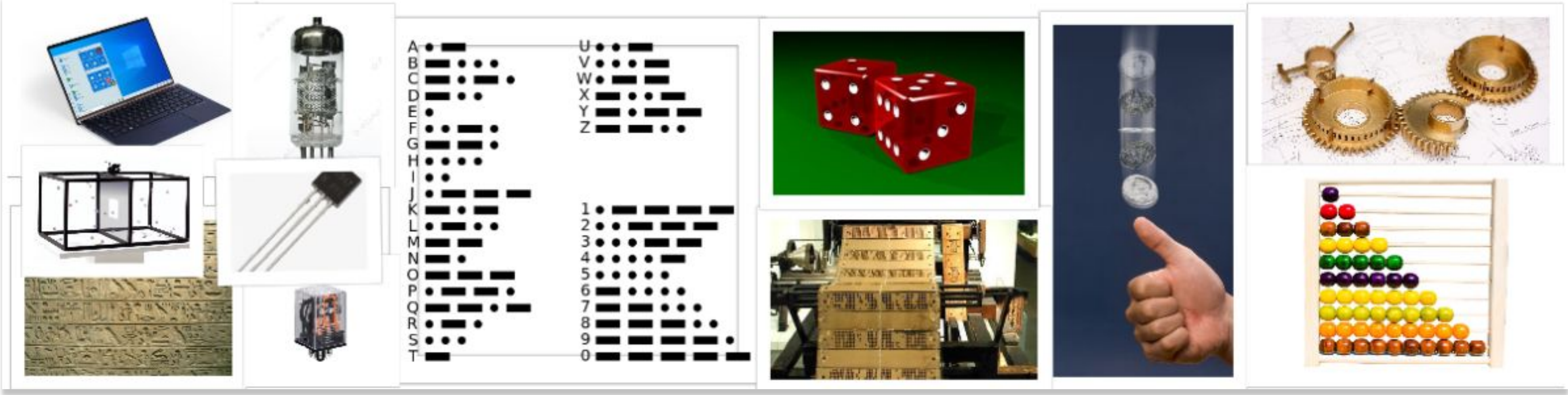


Computer Architecture

Design and Analysis of Instructions

Minimization of Control Hazards in Pipelined MIPS (RISC) Processor

Importance of Branch Instr. & its Penalty

Branch penalty –

- Intel core i7 - **14 cycles**
- ARM Cortex – A8 - **13 cycles**
- AMD – **19 cycles**
- MIPS – 5-stage pipeline - ?

Can you imagine a useful program without any branch instructions or if-else or loop?

Branchless programming

Program contains **15% to 25% branch instructions** among all instructions

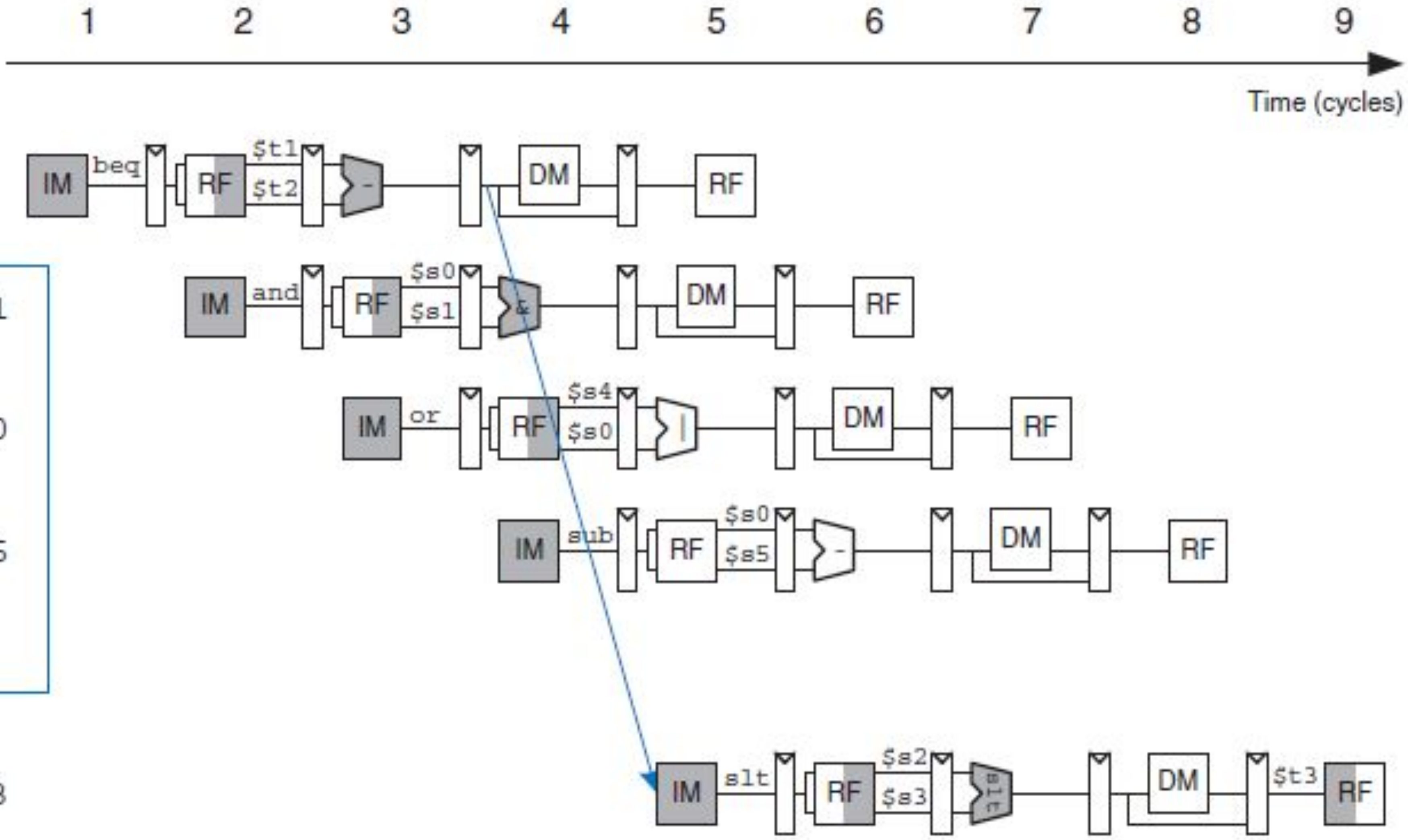
Objectives

- Minimize the penalty for control hazard
- Methods for the minimization
  - Appropriate time to take the branch decision
  - Filling the instruction the delayed branch slot
  - Branch prediction
    - Static [Compile time]
    - Dynamic [Execution time]
  - Branchless programming

What is this?

Consider $t1 and $t2 contain same value

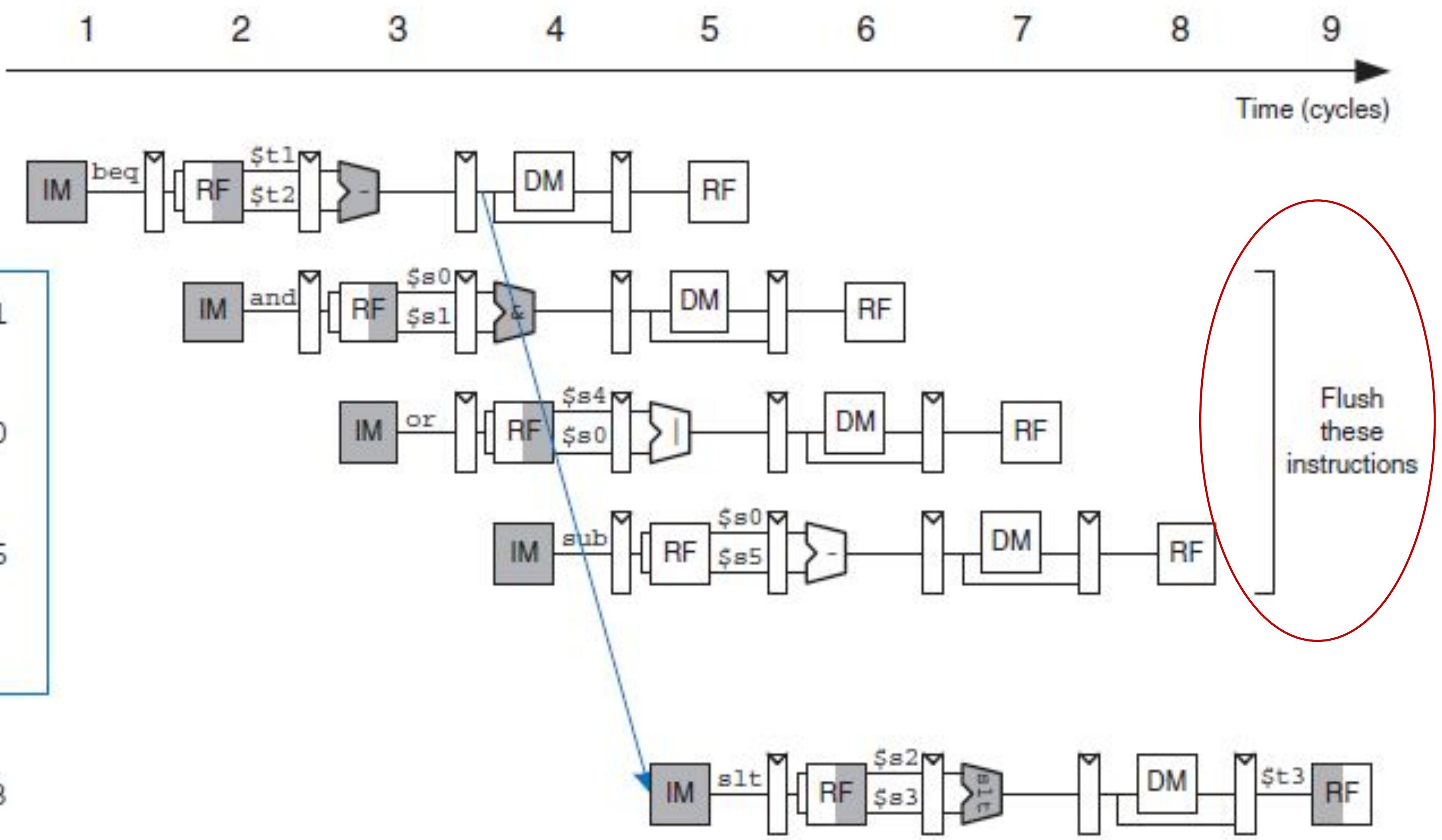
```
   beq $t1, $t2,  L

   and $t0, $s0, $s1

   or  $t1, $s4, $s0

   sub $t2, $s0, $s5

   ...

L: slt $t3, $s2, $s3
```



