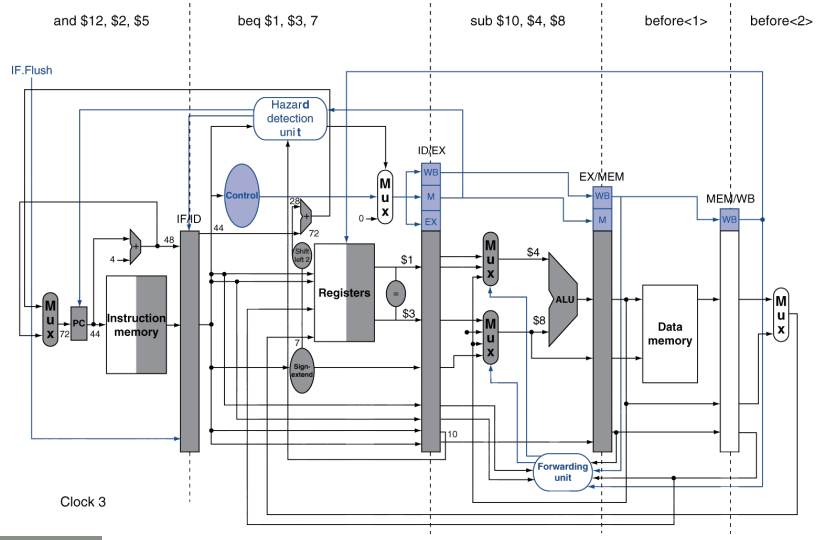


Reducing Branch Delay

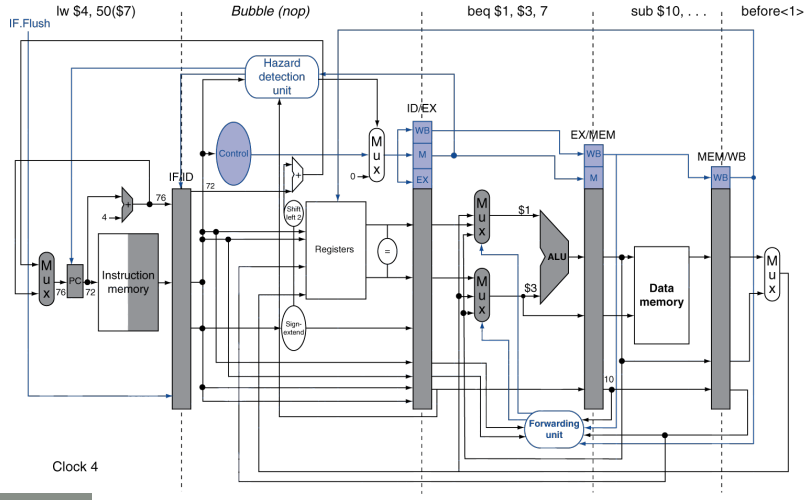
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
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)
```

Example: Branch Taken



Example: Branch Taken



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
…
beq $1, $4, target
```

IF ID EX MEM WB

- Can resolve using forwarding

Data Hazards for Branches

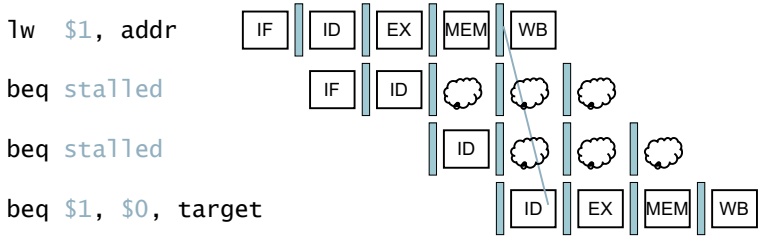
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

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

IF ID EX MEM WB

Data Hazards for Branches

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



Branch (Control) Hazards and Their Impact on the Pipeline Performance

- Control hazards can cause a greater performance loss for a pipelined implementation than data hazards. In first approximation:

$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + [\ \text{branchFrequency} \times \text{branchPenalty}\ ]}$$

  - performance loss can vary between 10% and 30% depending on the branch frequency
  - generally, the deeper the pipeline, the worse the branch penalty (measured in clock cycles)
- Various strategies to reduce the branch penalty
  - static (compile time) schemes
    - they are fixed for each branch during the entire execution
  - dynamic
    - hardware and software techniques
    - more sophisticated branch prediction schemes

