Lecture 3

Instruction scheduling techniques

- **Dynamically scheduled pipelines (Ch 3.4)**
  - ✓Tomasulo's algorithm (Ch 3.4.1)
  - ✓Speculative execution (Ch 3.4.2)
  - ✓Dynamic branch prediction (Ch 3.4.3)

Dynamic Instruction Scheduling

CHALMERS
Chalmers University of Technology

Dynamic Instruction Scheduling

Static pipelines can only exploit the parallelism exposed to it by the compiler

- Instructions are stalled in ID until they are hazard-free and then are scheduled for execution
- The compiler strives to limit the number of stalls in ID
- However, compiler is limited in what it can do

Potentially there is a large amount of Instruction Level Parallelism (ILP) to exploit (across 100s of instructions)

- Must cross basic block boundaries (10s of branches)
- Data-flow order (dependencies) – not program order

Dynamic Instruction Scheduling (Cont'd)

Dynamic instruction scheduling

- Decode instructions then dispatch them in queues where they wait to be scheduled until input operands are available
- No stall at decode for data hazard – only structural hazards

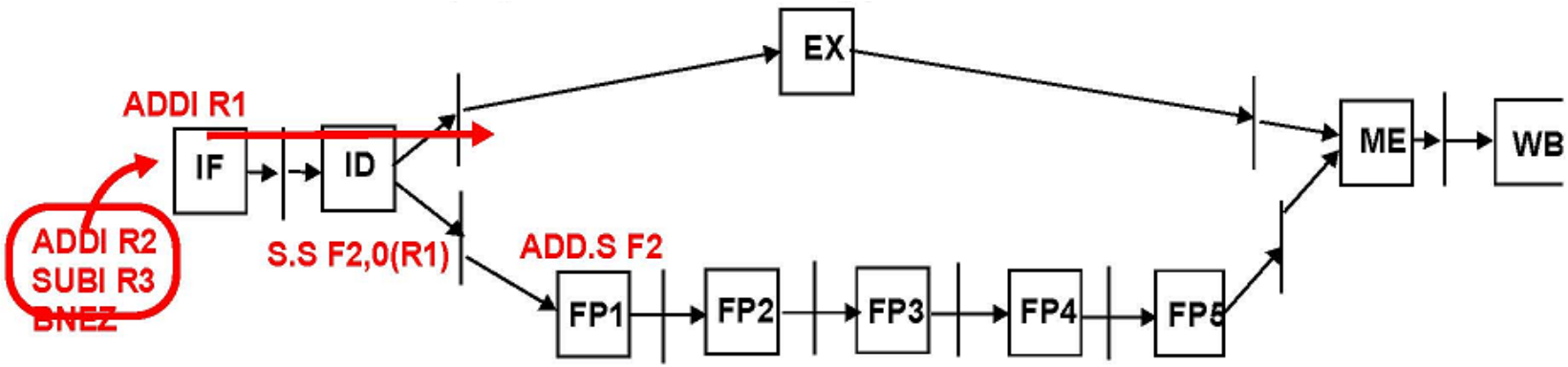
To extract and exploit the vast amount of ILP we must meet several challenges

- All data hazards – RAW, WAW, and WAR – are now possible both on memory and registers and must be avoided
- Execute beyond conditional branches – *speculative execution*
- Enforce the precise exception model

Example of Dynamic Scheduling

```
Loop  L.S   F0,0(R1)     (1)
      L.S   F1,0(R2)     (1)
      ADD.S F2,F1,F0     (2)
      S.S   F2,0(R1)     (5)
      ADDI  R1,R1,#4     (1)
      ADDI  R2,R2,#4     (1)
      SUBI  R3,R3,#1     (1)
      BNEZ  R3,Loop      (3)
```

- ADDI R1 could pass through ID while the Store waits (watch for WAR hazard on R1)
- ADDI R2 is fetched and bypass store as well
- SUBI R3 is fetched and bypass the store. By that time the ADD.S is in FP4 and the Store has to stall one more cycle
- Could even have the branch bypass the Store