An example of Control Hazards

Consider $t1 and $t2 contain same value

```
   beq $t1, $t2, L
   and $t0, $s0, $s1
   or  $t1, $s4, $s0
   sub $t2, $s0, $s5
   ...
L: slt $t3, $s2, $s3
```



Control Hazards

- **EX-stage computes** the branch target address
- Cause higher performance penalty as compared to data hazards
- How does one reduce such penalty?
  - Take the branch decision early
    - ID and/or IF
    - IF-stage
      - Can we predict the next PC value from the current PC value?
        - $PC_{next} = \boldsymbol{g}(PC_{current})$

Control Hazards

- What if branch target address is computed at the **end of ID-stage**

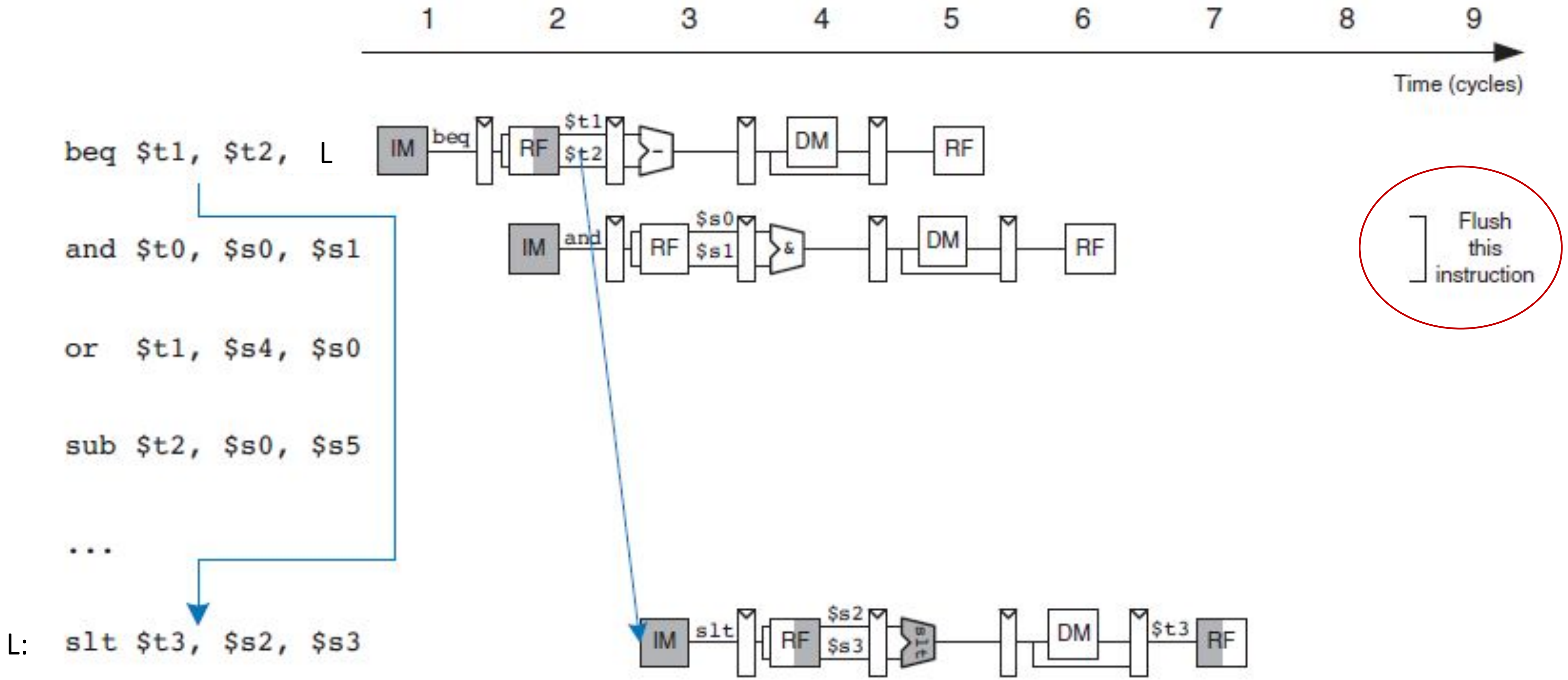
Branch instr.	IF	ID	EXE	MEM	WB		
Branch succ.		**IF**	**IF**	ID	EXE	MEM	WB
Branch succ + 1				IF	ID	EXE	MEM
Branch succ + 2					IF	ID	EX

$$\text{Speedup} = \frac{CPI\ unpipelined}{1+pipeline\ stall\ cycle\ from\ branchs}$$

$$= \frac{Pipeline\ depth}{1+Branch\ frequency \times Branch\ penalty}$$

Control Hazards

- Branch target address is computed at the end of ID-stage

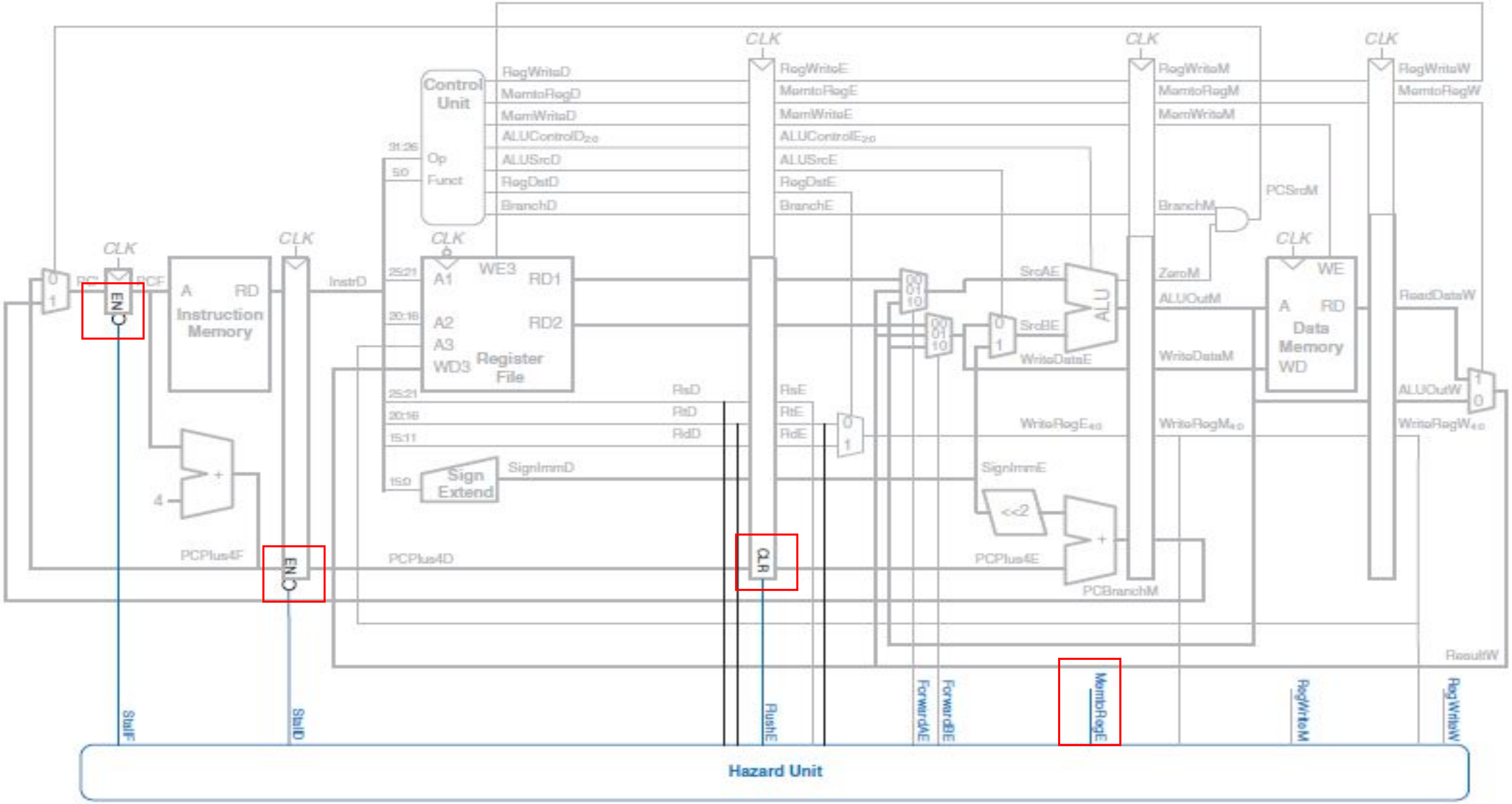


```
    beq $t1, $t2, L
    and $t0, $s0, $s1
    or  $t1, $s4, $s0
    sub $t2, $s0, $s5
    ...
L:  slt $t3, $s2, $s3
```