Reducing Pipeline Branch Penalties - I: Freezing (Flushing) the Pipeline

- Assuming now that the target address calculation and the comparison are performed during the ID stage, then, in case of "branch taken", the PC is changed at the end of the ID stage
  - simplest strategy: freezing the pipeline as soon as the instruction is *detected as a branch* and regardless of its outcome (branch penalty is fixed, SW cannot reduce it)

instruction	1	2	3	4	5	6	7	8	9
BNEZ R5, target	IF	ID							
fall-through inst		IF							
branch target			IF	ID	EX	MEM	WB		
branch target + 1				IF	ID	EX	MEM	WB	
branch target + 2					IF	ID	EX	MEM	WB

Reducing Pipeline Branch Penalties - II: Predicting "Non-Taken"

- If the branch is taken, all the fetched instruction must be turned into NOPs and it is necessary to restart the fetch at the target address. The state must be unaffected!!

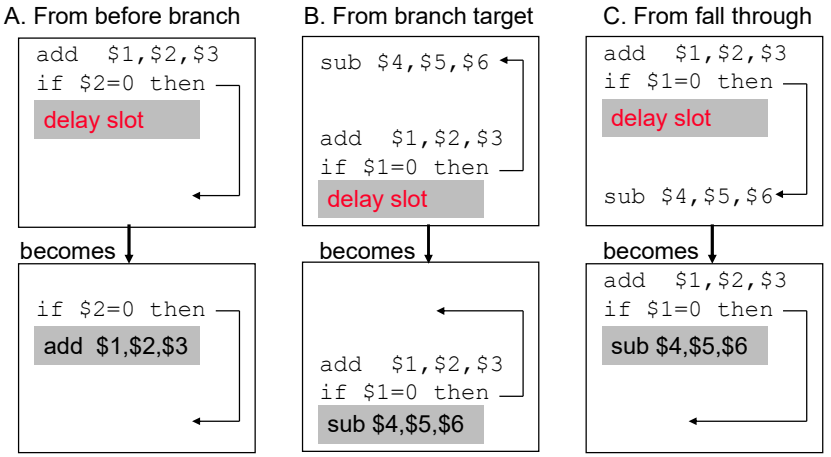
instruction	1	2	3	4	5	6	7	8	9
untaken branch	IF	ID							
Fall-Through Inst.		IF	ID	EX	MEM	WB			
FTI + 1			IF	ID	EX	MEM	WB		
FTI + 2				IF	ID	EX	MEM	WB	
FTI + 3					IF	ID	EX	MEM	WB

instruction	1	2	3	4	5	6	7	8	9
taken branch	IF	ID							
Fall-Through Inst.		IF							
branch target			IF	ID	EX	MEM	WB		
branch target + 1				IF	ID	EX	MEM	WB	
branch target + 2					IF	ID	EX	MEM	WB

Delayed Branches

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Basic Branch Prediction: Branch-History Table (or Branch-Prediction Buffer)

Branch-History Table

- 1-bit prediction scheme
  - remembers last outcome of some branches
- implemented as a small memory indexed by a lower portion (some least-significative bits) of the address of the branch instruction
  - *like a direct-map cache where an access is always a hit*
- useful only to reduce the branch delay when longer than the time to compute the possible target PC
- We don't even know if prediction is correct. It is just a hint (if wrong the bit is inverted and we restart from the correct PC)
  - e.g., branch instructions at the following addresses share the BHT entry
    - *00F2BC* 0101 1100
    - *010A5D* 1001 1100

Index	Taken?
0000	1
0001	0
0010	1
0011	0
0100	1
0101	0
0110	1
0111	1
1000	1
1001	0
1010	1
1011	1
1100	0
1101	1
1110	1
1111	1

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - 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

1-Bit Predictor: Shortcoming

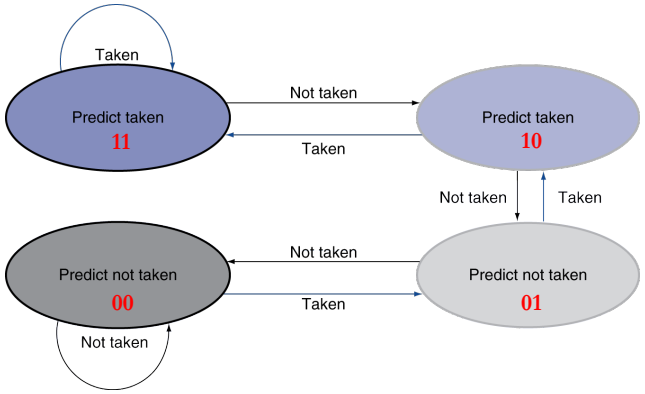
- Inner loop branches mispredicted twice!