Limits to this approach: Pipeline has to be redesigned

Quiz 3.1

```
Loop L.S   F0,0(R1)     (1)
     L.S   F1,0(R2)     (1)
     ADD.S F2,F1,F0     (2)
     S.S   F2,0(R1)     (5)
     ADDI  R1,R1,#4     (1)
     ADDI  R2,R2,#4     (1)
     SUBI  R3,R3,#1     (1)
     BNEZ  R3,Loop      (3)
```

In how many cycles would the above loop iteration be executed under ideal dynamic scheduling with an integer and floating point execution unit?

a) 8 cycles

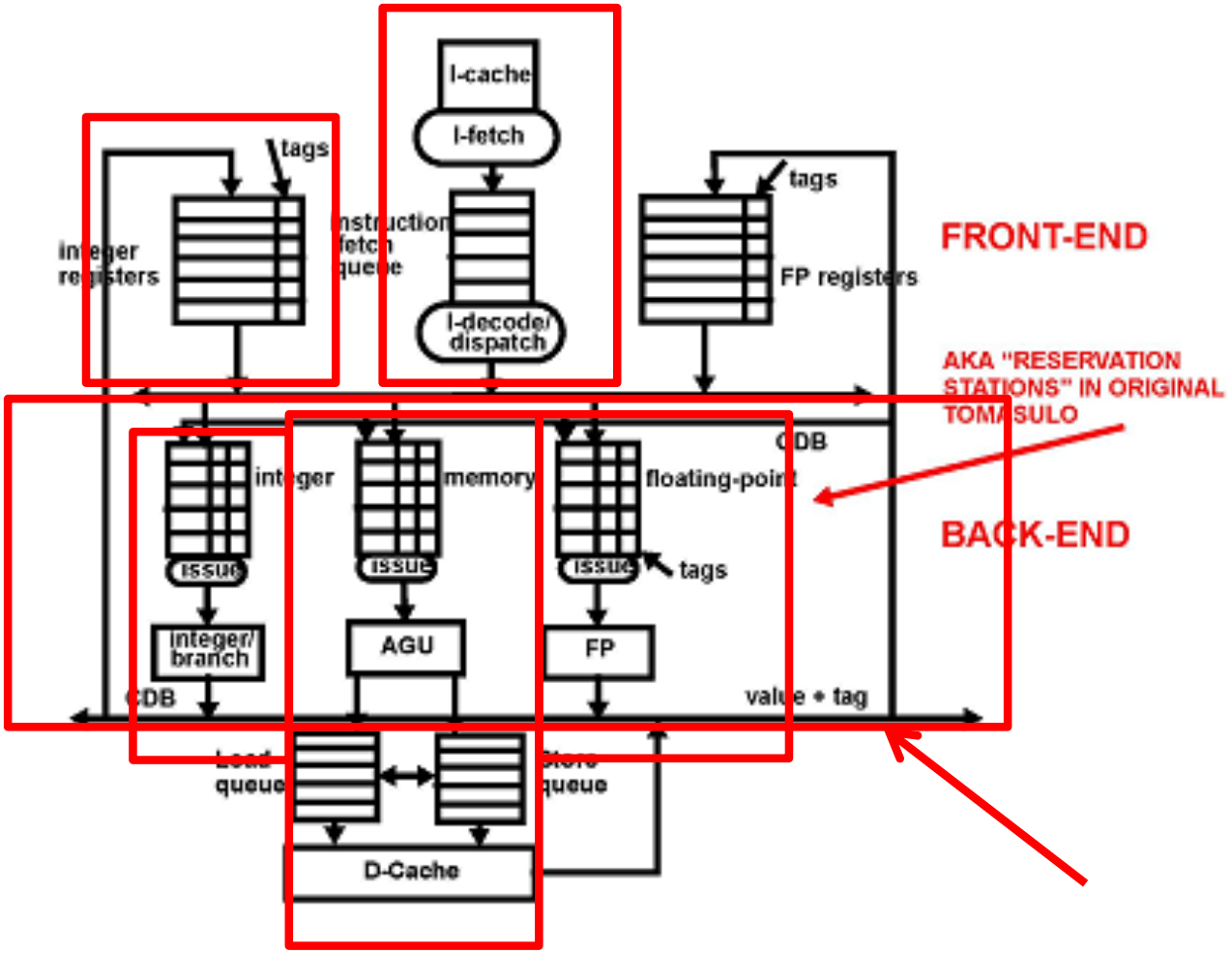
b) 13 cycles

c) 9 cycles