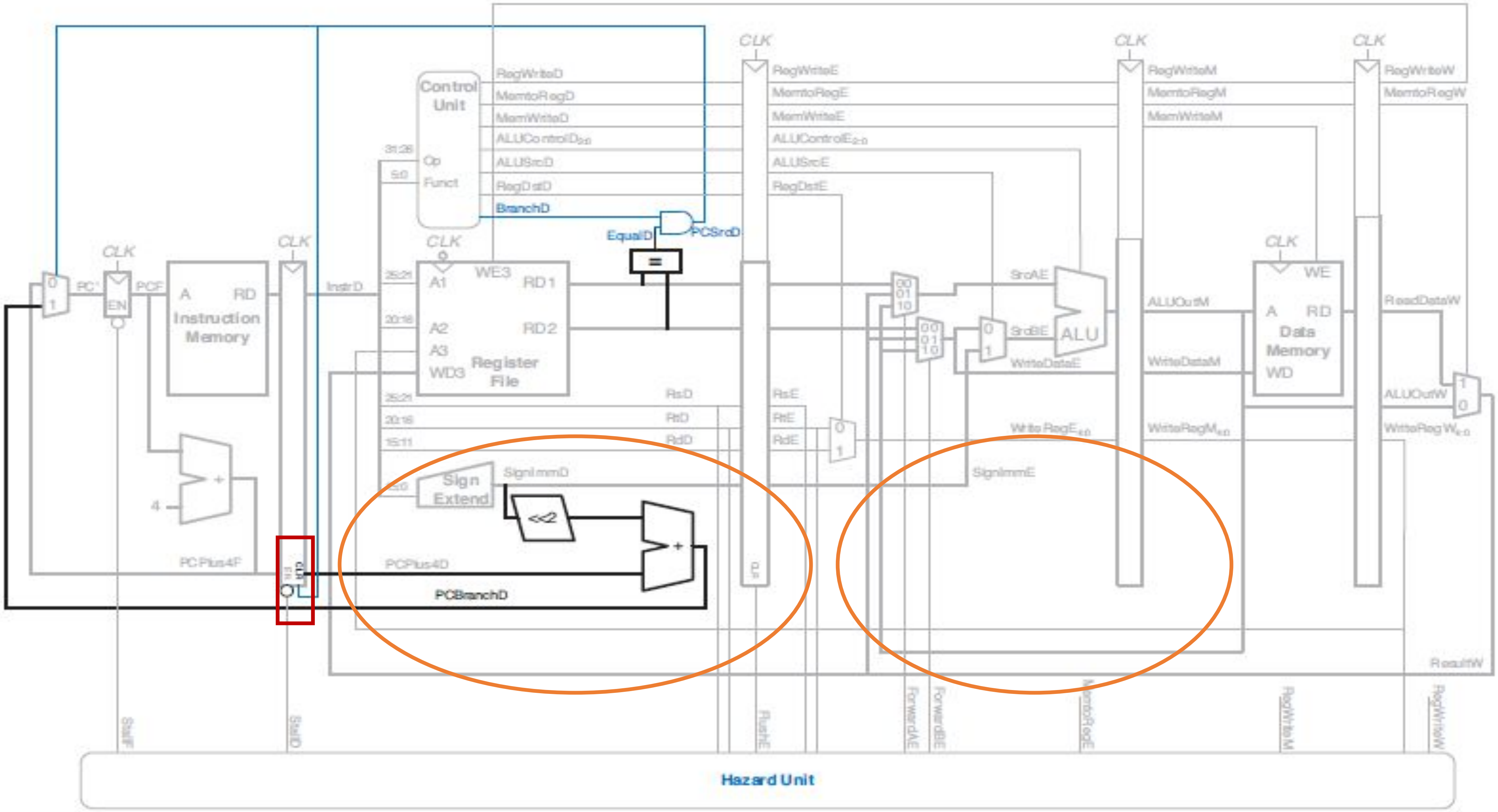
Flush this instruction

Control Hazards

- Branch target address is computed at the end of ID-stage
- What modifications are needed in the pipelined datapath & controlpath?



Control Hazards: Modified Pipeline

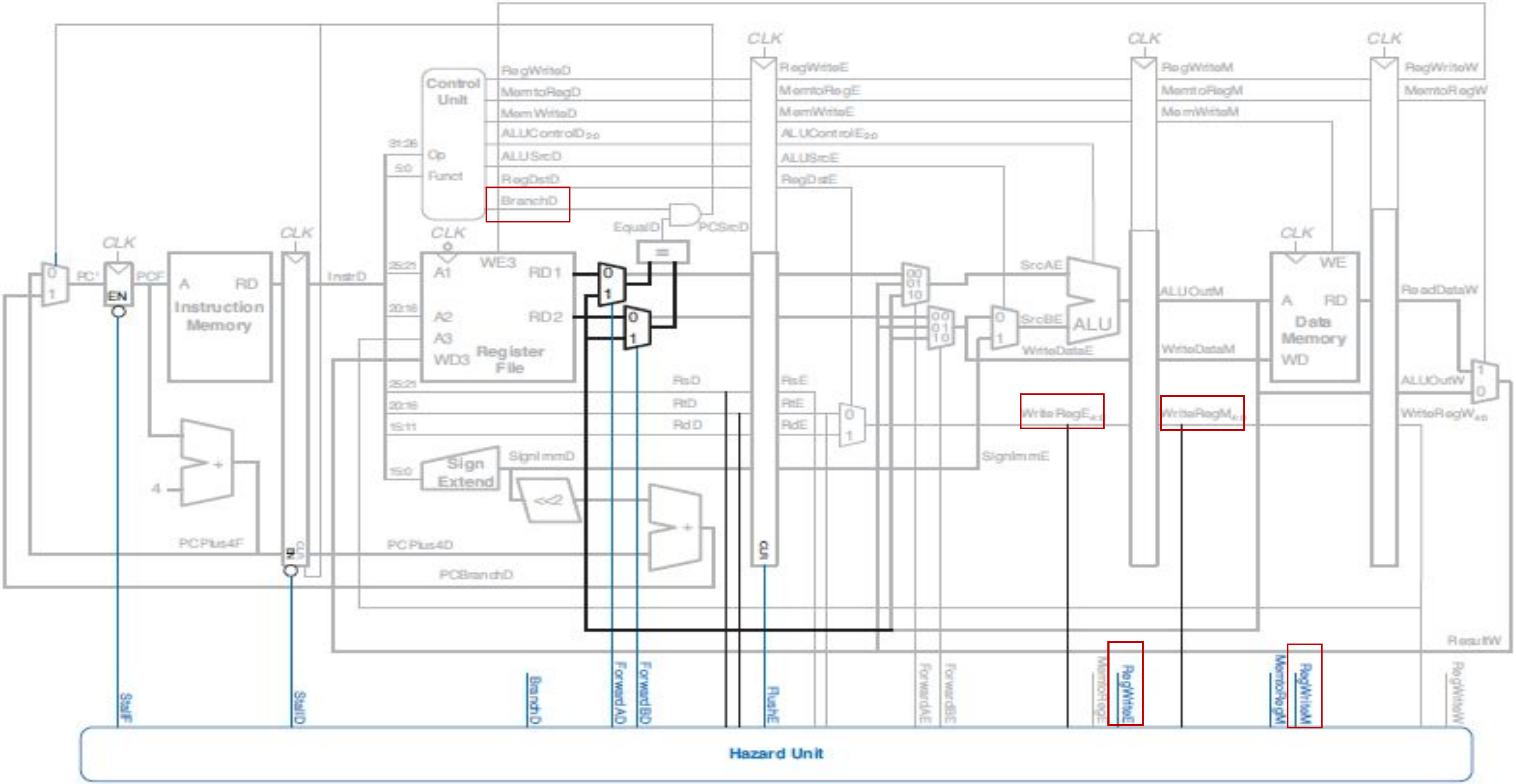


Control Hazards: Modified Pipeline

- What if one source operand of branch instruction was computed by a previous instruction and has not yet been updated into register file?

