CS5375 Computer Systems Organization and Architecture

Lecture 12

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Announcements

- Programming project #1

- How to analyze L2 miss/hit rate
  - Count total CPU requested memory accesses (*m*), L2 accesses (*n*) (i.e., L1 misses), and L2 misses (*s*) (i.e., actual memory accesses)
  - Then calculate L2 global miss rate and L2 local miss rate
    - L2 global miss rate == *s* / *m*
    - L2 local miss rate == *s* / *n*
  - L2 global hit rate == 1 - L2 global miss rate
  - L2 local hit rate == 1 - L2 local miss rate

Review of Last Lecture

- Instruction-level parallelism (ILP)
  - Hazards: situations that prevent starting the next instruction in the next cycle
  - Structural hazard
    - A required resource (datapath element) is busy, i.e., conflict for use of a resource
  - Data hazard
    - Read after write (RAW): True dependence
    - Write after read (WAR): Antidependence (a name dependence)
    - Write after write (WAW): Output dependence (a name dependence)
  - Resolving data hazard for improving ILP
    - Forwarding
    - Code scheduling to avoid stalls
    - Loop Unrolling

Outline

- Control Hazards and Branch Prediction
- Exploiting ILP Using Multiple Issue and Static Scheduling

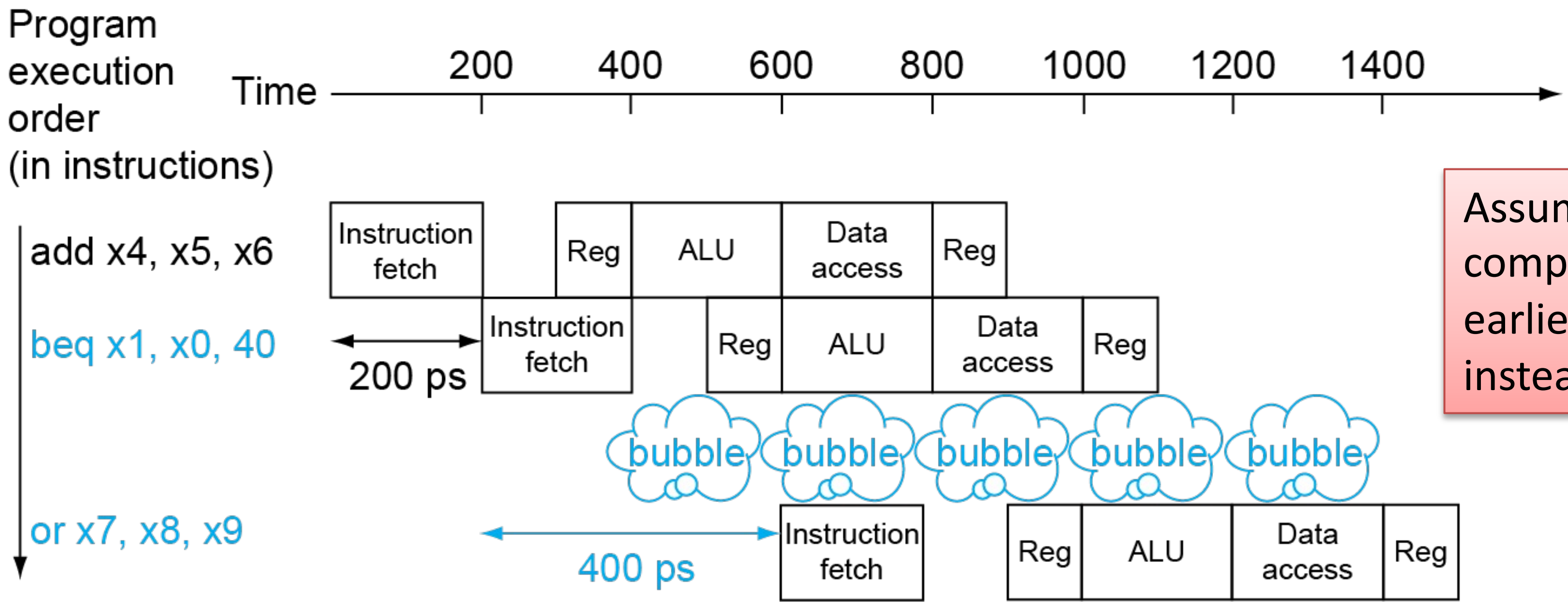
Control Hazards (or Branch Hazards)

- Branch determines the flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction, as still working on ID stage of branch

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Assume extra hardware can compute branch target earlier in the ID stage instead of the EX stage

- But even with added hardware to compare registers, compute target and update PC in the ID stage, there is still a pipeline stall for every conditional branch – need additional solution

Performance of “Stall on Branch”

- Conditional branches are 10% of the instructions executed in SPECint2006 and assume instructions have a CPI of 1
- If every conditional branch took one extra clock cycle for the stall
- Then weighted average CPI == 1.10
  - i.e., a slowdown of 1.10 versus the ideal case
- Stall penalty becomes too high and unacceptable for longer pipelines can’t readily determine branch outcome early

Solution: Branch Prediction

- The solution is to predict the outcome of branch
  - If correct, it doesn't slow down the pipeline
  - Only stall if prediction is wrong

- In RISC-V pipeline
  - A simple solution is to predict branches not taken (i.e., take the next instruction, PC = PC + 4)
  - Fetch instruction after branch, with no delay

Branch Prediction

Prediction correct (branch untaken)

Prediction incorrect (branch taken)