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

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Example: Accuracy of 1-Bit Prediction Scheme in the Presence of "Loop Branching"

```
loop: L.D F0, 0(R1)
      MUL.D F4, F0, F2
      S.D F4, 0(R1)
      DADDIU R1, R1, #-8
      BNEZ R1 loop
```

- Assumptions
  - R1 is initialized to #80

Iteration	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7	8	9	10
Predicted behavior	NT	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T
Actual behavior	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T	NT

- Branch taken 90% of the times, but branch prediction accuracy is only 80%
- A 1-bit predictor for "loop branches" mispredicts at twice the rate that the branch is not taken

Example: 2-Bit Prediction Scheme in Action with "Loop Branching"

```
loop: L.D F0, 0(R1)
      MUL.D F4, F0, F2
      S.D F4, 0(R1)
      DADDIU R1, R1, #-8
      BNEZ R1 loop
```

- Assumptions
  - R1 is initialized to #80

Iterationss & steps	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7	8	9	10
Predicted behavior	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
Actual behavior	T	T	T	T	T	T	T	T	T	T	NT	T	T	T	T	T	T	T	T	T	T	NT

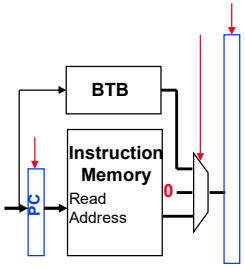
- Branch taken 90% of the times, and branch prediction accuracy is now 90%
- The 2-bit predictor mispredicts at the 10th step of the 1st iteration, but *doesn't change its mind.* It just moves to "weak-predict-taken" for the 1st step of the following iteration

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Branch Target Buffer

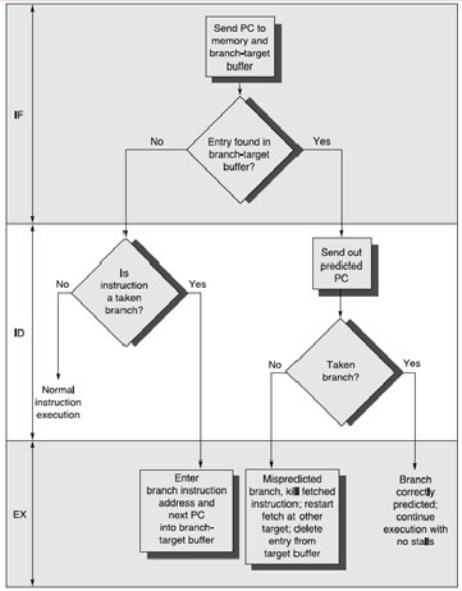
- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - Would need a two read port instruction memory
  - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



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

Predicting the Instruction Target Address: Branch-Target Buffer (or Branch-Target Cache)

- Goal: learn the **predicted instruction address** at the end of IF stage
  - one cycle earlier
    - at best, branch-prediction buffers know the next predicted instruction at the end of ID
- No branch delay
  - if entry found in buffer and prediction is correct
  - if entry not found and branch is not taken
- Two cycle penalty
  - if prediction is incorrect
  - if entry is not found and branch is taken



Branch Target Buffer: Penalty Table

Instruction in buffer	Prediction	Actual Branch	Penalty Clock Cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

- Assuming
  - 90% buffer hit rate, 85% prediction accuracy, 60% branches taken
- P(branch in buffer but not taken) = 0.9 x 0.15 = 0.135
- P(branch not in buffer, but taken) = 0.1 x 0.6 = 0.06
- Total branch penalty = (0.135 + 0.06) x 2 = 0.39
  - the penalty for delayed-branch was about 0.5 clock cycles
  - penalty is lower with better predictors (and bigger branch delays)