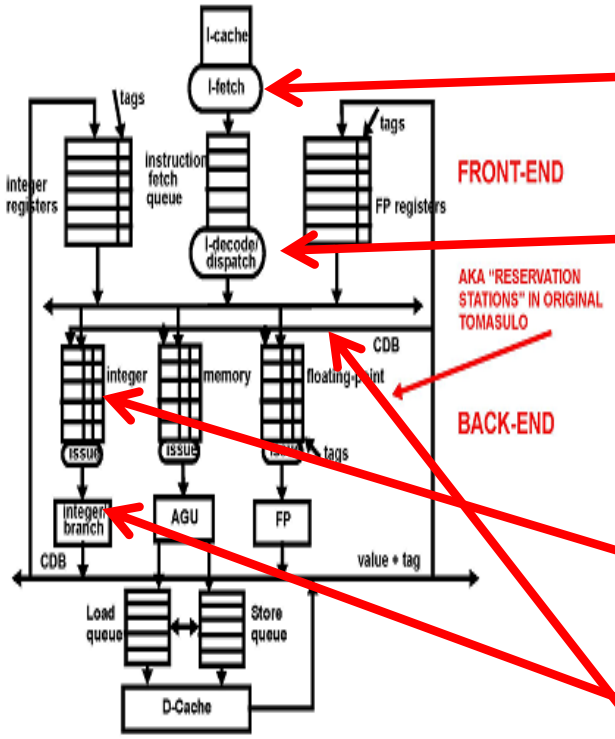
Dynamic Instruction Scheduling: Tomasulo's Algorithm (Ch 3.4.1)

CHALMERS
Chalmers University of Technology

Tomasulo: Pipeline Structure



Tomasulo: Algorithm Overview



Front-end

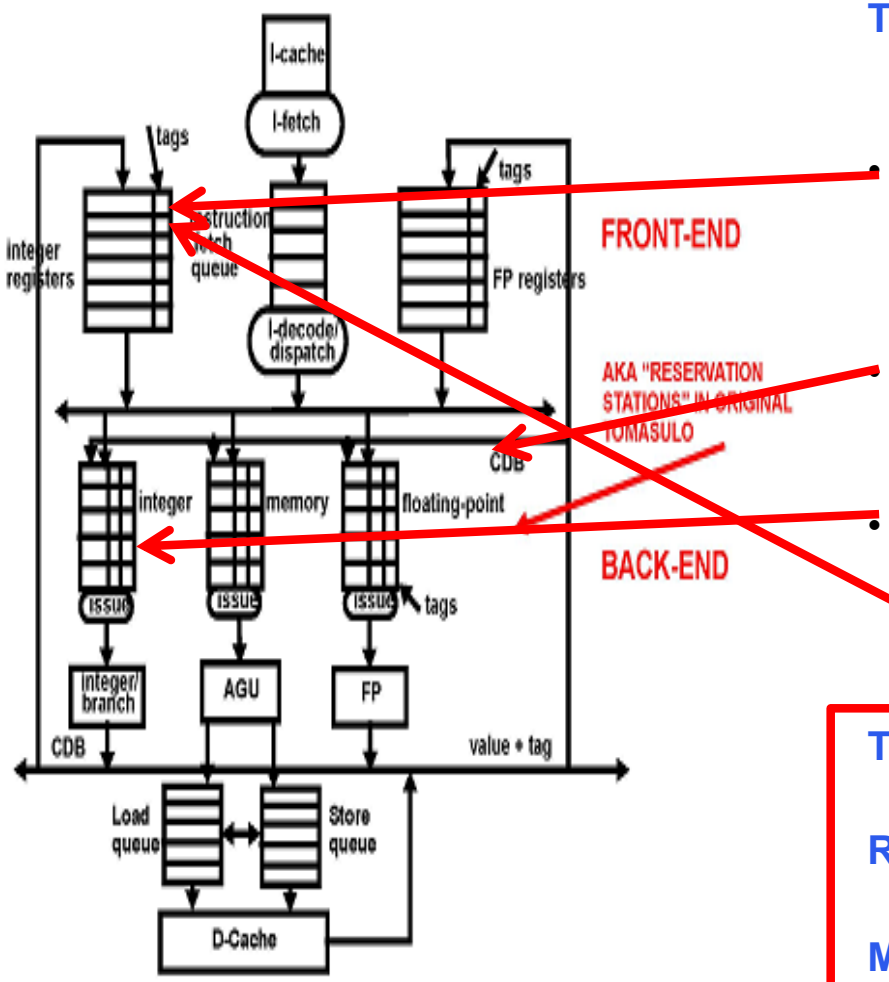
- Instructions are fetched and stored in a FIFO Queue – ”Instruction Fetch Queue” (IFQ)
- When an instruction reaches the top, it is
  - decoded and
  - dispatched to an Issue Queue (Integer/Branch, Memory or FP) even if some of its operands are being computed, i.e., not ready