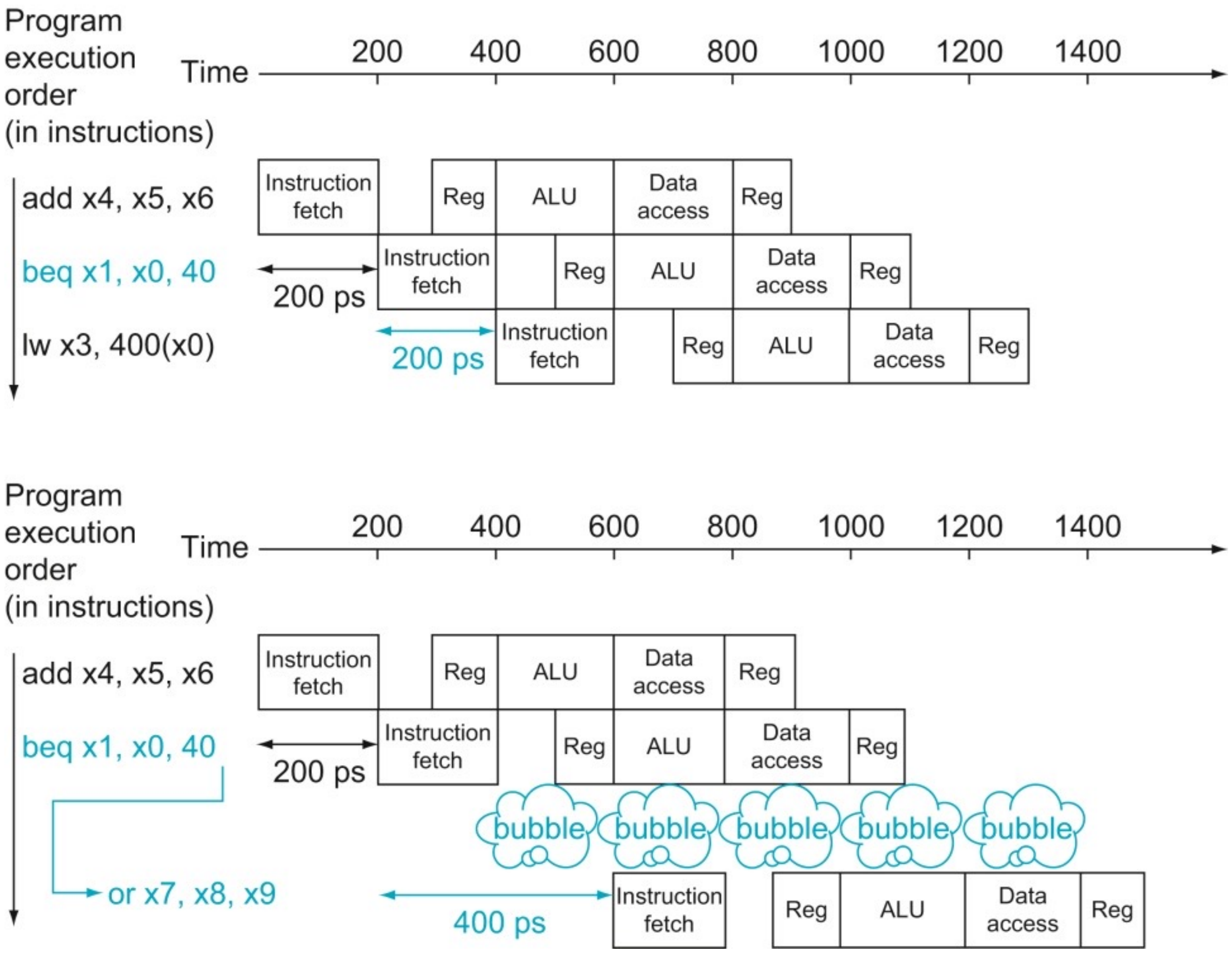


Figure 4.34 Predicting that branches are not taken as a solution to control hazard

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop branches (more likely more than one iteration)
    - Always predict backward branches taken (jump back)
    - Or, always predict forward branches not taken

- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

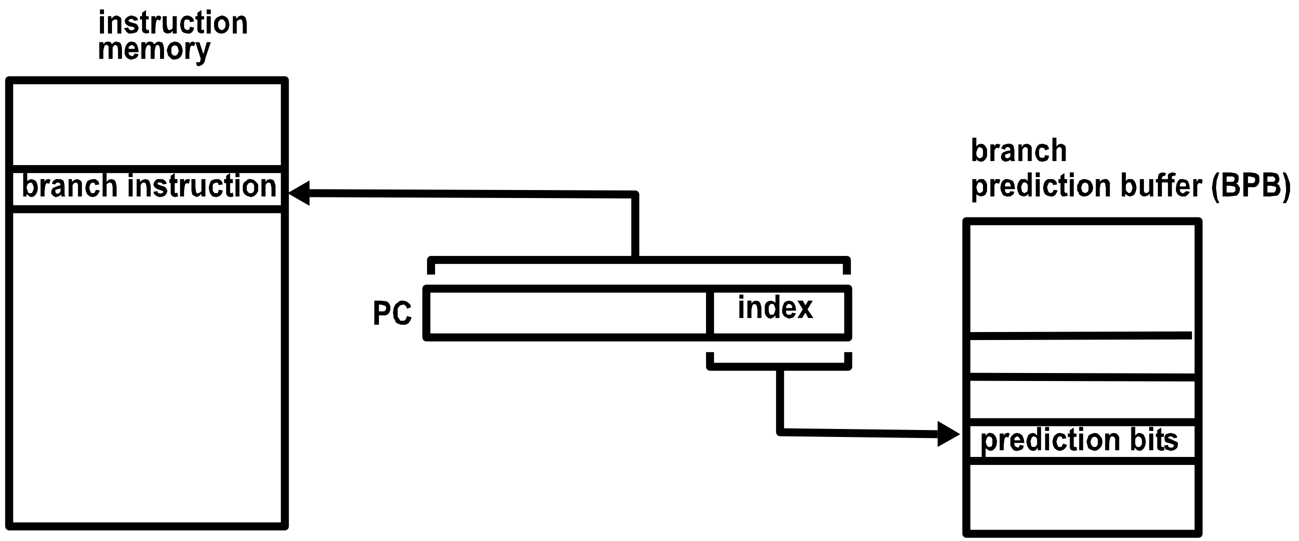
Dynamic Branch Prediction

- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities

- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - Important branches in programs that have dynamic behavior