Adding "Global" Information to Branch Prediction: Correlating (or Two-Level) Branch Predictors

```
if (a == 2)                    b1   DSUBUI R3, R1, #2
  a = 0;                            BNEZ R3, L1
…                                   DADD R1, R0, R0
if (b == 2)                  L1:  DSUBUI R3, R2, #2
  b = 0;                       b2   BNEZ R3, L2
…                                   DADD R2, R0, R0
if (a != b) {                L2:  DSUBU R3, R1, R2
  …                          b3   BEQZ R3, L3
```

- Prediction accuracy can be improved by accounting for branch spatial correlations and looking at the behavior of other branches
  - e.g. in the example above, if the first two branches (b1 and b2) are not taken then the third (b3) will be taken

A Simple Example: 1-Bit Predictor

```
if (d == 0)                    b1   BNEZ R1, L1
  d = 1;                            DADDIU R1, R0, #1
if (d == 1) {                L1:  DSUBUI R3, R1, #-1
  …                          b2   BNEZ R3, L2
                                    …
                               L2:  …
```

Assumption: d alternates between 2 and 0

d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

A 1-bit predictor that is initialized to not-taken mispredicts all branches

A Simple Example: Set-Up

```
if (d == 0)                    b1   BNEZ R1, L1
  d = 1;                            DADDIU R1, R0, #1
if (d == 1) {                L1:  DSUBUI R3, R1, #-1
  …                          b2   BNEZ R3, L2
                                    …
                               L2:  …
```

init d	d==0?	b1	d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
x ≠{0,1}	no	taken	x ≠ {0,1}	no	taken

Correlation: if b1 is not taken then d is set to 1 and b2 is also not taken

A Simple Example: 1-Bit Predictor with 1-Bit Correlation {i.e., a (1,1) Predictor}

```
if (d == 0)                    b1   BNEZ R1, L1
  d = 1;                            DADDIU R1, R0, #1
if (d == 1) {                L1:  DSUBUI R3, R1, #-1
  …                          b2   BNEZ R3, L2
                                    …
                               L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken

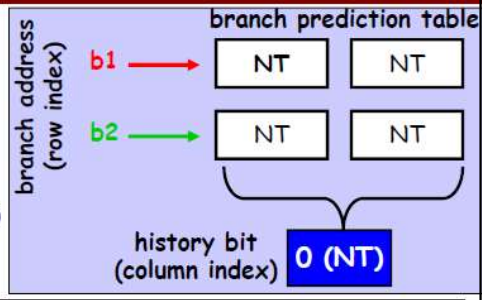
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 0

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



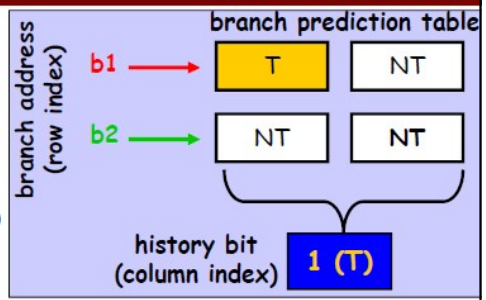
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T		NT/NT	T	
0		NT			NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 1

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



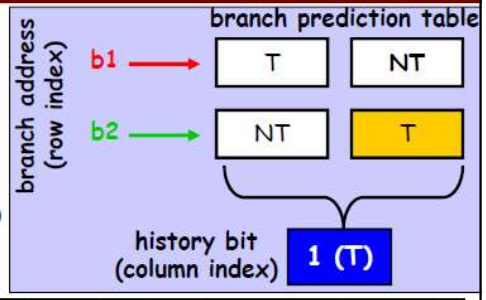
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	
0	T/NT	NT			NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 2

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



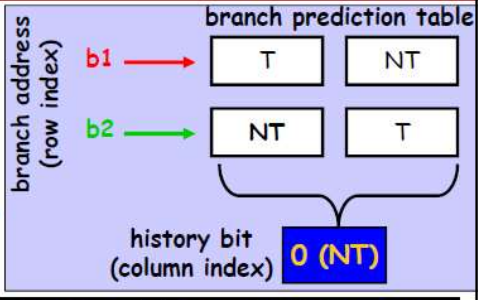
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT		NT/T	NT	
2		T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 3

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



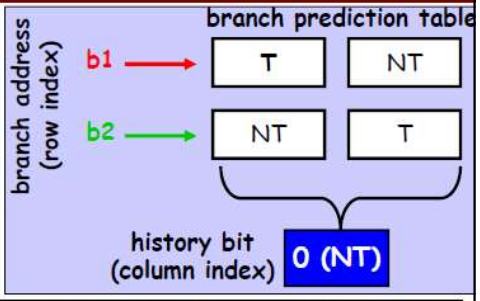
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	
2	T/NT	T			T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 4

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



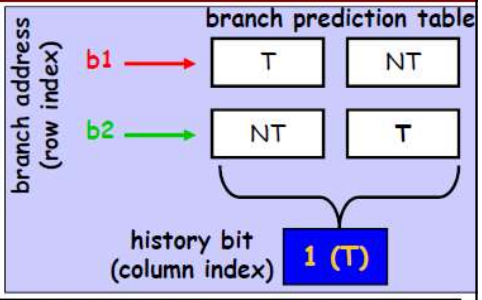
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T		NT/T	T	
0		NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 5

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



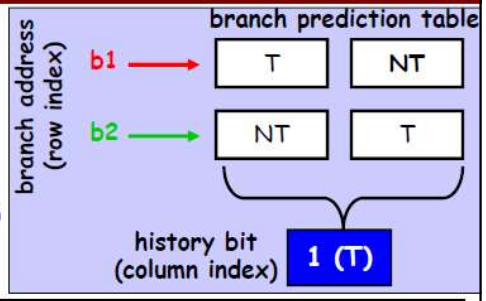
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	
0	T/NT	NT			NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 6

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



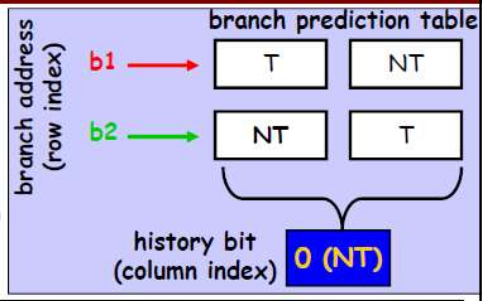
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT		NT/T	NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 7