- One can use forwarding techniques
- Stalling technique can be used (lw-type)

Control Hazards & H/W-based solutions



Control Hazards & H/W-based solutions

- The function of the decode stage & forwarding logic

- FowardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
- FowardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM

E: EXE/MEM, D: ID/EXE, M: MEM/WB

Control Hazards & H/W-based solutions

- The function of the stall detection logic for a branch instruction
  - What if one of the source operands is in Execute stage and R-type instr. **or**
  - What if one of the source operands is in the Memory stage for a lw-type instruction?
- Branchstall=

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)

OR

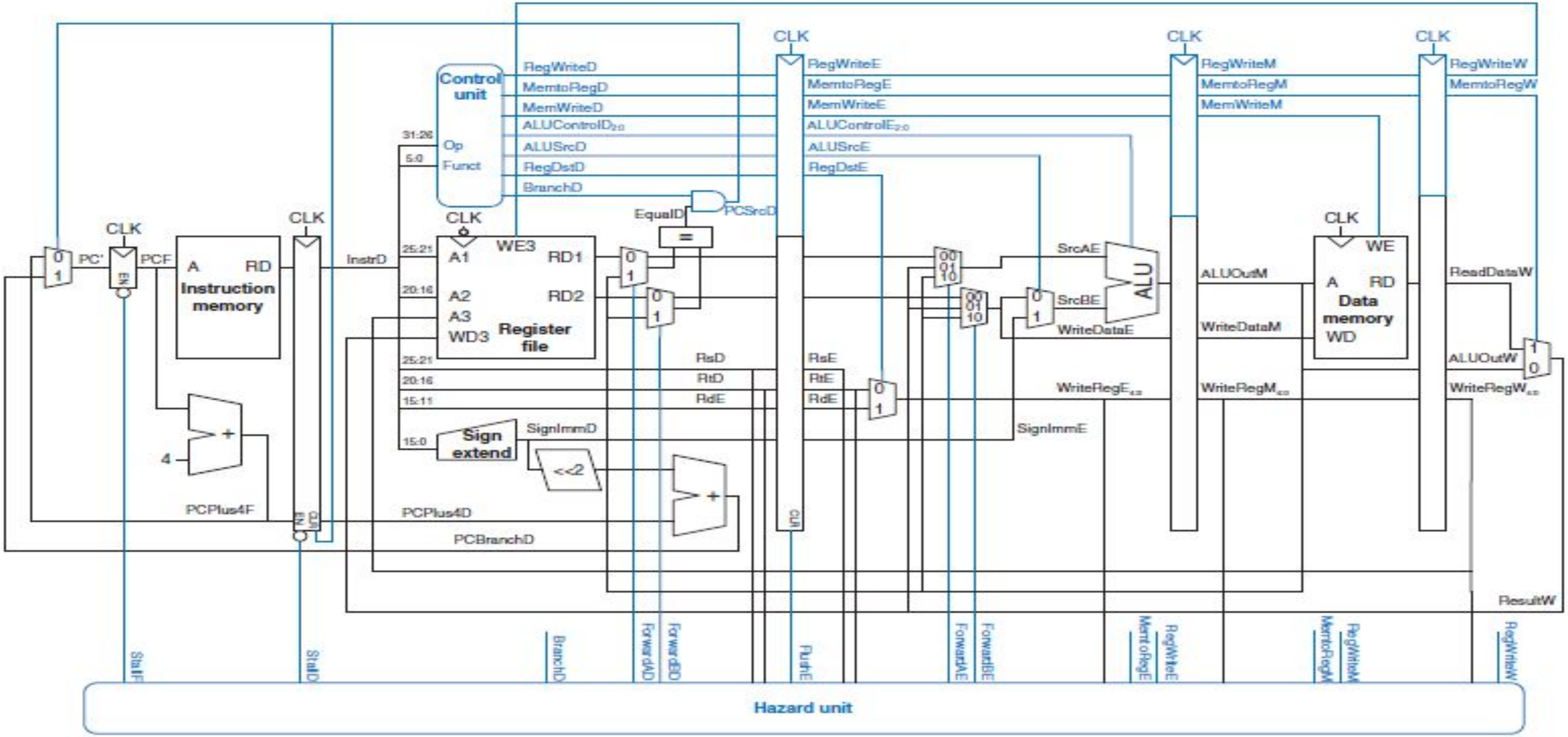
BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

E: EXE/MEM, D: ID/EXE, M: MEM/WB

Control Hazards & H/W-based solutions

- Now the pipelined processor stall due to either a load or a branch hazard
- StallF = StallD = FlushE = lwstall OR branchstall

Pipelined processor with full hazard handling unit



Pipelined processor with full hazard handling unit

- Determine the cycle time
  - consider the critical path
  - five pipeline stages
- Register file is written in the first half of the Writeback cycle and read in the second half of the Decode cycle
- The cycle time of the Decode and Writeback stages is <u>twice</u> the time necessary to do the half-cycle of work

- Cycle period, $T_c = max\begin{pmatrix} t_{pcq} + t_{mem} + t_{setup} \\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup} \\ t_{pcq} + t_{memwrite} + t_{setup} \\ 2(t_{pcq} + t_{mux} + t_{RFWrite}) \end{pmatrix}$

Pipelined processor with full hazard handling unit

- XYZ needs to compare the pipelined processor performance to that of the single-cycle and multicycle processors considered in earlier. Most of the logic delays is given in Table. The other element delays are 40 ps for an equality comparator, 15 ps for an AND gate, 100 ps for a register file write, and 220 ps for a memory write. Help XYZ compare the execution time of 100 billion instructions of the program for each processor.

Parameter	Delay (ps)
	30
	250
	20
	200
	25
	20
	20

Characteristics of the Program and Avg. CPI

Instruction type	%
Load	25
Store	10
Branch	11
Jump	2
R-type	52

Additional information:
40% loads are immediately followed by an instruction that uses the result, required a stall.
25% of branches are mispredicted, required a flush.
Jumps always flush the subsequent instructions.
Ignore other hazards.

Loads take 1-clock cycle when there is no dependency
2-clock cycle when there is a dependency
0.6 * (1) + 0.4 * (2)
Branches take 1-clock cycle for correct prediction
2-clock cycle for misprediction
0.75 *(1) + 0.25 *(2)
Jumps always have a CPI of 2
Other instructions have a CPI of 1

Average CPI=
0.25 [0.6 * (1) + 0.4 * (2)] +
0.1 [1] +
0.11 [0.75 *(1) + 0.25 *(2)] +
0.02 [2] +
0.52 [1]

=1.15

Pipelined processor with full hazard handling unit

- According to Equation on slide 15, the cycle time of the pipelined processor is

T_{c3} = max[30 + 250 + 20, 2(150 + 25 + 40 + 15 + 25 + 20), 30 + 25 + 25 + 200 +20, 30 + 220 + 20, 2(30 + 25 + 100)] = 550 ps.

- According to Equation of CPI, the total execution time is T_3 = (100 × 10^9 instructions)(1.15 cycles/instruction) (550 × 10^{-12} s / cycle) = 63.3 seconds.
- For the single-cycle processor it is 92.5 seconds and 133.9 seconds for the multicycle processor.
- What can be observed?

Static branch prediction

Control Hazards & Software-based solutions

- Assumption: predict-not-taken or predict-untaken
- Where does this assumption come from?
- Compiler rearranges the code
- Control flow will change only when prediction is wrong
- Example:

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Control Hazards & Software-based solutions

- Assumption: predict-not-taken or predict-untaken
- Where does this assumption come from?
- Compiler rearranges the code
- Control flow will change only when prediction is wrong
- Example:

Prediction goes wrong

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target +1				IF	ID	EX	MEM	WB	
Branch target +2					IF	ID	EX	MEM	WB

Control Hazards & Software-based solutions

- Assumption: predict-taken
- Compiler rearranges the code, for both the assumptions, so that the most frequent path matches the hardware’s choice

Control Hazards & Software-based solutions

- Which assumption is suitable for pipelined MIPS-based processor?
- Predict-untaken

Control Hazards & Software-based solutions

- Prediction based on control flow
- How many types of branch are possible?
  - Forward branch
  - Backward branch
- For backward-type branch, predict-taken is suitable
- For forward-type branch, predict-untaken is suitable

Control Hazards & Software-based solutions

- Is there another way to specify predict-taken or predict-untaken in the branch instruction?
- A bit in the branch opcode
    - Bit set means predict-taken
    - Bit not set means predict-untaken
- Who will set or reset such bit?
    - Compiler

Control Hazards & Software-based solutions

- Can we eliminate one cycle delay associated in branch prediction?
- Delayed branch technique

Control Hazards & Delayed branch (S/W-based approach)

- Delayed branch technique
- Examples

- Find useful & independent instruction
- How many delay slots?