Dynamic Branch Prediction

- Branch Prediction Buffer (BPB) accessed with Instruction on I-Fetch



- Also called Branch History Table (BHT), Branch Prediction Table (BPT)

1-bit Predictor

- Each BHT entry is 1-bit
  - Bit records last outcome of the branch
  - Predicts that next outcome is the same as the last

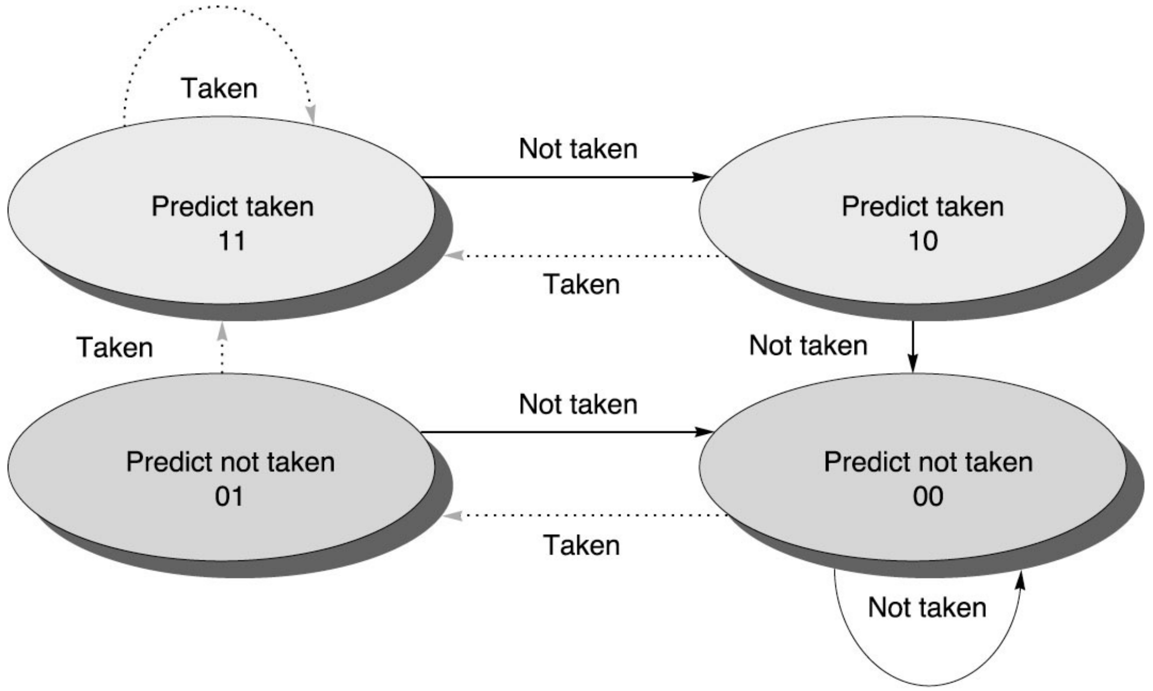
```
for (i=4; i>=0; i=i-1)
  x[i] = x[i] + s;
```

- Always mispredicts twice for every run of this loop
  - Once on entry and once on exit
  - Poor for small loops

```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop
```

2-bit Predictor

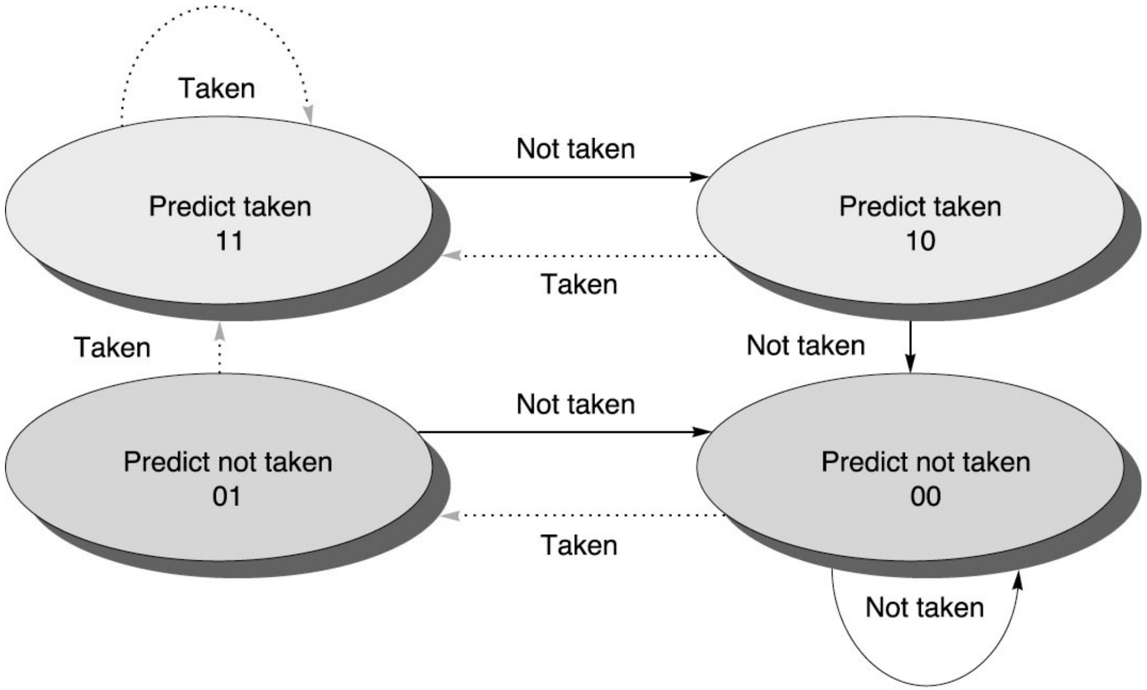
- Prediction must miss twice before it is changed