Back-end

- Instructions in Issue Queues wait for their input (register) operands and are scheduled when available
- Instructions execute in their functional unit and their result is put on the common databus (CDB)
- All instructions in queues and all registers watch the CDB and grab the value they are waiting for

Tomasulo Algorithm: Hazards

Tomasulo: Hazards on Registers 2(2)



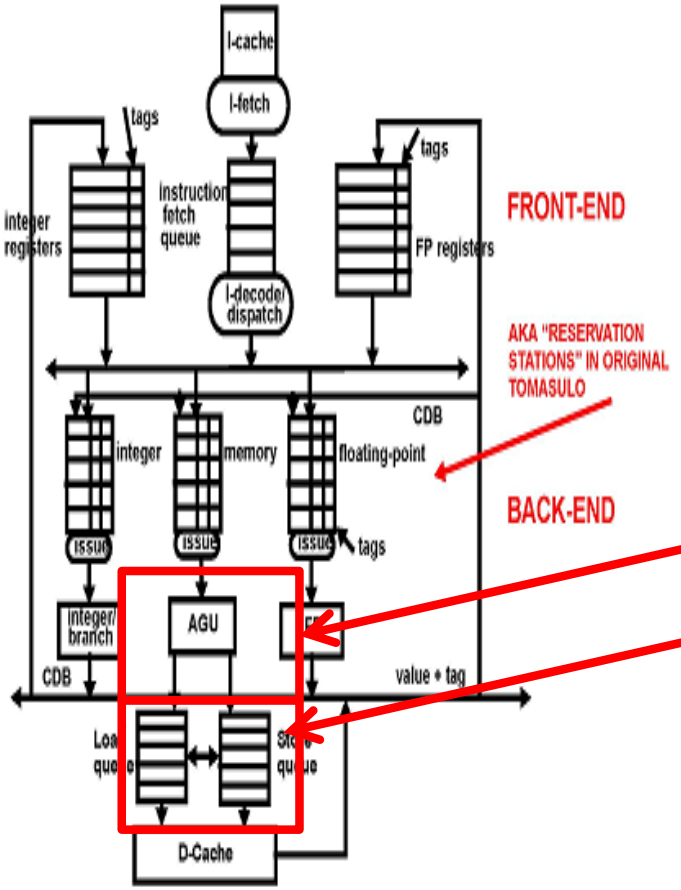
- The output register is assigned a TAG (the Q entry number where instruction is dispatched)
- The TAG is stored in the register file and is reclaimed when the instruction has written its value on the CDB and releases its Q entry
- The TAG of the result is carried by the CDB and "snooped" by queues and by register files
- When TAG match is detected the value is stored in Q or in register
  - TAG is invalidated in registers
  - Operand in instruction is valid