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch Delay Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch Delay Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

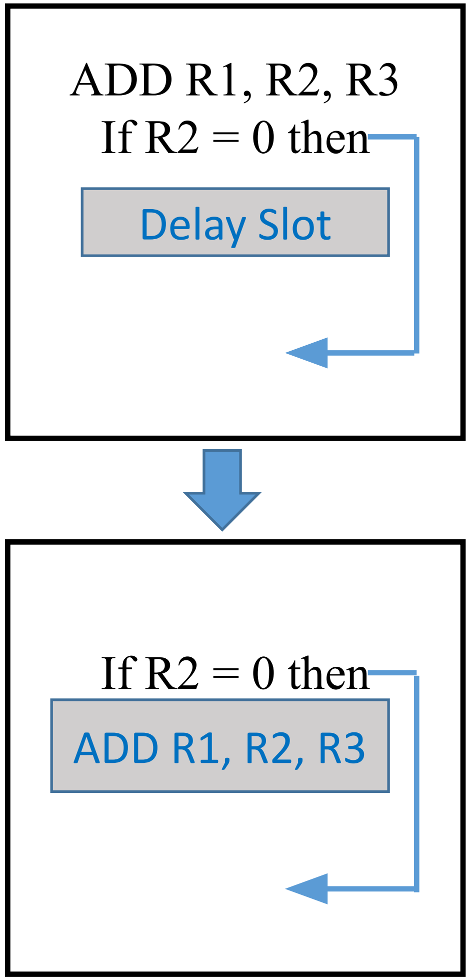
Control Hazards & Delayed branch (S/W-based approach)

- How does compiler find useful & independent instruction?
- Can compiler always find such instruction?

Control Hazards & Delayed branch (S/W-based approach)

- How does compiler find useful & independent instruction?

- Can we put a branch instruction in the delay slot?

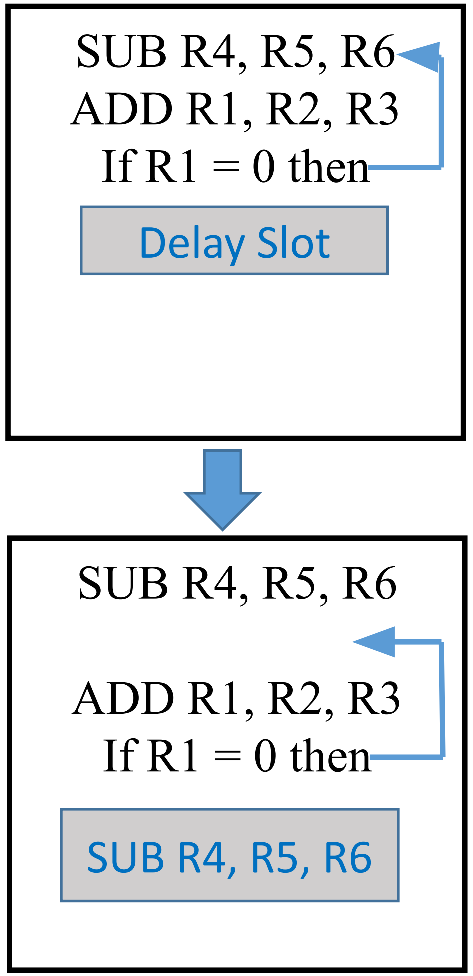


ADD R1, R2, R3
If R2 = 0 then
Delay Slot

If R2 = 0 then
ADD R1, R2, R3

From before

Assumption: Always taken

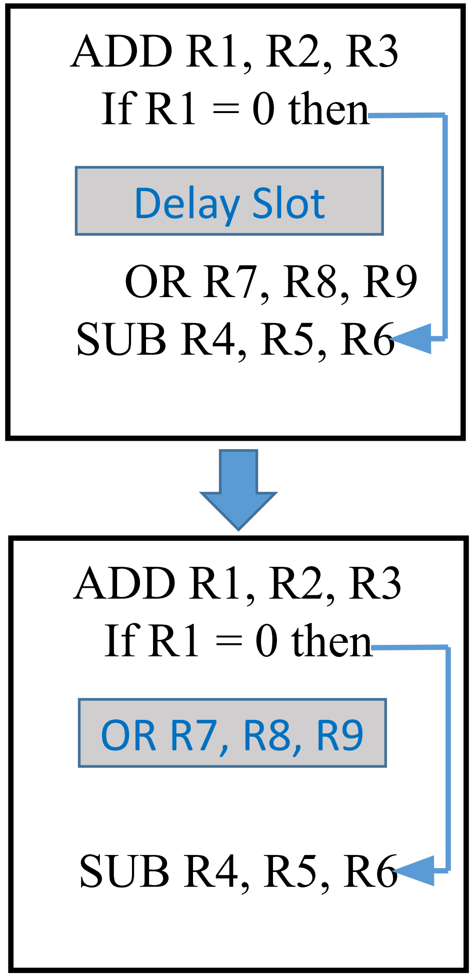


SUB R4, R5, R6
ADD R1, R2, R3
If R1 = 0 then
Delay Slot

SUB R4, R5, R6
ADD R1, R2, R3
If R1 = 0 then
SUB R4, R5, R6

From target

Assumption: Always not taken



ADD R1, R2, R3
If R1 = 0 then
Delay Slot
OR R7, R8, R9
SUB R4, R5, R6

ADD R1, R2, R3
If R1 = 0 then
OR R7, R8, R9
SUB R4, R5, R6

From fall-through

Control Hazards, static analysis & software-based solution

- Previous software-based approaches are based on static analysis
    - Assumption on Processor Design
        - Predict-not-taken
        - Predict-taken
    - Assumption on Control flow
        - Forward
        - Backward
    - Delayed branch
- Can we take decision during execution of the program?

Dynamic branch prediction

Control Hazards and Dynamic Branch Prediction

- Previous software-based approaches are based on static analysis
- Equal priority given to each branch instruction, however reality may differ
    - Branch prediction change if inputs of the program change
- We need to predict the branch behavior during runtime

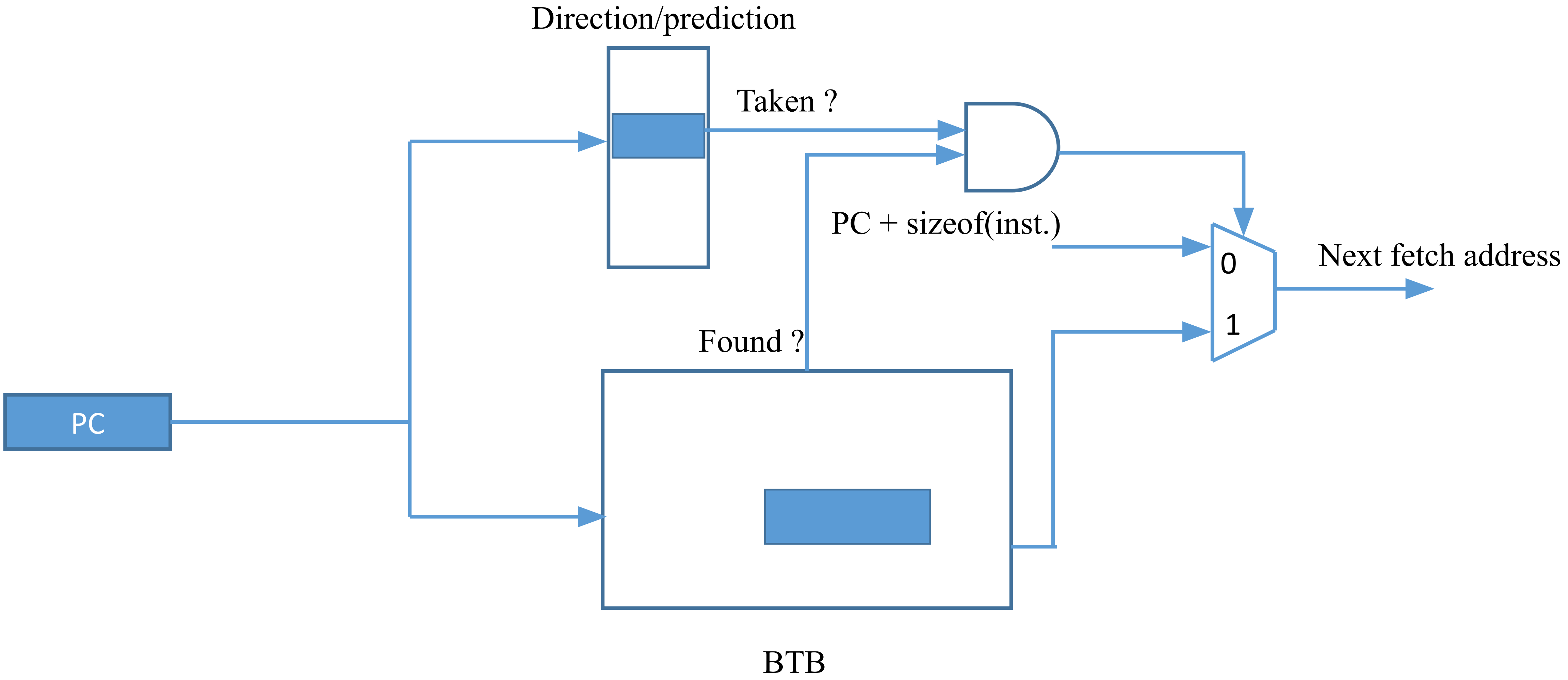
Control Hazards and Dynamic Branch Prediction

- How does one predict the branch behavior during execution?
- Exploit the previous execution history of the instruction
- What are the components needed to do such thing?
- Component to make prediction & store target address
- Which stage of the pipeline is suitable for that?
- IF-stage

Control Hazards and Dynamic Branch Prediction

- Three items are needed in IF-stage
  - Is it a branch instruction?
  - What is the branch direction?
    - Conditional
    - Unconditional
  - What is the target address (if taken)?
- Is the target address fixed or changed?