- 2-bit BHT
- Also called 2-bit saturating counter
- Can be extended to N-bits (typically N=2)

2-bit Predictor

- Prediction must miss twice before it is changed

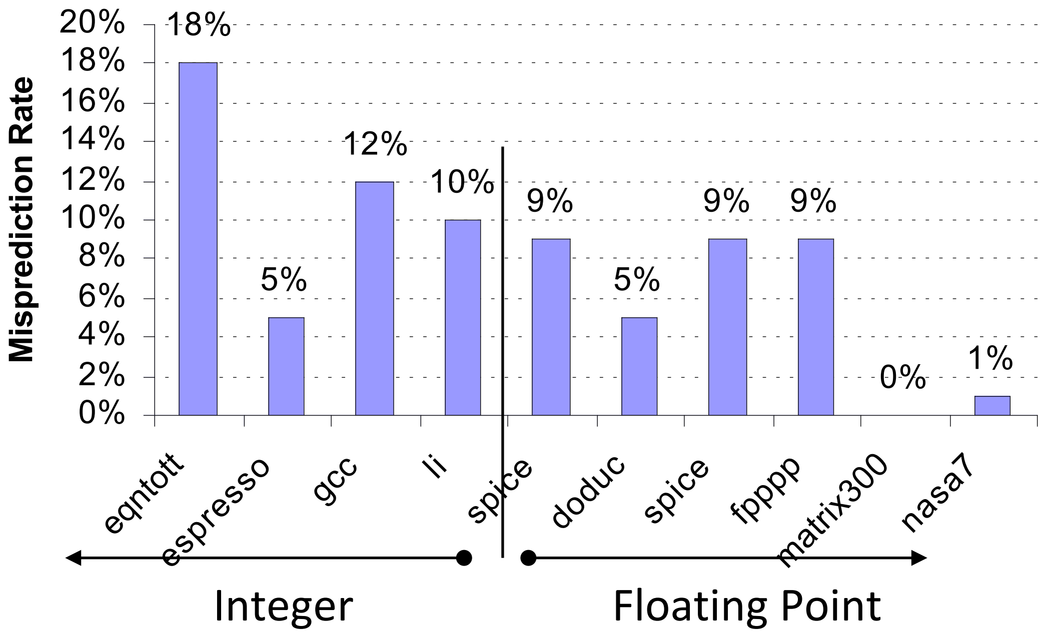


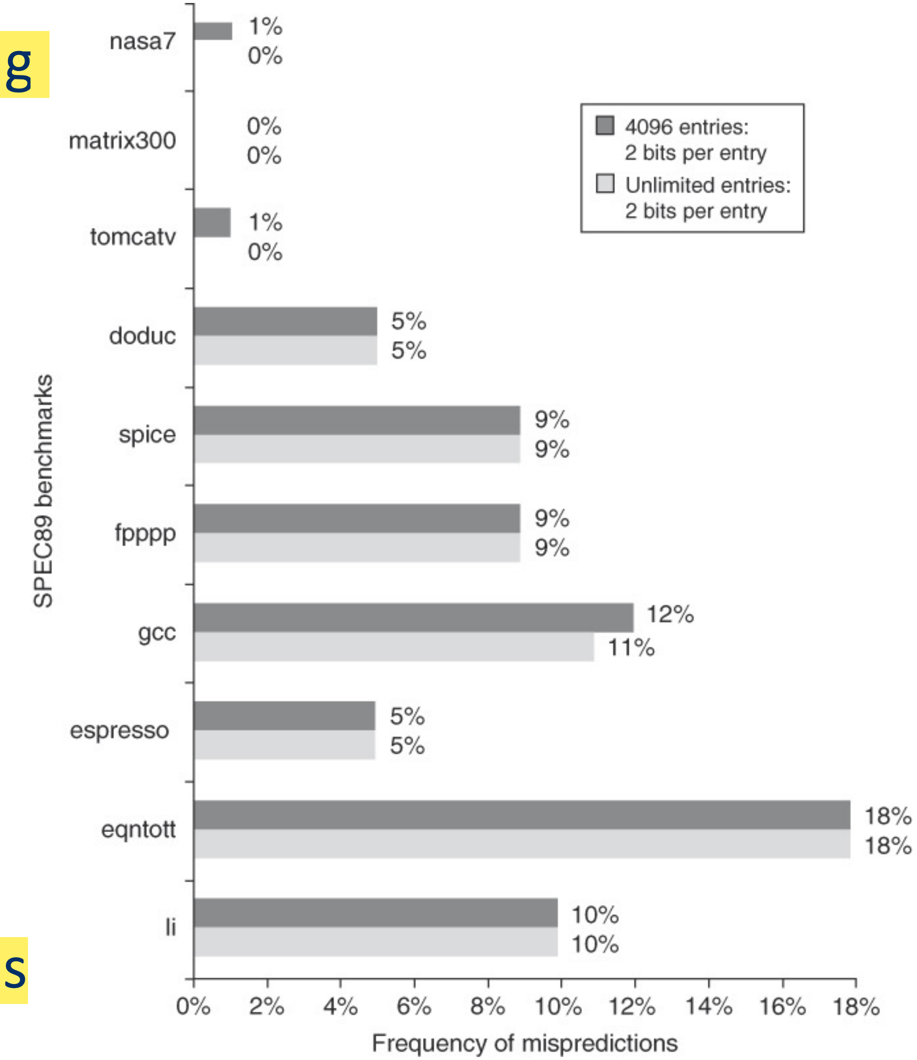

```
for (i=4; i>=0; i=i-1)
  x[i] = x[i] + s;
```

```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop
```