This is a form of *dynamic register renaming*

Register values are renamed to Q entry numbers

Multiple values for the same register may be pending at any time

Tomasulo: Memory Hazards



- All types of hazards are possible on memory (RAW, WAR, WAW)
- Load/Store Q (L/S Q): Staging buffer to solve memory hazards
  - Stores are split in 2 sub-instructions:
  1) compute address
  2) wait f. data
  - Both are dispatched to the memory and results are latched in the L/S Q
  - L/S resolves memory hazards (memory disambiguation)

Structural and Control Hazards

Structural hazards:

- I-fetch must stall if the IF Q is full
- Dispatch must stall if all entries in the issue Q or L/S Q are occupied
- Instructions cannot be issued in case of conflicts for the CDB or functional units

Control hazards:

- Dispatcher stalls when it reaches a branch instruction
- Branches are dispatched to integer issue Q as integer instr.
- They wait for their register operands and put outcome on CDB:
  - If untaken, then dispatch resumes from the IFQ
  - If taken, then dispatch clears the IFQ and directs I-fetch to fetch the target instruction stream

Precise exceptions

- Not supported

Quiz 3.2

Which of the following statements are correct

a) Tomasulo resolves structural hazards by not issuing any instruction to a queue until an empty slot in the queue is available

b) Tomasulo resolves WAR hazards by forcing instructions to read from the registerfile in program order

c) Tomasulo resolves RAW, WAR and WAW hazards by associating a TAG instead of a register to the destination operand

Discussion 1(2)

WAW and WAR dependencies are also called ”False” or ”Name” dependencies

- RAW dependencies are called ”True” dependencies
- False or Name dependencies are due to limited memory resources

Tomasulo algorithm solves WAW and WAR hazards by assigning result (output) operands a TAG – the issue Q number

Discussion 2(2)

Example:

I1: L.S F0,0(R1)

I2: ADD.S F1,F1,F0

I3: L.S F0, 0(R2)

Important observations:

- **I3 may complete its execution before I1, if I1 misses and I3 hits in cache**
- **BUT: I2 waits on the TAG of I1, not on F0 or on the TAG of I3. Thus, the WAR hazard between I2 and I3 is avoided**
- **The TAG of F0 in the register file is set to I1's TAG when I1 is dispatched and then to I3's TAG when I3 is dispatched**
- **Even if I3 completes before I1, the final value of F0 is I3's. Thus, the WAW hazard between I1 and I3 is avoided.**
- **The value of F0 produced by I1 is never stored in register. It is a fleeting value only consumed by I2**

Tomasulo: Example

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	13	(14)	14	(15)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I11 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

Integer instr.: 1 cycle
FP instr.: 5 cycles

Each entry is clock cycle number; fill table clock by clock

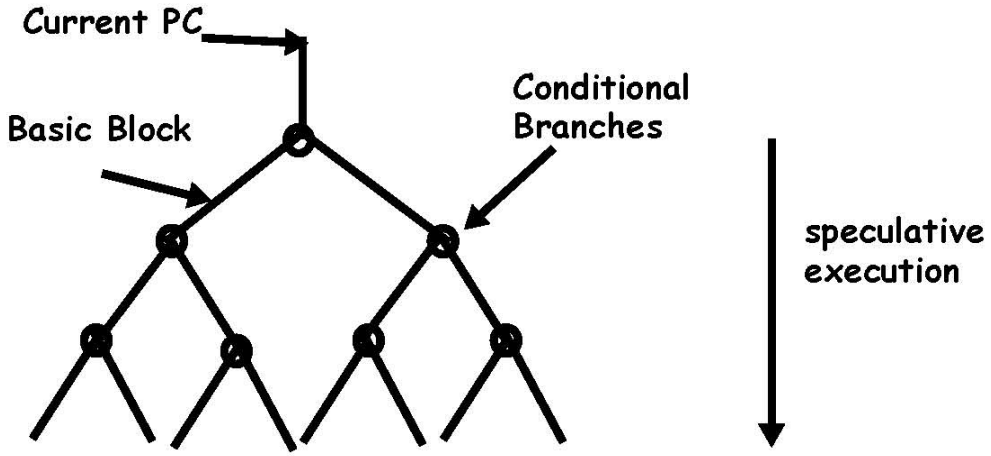
- **Instructions are issued and start execution out of (process) order**
- **Large overhead to manage instructions**
- **Latency of operation is effectively increased**
- **Structural hazards: CDB/FU conflicts**
  - Could take advantage of static scheduling (e.g., SUBI->BRANCH)
  - Branch acts as a barrier to parallelism