Dynamic Branch Prediction: BTB

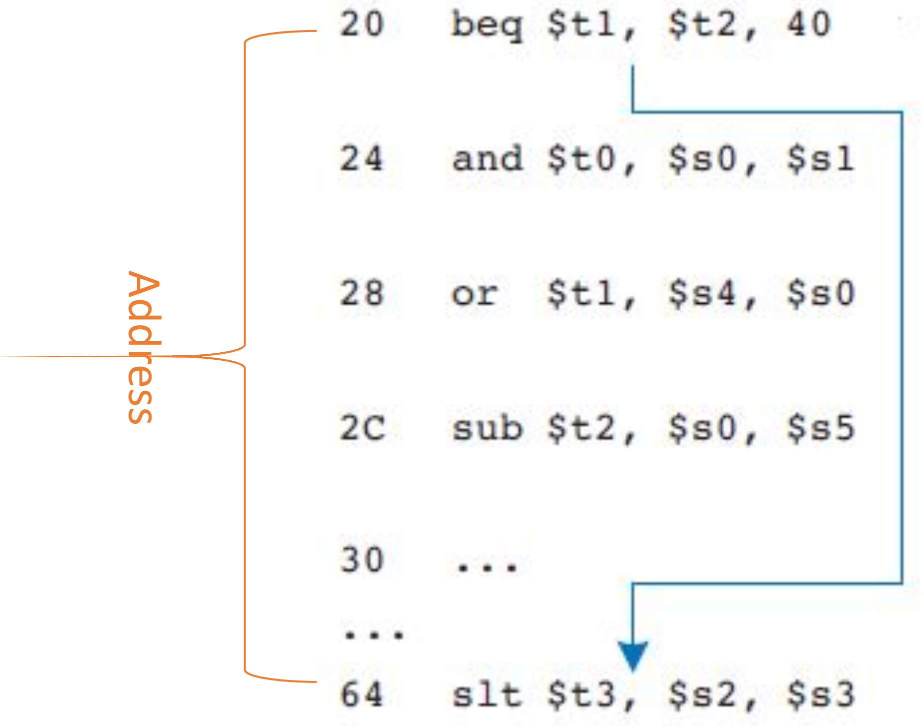


Control Hazards and Dynamic Branch Prediction

- The *branch-target buffer* (BTB) or *branch-target (address) cache* (BTAC) is a branch-prediction cache that stores the predicted address for the next instruction after a branch

Consider $t1 and $t2 contain same value

PC = (PC + 4) + signExtn(40)



```
20   beq $t1, $t2, 40

24   and $t0, $s0, $s1

28   or  $t1, $s4, $s0

2C   sub $t2, $s0, $s5

30   ...
...
64   slt $t3, $s2, $s3
```

Branch address	Target address	Prediction bits
20	64	
	… … … … …	

Control Hazards and Dynamic Branch Prediction

- The BTB is accessed during the IF-stage
- A table with branch addresses, the corresponding target addresses, and prediction information
- The PC for the next instruction to fetch is compared with the entries in the BTB. If a matching entry is found in the BTB, fetching can start immediately at the target address

Control Hazards and Dynamic Branch Prediction

- What is this prediction bit in the BTB?
- How many bits are needed?

Control Hazards and Dynamic Branch Prediction

- Example: branch outcome sequence (T – Taken & N – Not Taken)
    - T N T N T N T N T N

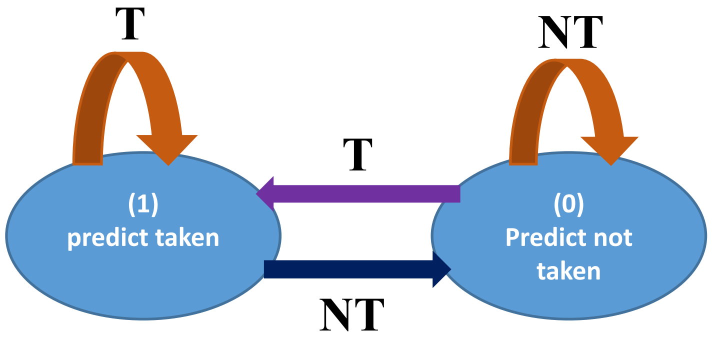
- How does one design such a predictor?

Control Hazards and Dynamic Branch Prediction

- One bit predictor
  - If the bit is set, the branch is predicted taken
  - If the bit is not set, the branch is predicted not taken
  - In the case of a misprediction, the bit state is reversed and stored back and so is the prediction direction
- How does one design such predictor?
  - Encode the states
  - Relation among the states
  - Finite State Machine (FSM)

Control Hazards and Dynamic Branch Prediction

- One bit predictor
  - If the bit is set, the branch is predicted taken
  - If the bit is not set, the branch is predicted not taken
  - In the case of a misprediction, the bit state is reversed and stored back and so is the prediction direction



T for Taken
NT for Not Taken

Example

B-name	Code	Addr
	m=10	0
B1	while(m>0){	4
	n=5;	8
B2	while(n>0){	12
	n--;	16
	}	
	m--;	20
	}	

BTB		
BA	TA	Prediction(P)
4	8	0
12	16	0

Initial stage

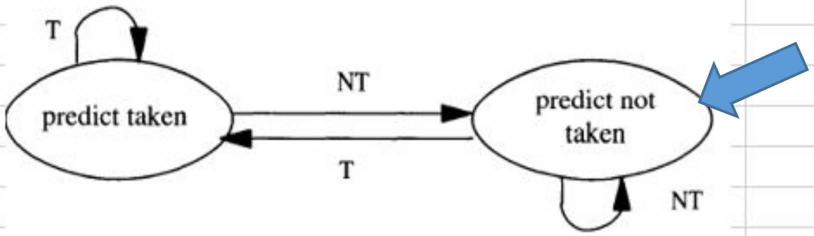
P	P	P	P	P	P	P	P	P
1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0	1

- - - - -

	m=10, n=5	m=10, n=5	m=10, n=4	m=10, n=3	m=10, n=2	m=10, n=1	m=10, n=0	m=9, n=5	m=9, n=5
Branch Name	B1	B2	B2	B2	B2	B2	B2	B1	B2
Branch Prediction	NT	NT	T	T	T	T	T	T	NT
Branch Outcome	T	T	T	T	T	T	NT	T	T

MSB bit of a state used for prediction

0	NT
1	T



Is there any shortcoming of this approach?

Control Hazards and Dynamic Branch Prediction

- Shortcoming of one bit predictor
- Predict taken always
- Predict incorrectly twice (why?), rather than once, when it is not taken
- What if we have the nested loops
  - Two misprediction for inner loop
    - Entry in the loop
    - Exit from the loop
  - For a loop with N-iteration the accuracy is $\frac{N-2}{N}$
  - How does one avoid such double misprediction?

TTTTTNNNNN…

What is the accuracy?

80%

TNTNTNTNTN…

What is the accuracy?

0%

Control Hazards and Dynamic Branch Prediction

Two-bit predictor:

- Two-bits are in each entry in the BHT
- The two bits stand for the prediction states:
  - “predict strongly taken”
  - “predict weakly taken”
  - “predict strongly not taken”
  - “predict weakly not taken”

- For a misprediction in the “strongly” state cases, the <u>prediction direction is not changed</u>, rather the prediction goes into the respective “weakly” state

- A prediction must <u>miss twice, before changing the state</u>

Control Hazards and Dynamic Branch Prediction

- How does one design such 2-bits predictor?

Control Hazards and Dynamic Branch Prediction

- How does one design such 2-bits predictor?
  - Encode the different states
  - Relation among the states
  - Finite State Machine (FSM)

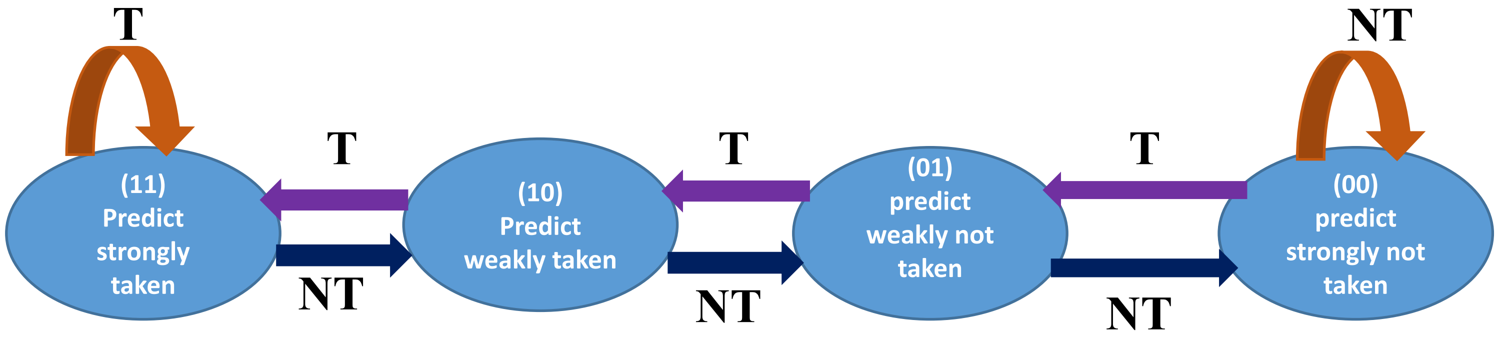
Control Hazards and Dynamic Branch Prediction