Observations

- Misprediction higher for integer programs than floating point programs





- Prediction accuracy doesn't improve beyond 4K entries

Correlating Predictors

- Look at other branches for clues

https://www.geeksforgeeks.org/correlating-branch-prediction/

```
if (aa==2)          -- branch b1
    …
if (bb==2)          -- branch b2
    …
if(aa!=bb) { …      -- branch b3 – Clearly depends on the results of b1 and b2
```

Correlating Predictors

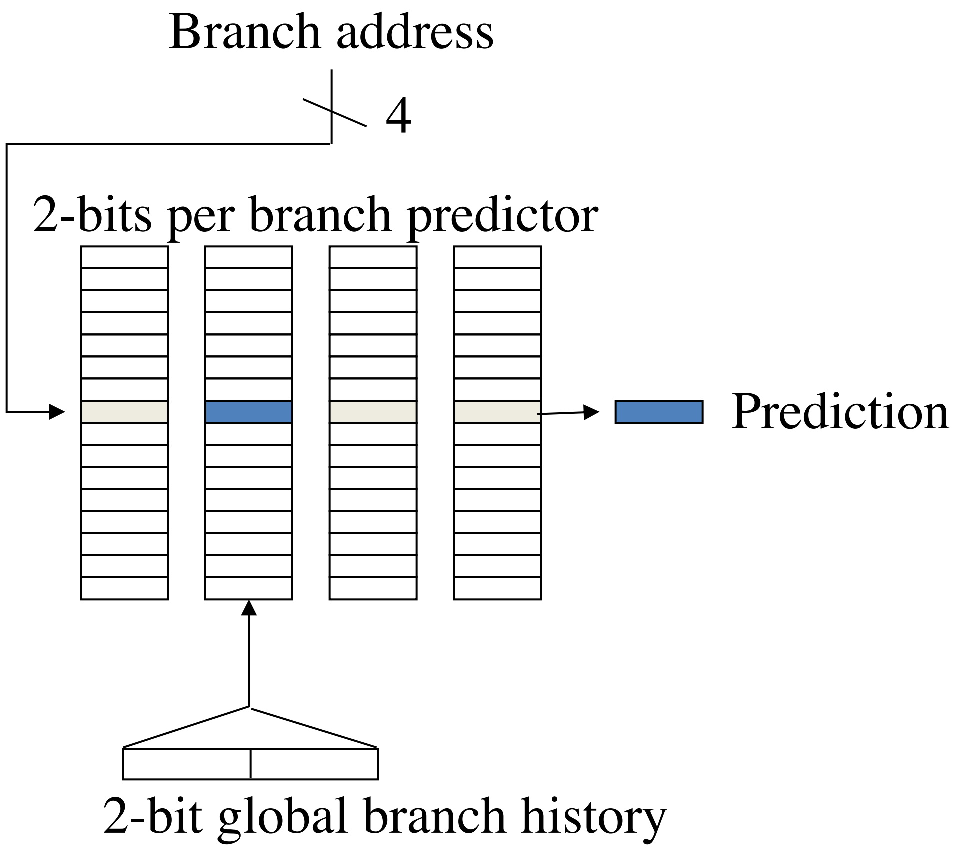
- Record *m* most recently executed branches as taken / not taken and use that pattern to select proper *n*-bit branch history table

- (*m*,*n*) predictor
  - Record last *m* branches to select between 2^m BHT
  - Each BHT has *n*-bit counters
    - Simple 2-bit BHT is a (0,2) predictor

- Global Branch History: *m*-bit shift register
- Also called Two Level predictors