Dynamic Branch Prediction (Ch 3.4.2)

CHALMERS
Chalmers University of Technology

Execution Beyond Unresolved Branches

Basic block: **Block of consecutive instructions with no branch and no target of branch. Execution looks like a tree of possibilities.**



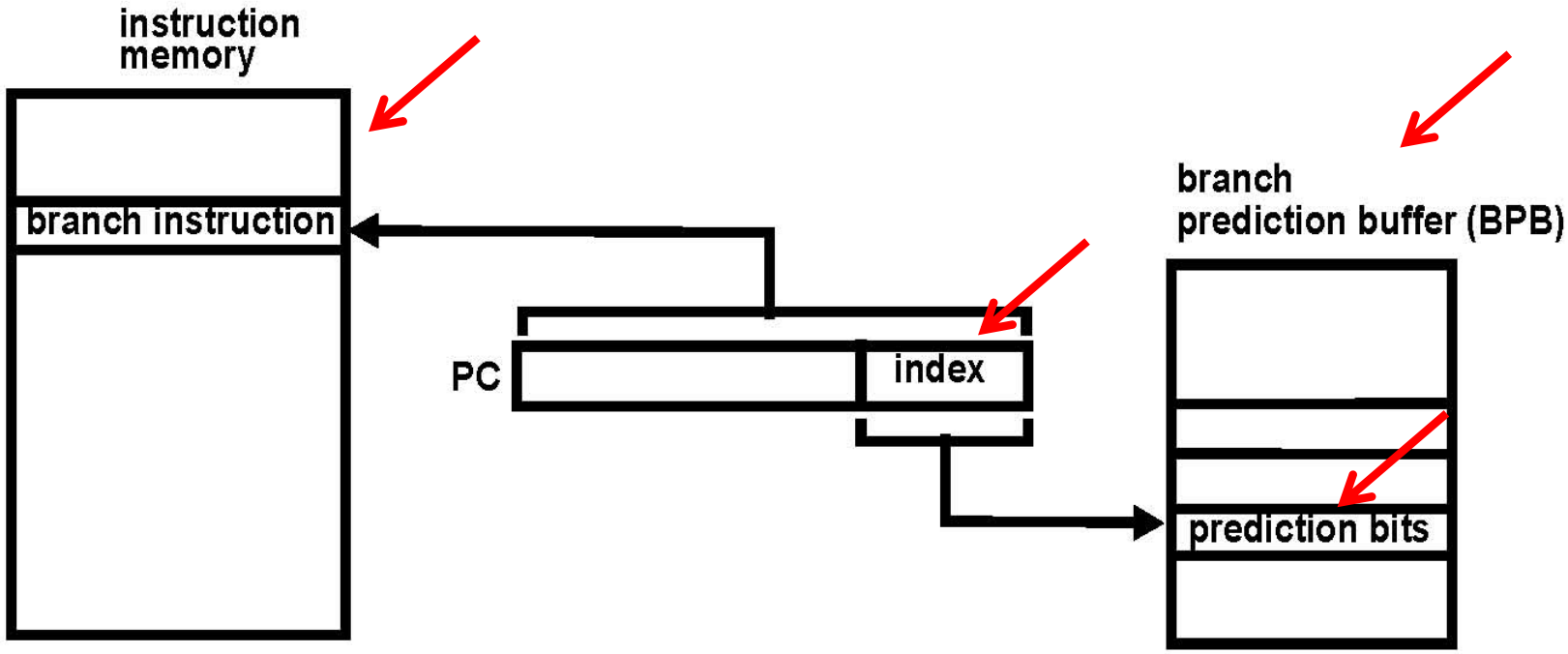
All-path execution: **Execute all paths after branch and then cancel all but one path.**