```
    BNEZ R1, L1
    DADDIU R1, R0, #1
L1: DSUBUI R3, R1, #-1
    BNEZ R3, L2
    …
L2: …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



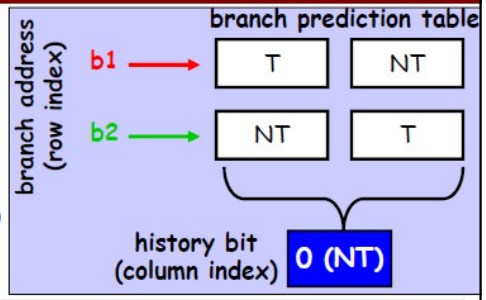
d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

A Simple Example of (1,1) Predictor: Simulation - Cycle 8

```
     BNEZ R1, L1
     DADDIU R1, R0, #1
L1:  DSUBUI R3, R1, #-1
     BNEZ R3, L2
     …
L2:  …
```

Assumption: d alternates between 2 and 0
X/Y: use X if last branch was not taken, use Y if last branch was taken



d=?	b1 pred.	b1 action	b1 new pred.	b2 pred.	b2 action	b2 new pred.
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

A (1,1) predictor initialized to NT/NT mispredicts only at the first iter.

Branch Prediction in the Alpha 21264 μP (Tournament Predictors)

- Local Prediction
  - each table entry in the first-level local-history table is a 10-bit pattern indicating the direction of the selected branch's last 10 executions. The local-branch prediction is a prediction counter bit from a second table indexed by the local-history pattern
- Global Prediction
  - the path history is a 12-bit pattern indicating the taken/not taken direction for the last 12 executed branches (in fetch order). The 21264's global-history predictor is a table of saturating counters indexed by the path history