Correlating Predictors

- Example (2,2) predictor



A gshare Correlating Predictor

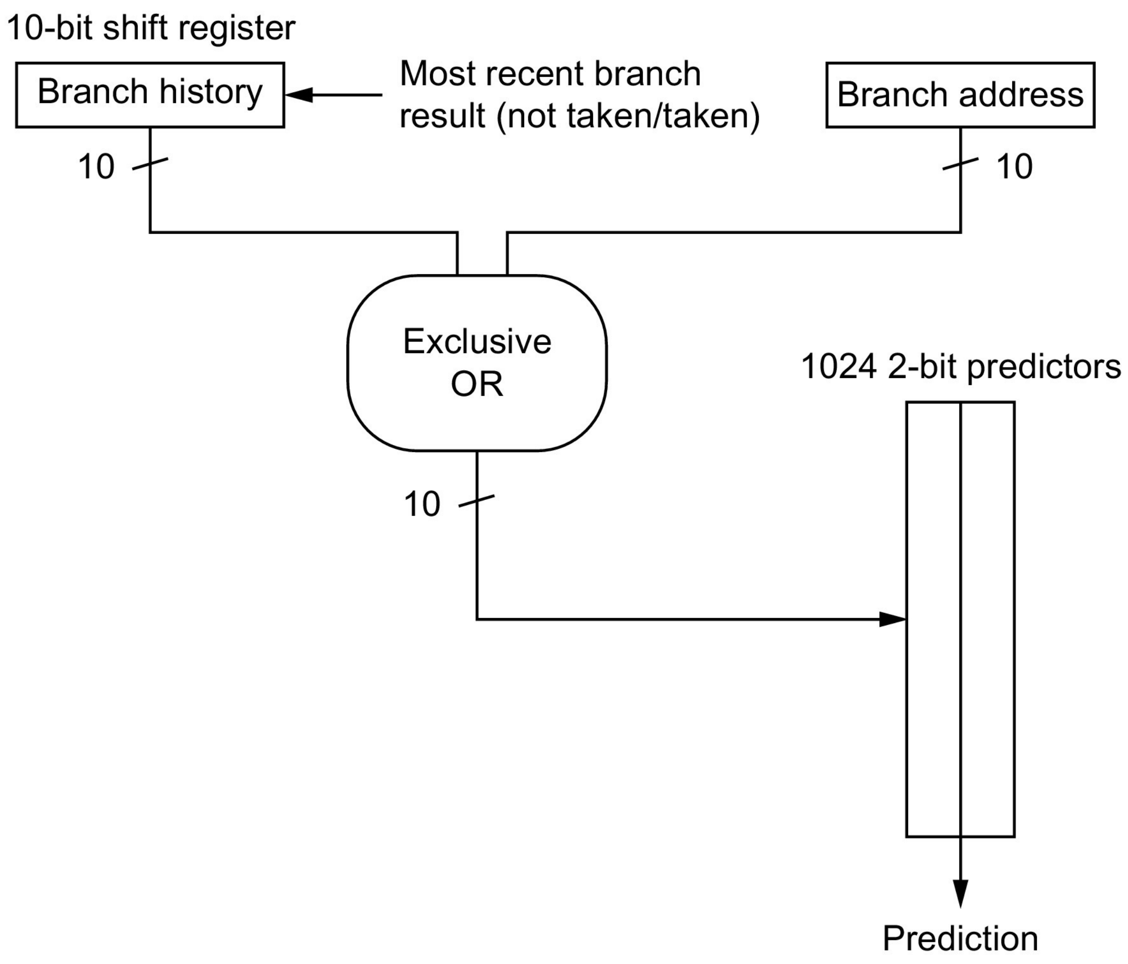
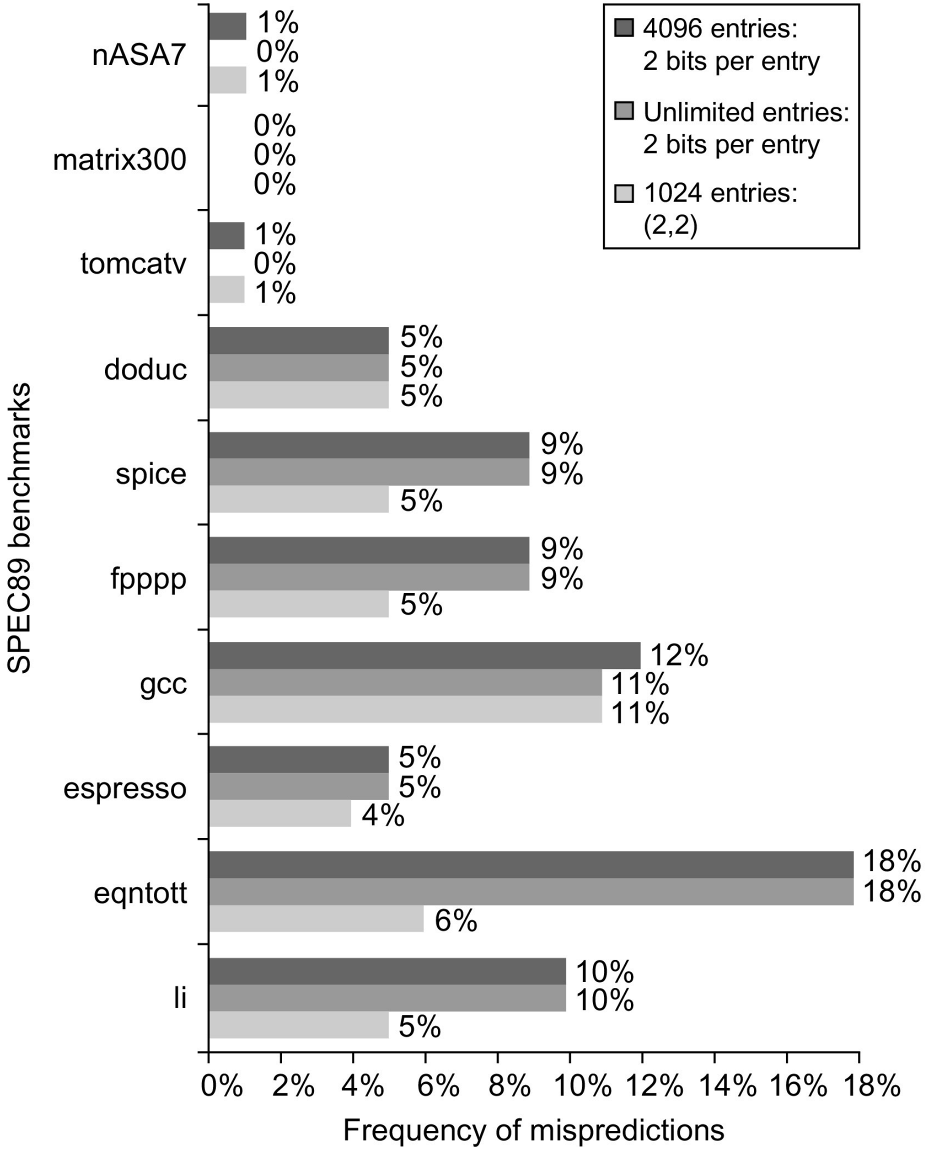


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

Correlating Predictor Accuracy

first 2 is for last 2 branches result(11,10,01,00)
second 2 is for combine it with 2 bit predictor

- With 1K entries, (2,2) performs better than 2-bit predictor with unlimited entries!