- **Very hardware intensive**
- **Hard to keep track of order of instructions in a tree**
- **Unwanted exceptions**

Predict branches and execute most likely path

Dynamic Branch Prediction 1(2)

Branch prediction buffer (BPB) accessed with instruction in I-fetch



Dynamic Branch Prediction 2(2)

Branch prediction buffer (BPB)

- •Small memory indexed with LSBs of PC in I-fetch
- •Prediction is dropped if not a branch
- •Otherwise the prediction bits are decoded into T/NT prediction
- •Once the branch condition is known and if it is incorrect
  - Rollback execution
  - Update prediction bits
- •Aliasing in BPB (different branches affect each others' predict.)

1-Bit Predictor

Each BPB entry is 1 bit

- Bit records the last outcome of the branch
- Predicts that outcome is same as last outcome

```
Loop 1:
    Loop 2:
    BEZ R2, Loop2
BEZ R3,Loop1
```

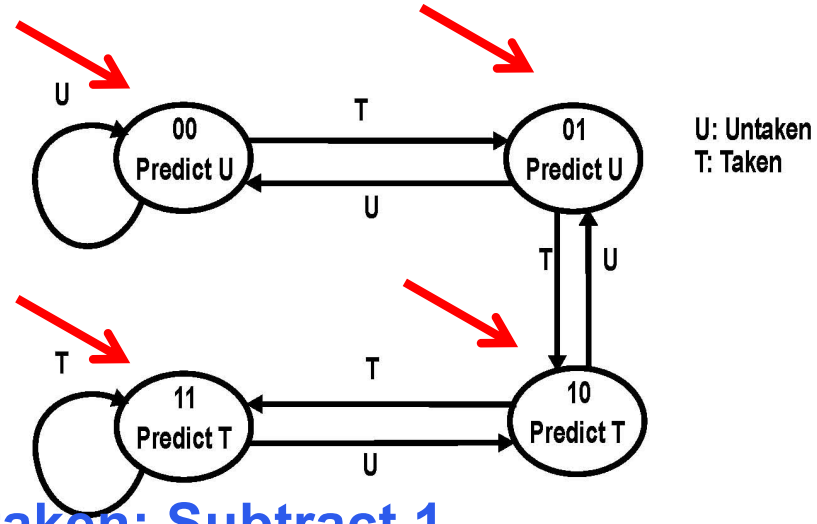
Always mispredict twice for every loop execution

- Once on entry and once on exit
- The mispredict on exit is unavoidable (loop count unknown)
- But the next mispredict on entry could be avoided

SOLUTION: Use a 2-bit predictor

2-Bit Branch Predictor

2-bit Up-Down saturating counter in each entry of the BPB



Taken: Add 1; Untaken: Subtract 1

- Now it takes 2 mispredictions in a row to change the prediction
- For the nested loop, the misprediction at entry is avoided
- Could have more than 2 bits, but two bits cover most patterns

Quiz 3.3

Assume a branch predictor with a 2-bit counter with the initial state 00. How many times does it mispredict for the following loop:

```
for (i=0; i<100; i++);
```

a) 1
b) 2
c) 3
d) 4

What you should know by now

Dynamically scheduled pipelines

- Tomasulo algorithm
- Pipeline structure
- How to avoid RAW, WAR, WAW register and memory hazards
- Simple dynamic branch prediction techniques