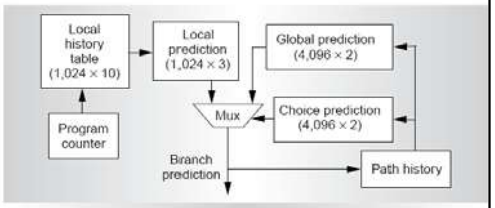
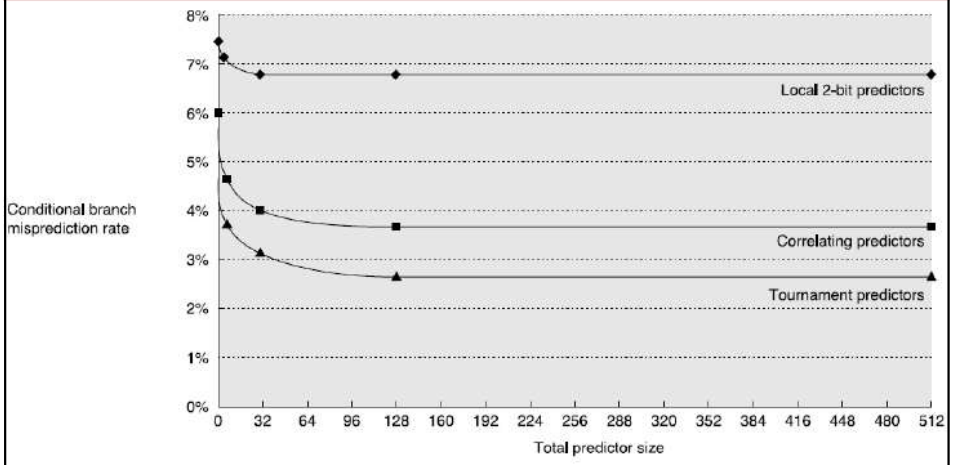


Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

Measure: Misprediction Rate for Three Predictors on SPEC89 as Total Number of Bits is Increased



- The prediction capability does not improve beyond a certain size

Branch (Control) Hazards and Their Impact on the Pipeline Performance

- Control hazards can cause a greater performance loss for a pipelined implementation than data hazards. In first approximation:

$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + [\ \text{branchFrequency} \times \text{branchPenalty}\ ]}$$

  - performance loss can vary between 10% and 30% depending on the branch frequency
  - generally, the deeper the pipeline, the worse the branch penalty (measured in clock cycles)
- Various strategies to reduce the branch penalty
  - static (compile time) schemes
    - they are fixed for each branch during the entire execution
  - dynamic
    - hardware and software techniques
    - more sophisticated branch prediction schemes

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
- Dealing with them without sacrificing performance is hard

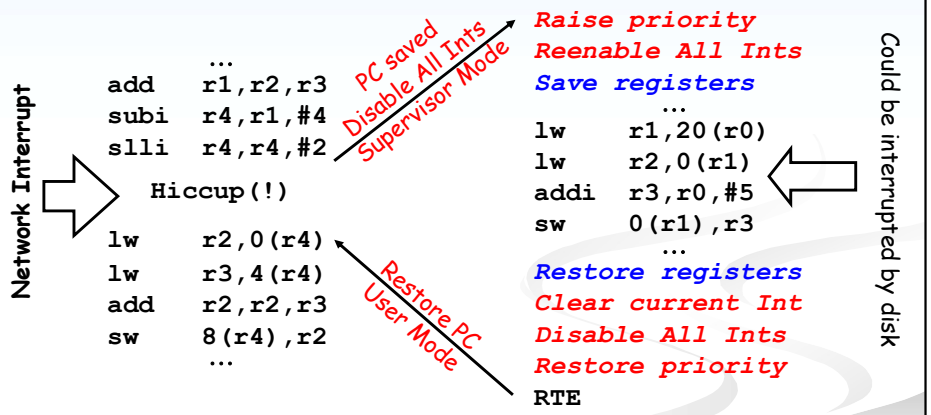
Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

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, …

Interrupt Handler



Note that priority must be raised to avoid recursive interrupts!

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

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

Exception Example

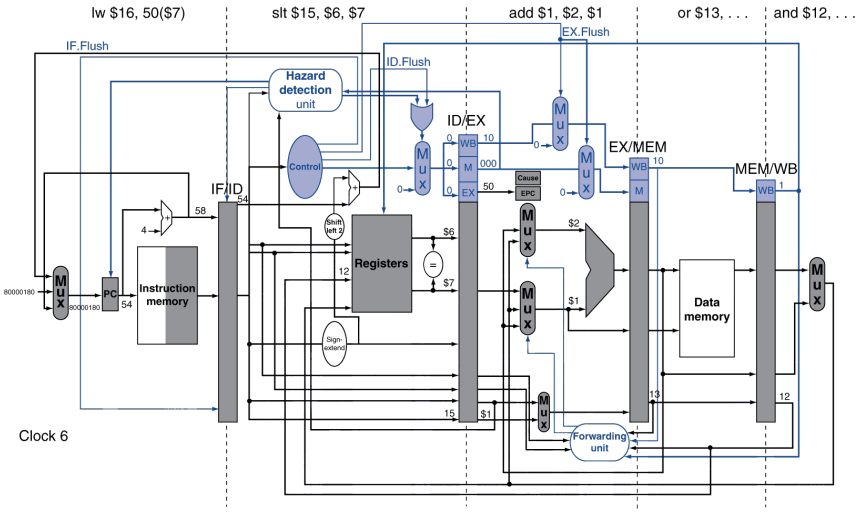
- Exception on `add` in

```
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)
…
```