Global Predictor v.s. Local Predictor

- Previously, Global Branch History captures global behaviors (global predictor)
  - Patterns including neighboring branches

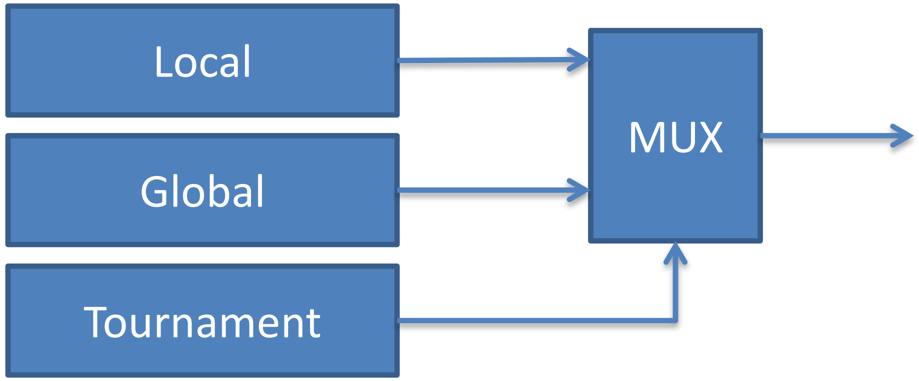
- Local predictor capture patterns belonging to the branch being predicted

```
if (aa==2)          --  branch b1
    …
if (bb==2)          --  branch b2
    …
if(aa!=bb) { …      --  branch b3
```

Tournament Predictors

- Problem: some branches work well with local predictors, while other branches work well with global predictors

- Solution: use multiple predictors
  - One based on global information, one based on local information
  - Add a selector to pick between predictors



Tournament Predictor

- How to pick between local or global predictor?
- Use saturating counter to choose between predictors