- Two kinds of 2-bits prediction methodology
    - The saturation up-down counter
    - Others

Control Hazards and Dynamic Branch Prediction

- The saturation up-down counter
  - Taken branch
    - Increment
  - Not taken
    - Decrement
  - Saturation
  - MSB of a state determine the prediction

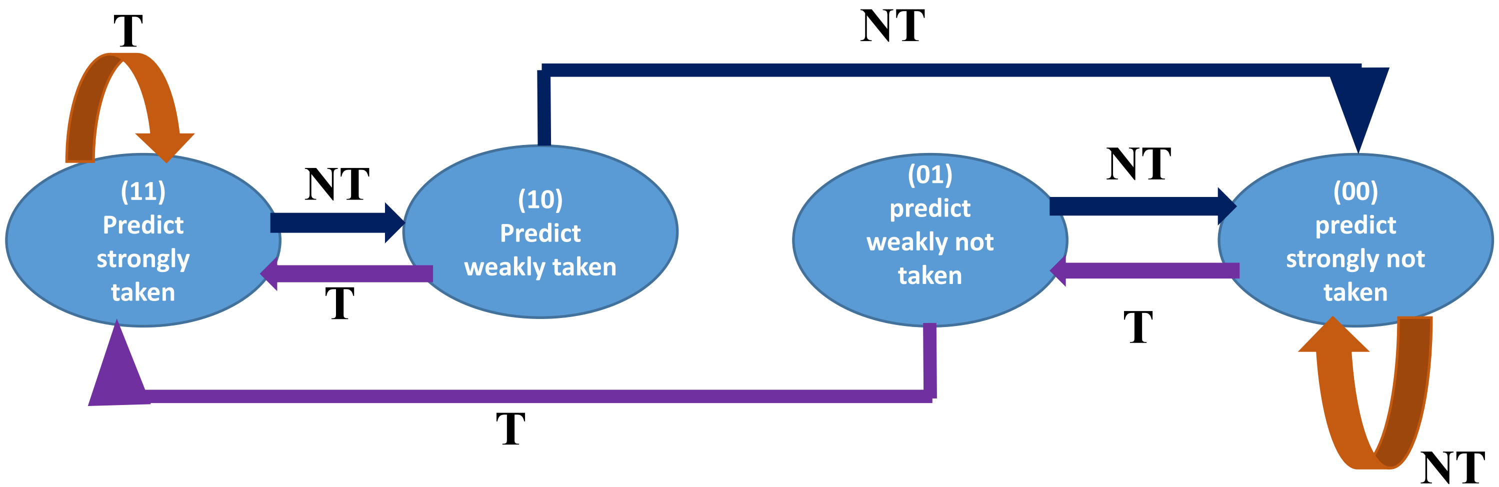
T for Taken
NT for Not Taken


T
NT
(11) Predict strongly taken
T
NT
(10) Predict weakly taken
T
NT
(01) predict weakly not taken
T
NT
(00) predict strongly not taken

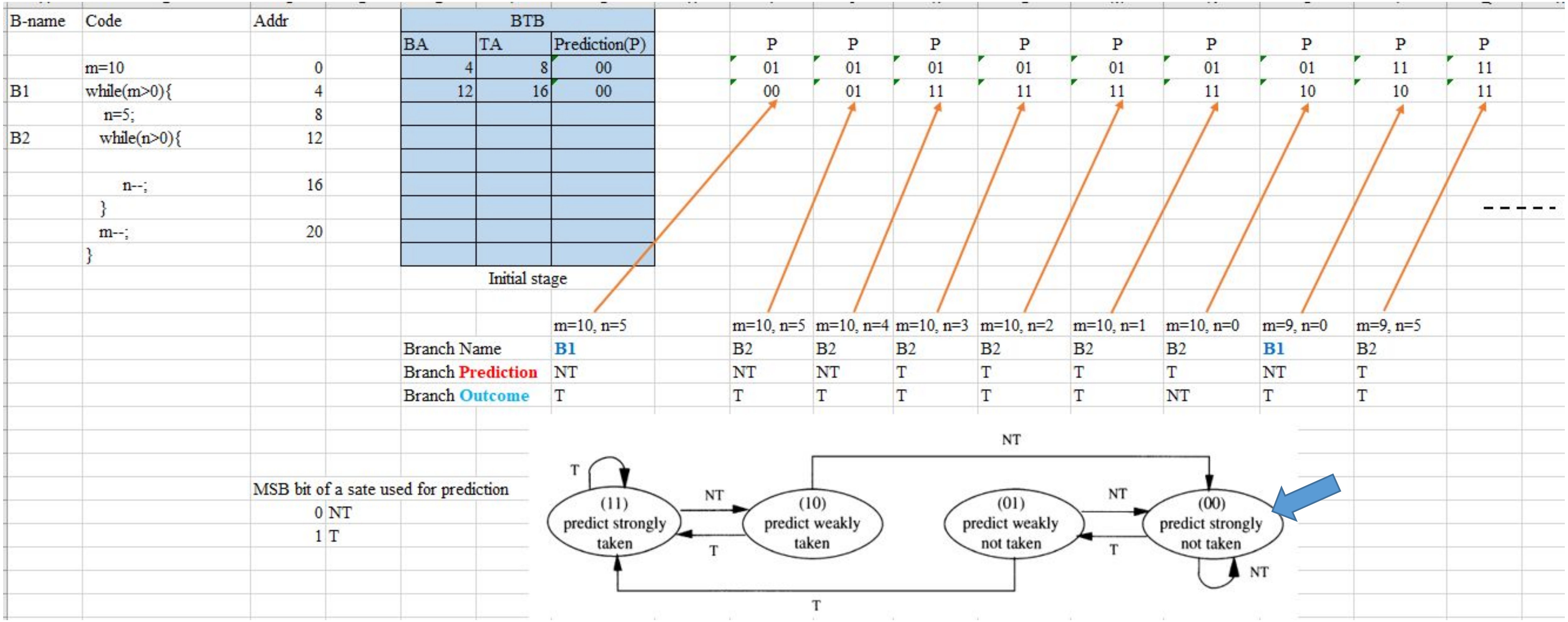
Control Hazards and Dynamic Branch Prediction

- Other methodology
  - It differs from the saturation up-down counter method by changing directly from the “weakly” to the “strongly” states, in case of misprediction
  - Minimize the switching (or power consumption)

T for Taken
NT for Not Taken


T
NT
NT
NT
(11) Predict strongly taken
(10) Predict weakly taken
(01) predict weakly not taken
(00) predict strongly not taken
T
T
T
NT

Example



B-name	Code	Addr
	m=10	0
B1	while(m>0){	4
	n=5;	8
B2	while(n>0){	12
	n--;	16
	}	
	m--;	20
	}	

BTB		
BA	TA	Prediction(P)
4	8	00
12	16	00

Initial stage

P	P	P	P	P	P	P	P	P
01	01	01	01	01	01	01	11	11
00	01	11	11	11	11	10	10	11

	m=10, n=5	m=10, n=5	m=10, n=4	m=10, n=3	m=10, n=2	m=10, n=1	m=10, n=0	m=9, n=0	m=9, n=5
Branch Name	**B1**	B2	B2	B2	B2	B2	B2	**B1**	B2
Branch **Prediction**	NT	NT	NT	T	T	T	T	NT	T
Branch **Outcome**	T	T	T	T	T	T	NT	T	T

MSB bit of a sate used for prediction	
0	NT
1	T

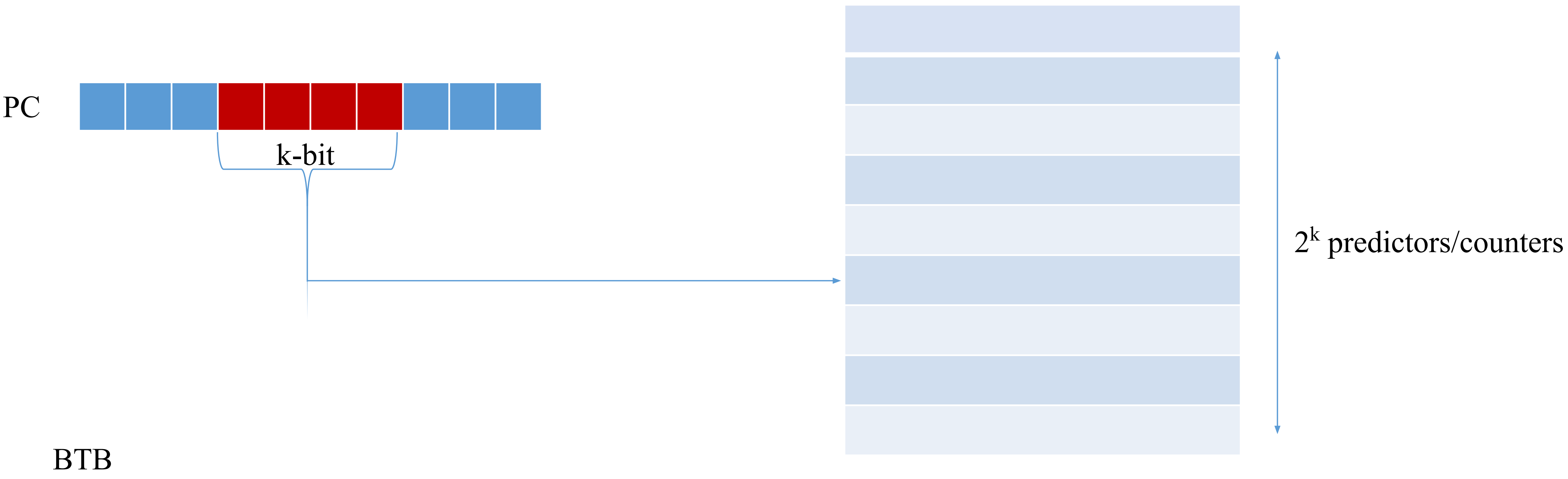
Control Hazards and Dynamic Branch Prediction