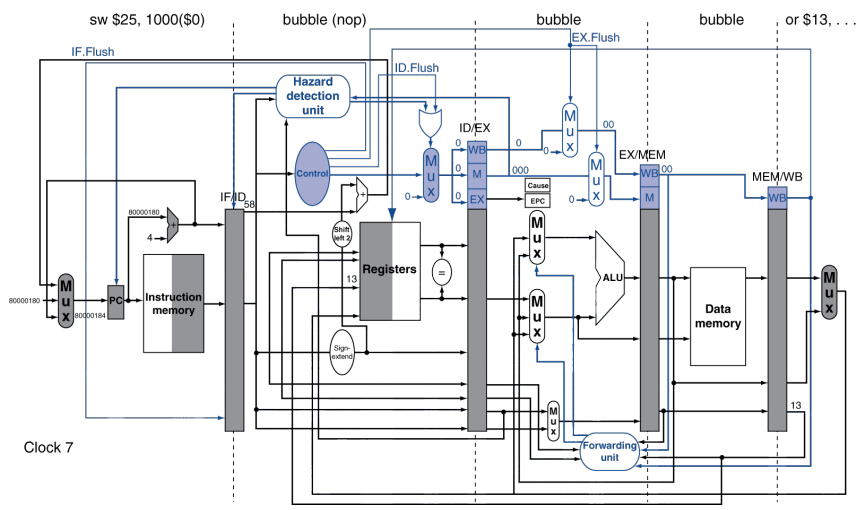
- Handler

```
80000180    sw    $26, 1000($0)   //$k0
80000184    sw    $27, 1004($0)   //$k1
…
```


Exception Example



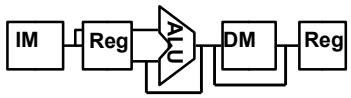
Exception Example



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!

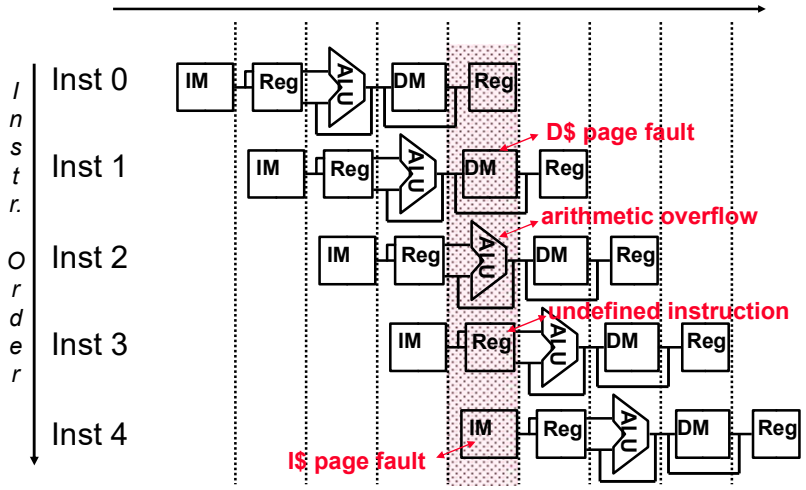
Where in the Pipeline Exceptions Occur



	Stage(s)?	Synchronous?
❑ Arithmetic overflow	EX	yes
❑ Undefined instruction	ID	yes
❑ TLB or page fault	IF, MEM	yes
❑ I/O service request	any	no
❑ Hardware malfunction	any	no

❑ Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Multiple Simultaneous Exceptions



❑ Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

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
- Not feasible for complex multiple-issue out-of-order pipelines