- n-bits predictor
  - n-bit counter (0 to 2^n-1)
  - Taken when counter value is one-half of the max. value (2^n-1)
  - Otherwise, Not taken
  - Studies of n-bits predictor have shown that 2-bits predictor do almost well, thus most systems rely on 2-bits predictor

Bimodal Predictor

- Prediction data are not stored with BTB
- Prediction data are stored in the Pattern History Table (PHT), which is separated from BTB. Why?

Pattern History Table (PHT)



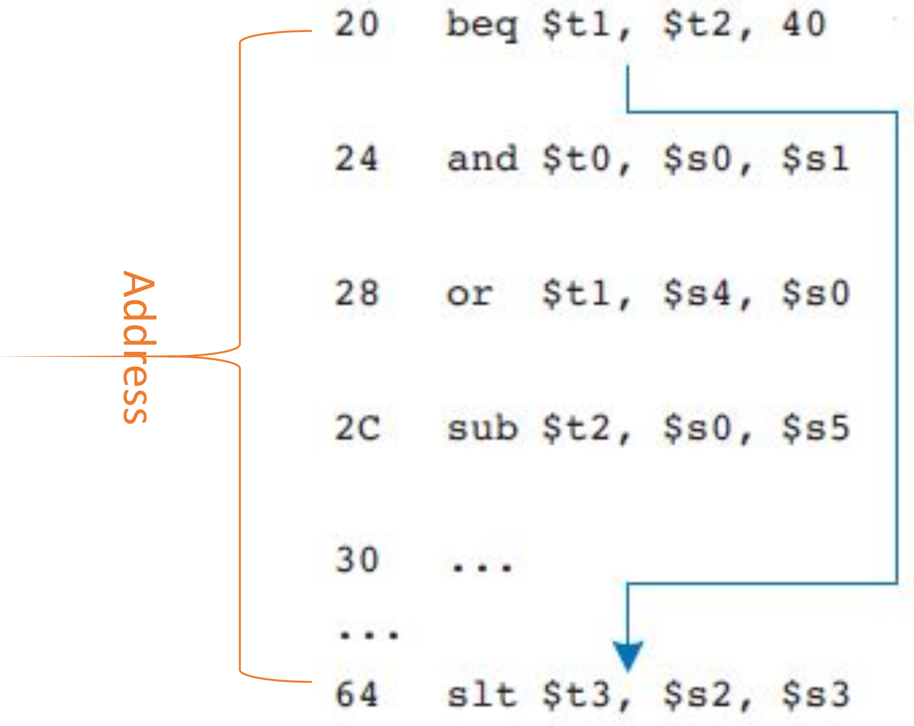
How does size of the table affect the performance?

Control Hazards and Dynamic Branch Prediction

- The *branch-target buffer* (BTB) or *branch-target (address) cache* (BTAC) is a branch-prediction cache that stores the predicted address for the next instruction after a branch

Consider $t1 and $t2 contain same value

PC = (PC + 4) + signExtn(40)



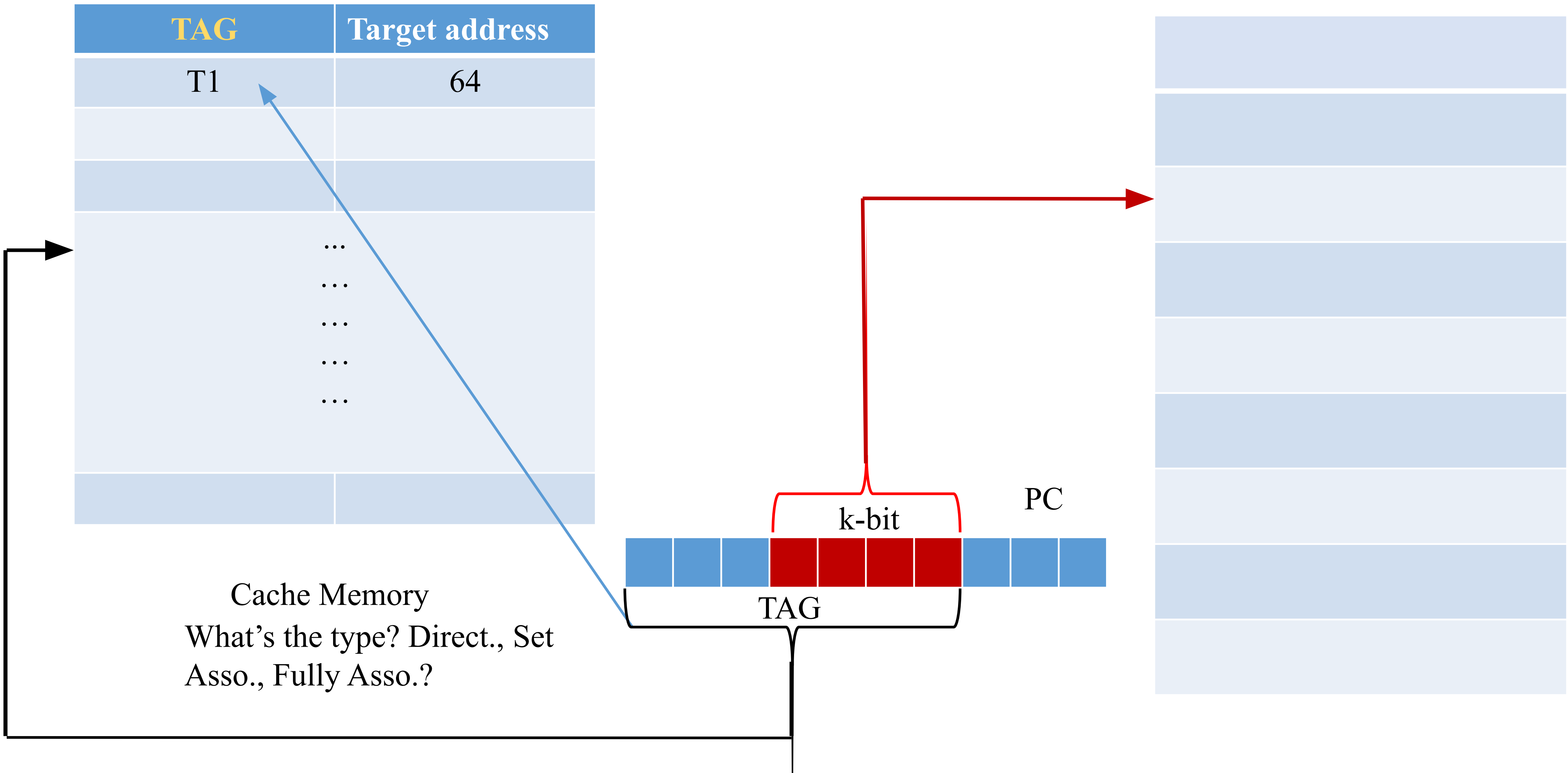
```
20   beq $t1, $t2, 40

24   and $t0, $s0, $s1

28   or  $t1, $s4, $s0

2C   sub $t2, $s0, $s5

30   ...
...
64   slt $t3, $s2, $s3
```

Integrated BTB-PHT

Branch address	Target address	Prediction bits
20	64	
	… … … … …	

Decoupled BTB-PHT

Pattern History Table (PHT)



TAG	Target address
T1	64
…	
…	
…	
…	
…	

Cache Memory

What’s the type? Direct., Set Asso., Fully Asso.?

Homework

- Write a Verilog and C++ code for pipelined with full hazard detection unit
  - Incorporate the dynamic branch prediction
- Write a Verilog and C++ code for generalized N-bit branch predictor
- Is there any effect on the branch prediction for the starting state?

Homework: Control Hazards: Performance Analysis

- No penalty for correct prediction
- 3 bubbles for incorrect prediction
- No data dependency among the instructions
- 20% instructions are branch
- 70% of branches are taken
- What is the CPI?
- What is the probability of wrong guess?
- What is the penalty of wrong guess?

Summary

- Control hazard
- Performance analysis
- Flush the pipeline
- Hardware-based solution technique
  - Take decision at ID stage
- Software-based solution technique
  - Predict-taken
  - Predict-untaken
  - Delayed branch
- Dynamic branch prediction techniques
  - N-bit FSM