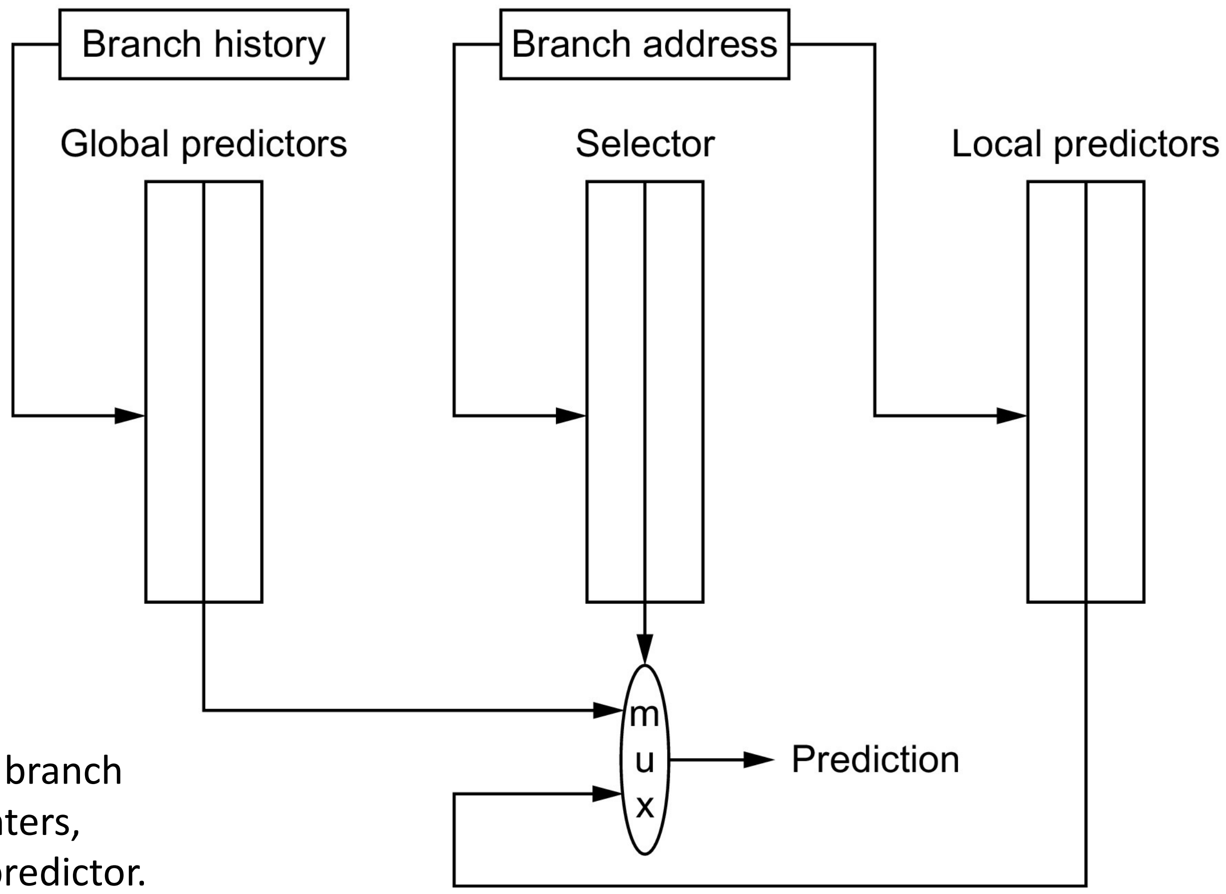
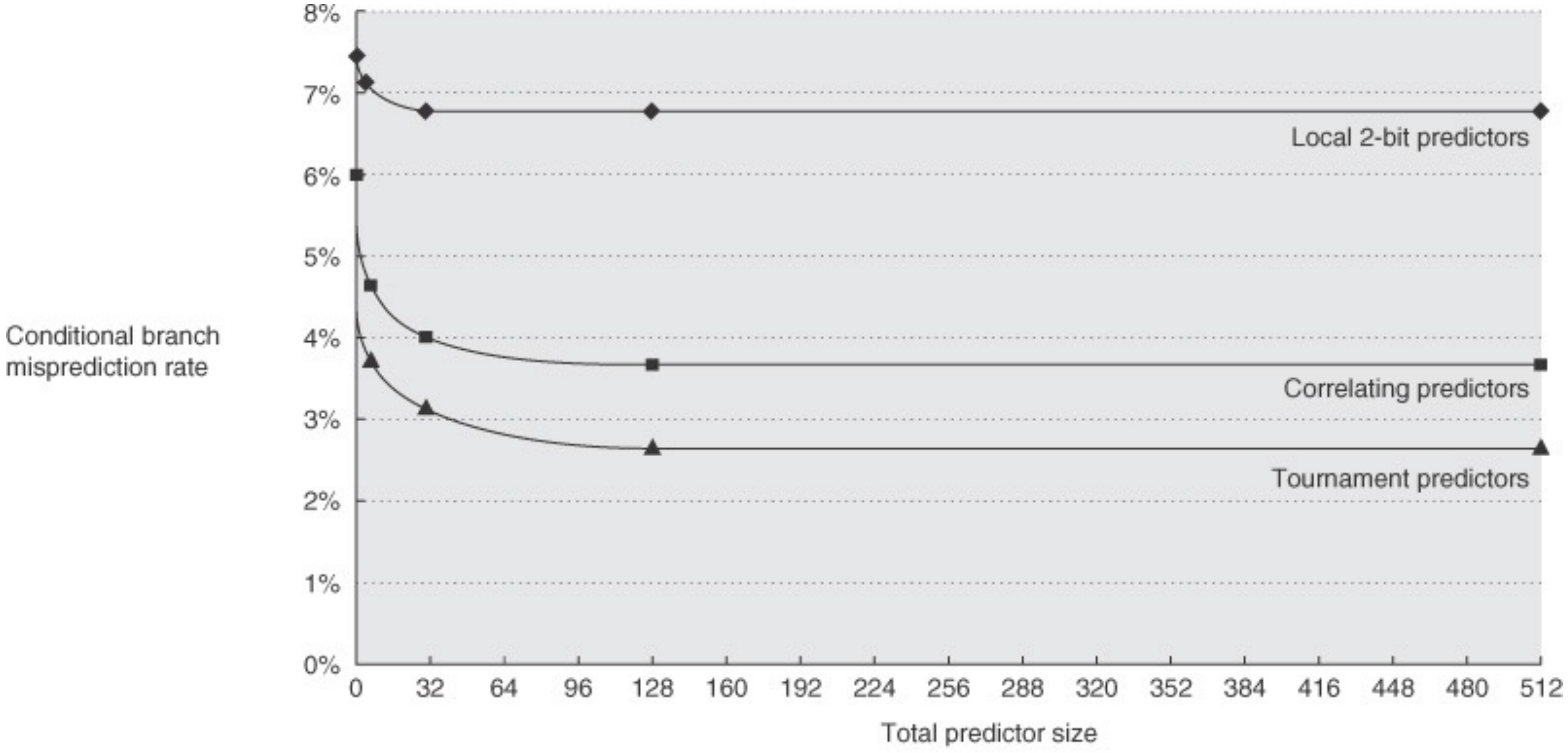


Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor.

Predictor Accuracy



Outline

- Control Hazards and Branch Prediction
- Exploiting ILP Using Multiple Issue and Static Scheduling

Multiple issue and Static Scheduling

- To further increase ILP
  - Multiple issue (N-way multi-issue CPU)
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines (i.e., multiple washers, dryers, “folders”, “storers” in the laundry example)
    - Start multiple instructions per clock cycle
  - Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate, or in other words, CPI < 1
  - Thus it’s also often to use Instructions Per Cycle (IPC), the inverse of CPI
  - E.g., 4-way multiple-issue 4GHz CPU
    - Peak CPI = 0.25, peak IPC = 4, 16 BIPS (billion instructions per second)
  - But hazards (dependencies) reduce this in practice

Multiple Issue

- Static multiple issue (at compile time, “static”)
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- Dynamic multiple issue (during execution, “dynamic”)
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Static Multiple Issue

- Compiler groups instructions into "issue packets"
  - Issue packet: a set of instructions that can be issued together in one clock cycle
    - E.g., an arithmetic instruction and a load/store instruction, if there's no dependence/no hazards

- An issue packet can be considered as a very long instruction
  - Specifies multiple concurrent operations
    $\Rightarrow$ Very Long Instruction Word (VLIW)

  - VLIW also refers to a style of ISA that launches many operations that are defined to be independent in a single wide instruction (typically with many separate opcode fields)

RISC-V with Static Dual Issue (Two-way/Two-issue)

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - Fetching and decoding 64 bits of instructions (two instruction words)
    - 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

Two-issue pipeline example (v.s. single-issue pipeline in the earlier lecture)

Hazards in the Dual-Issue RISC-V

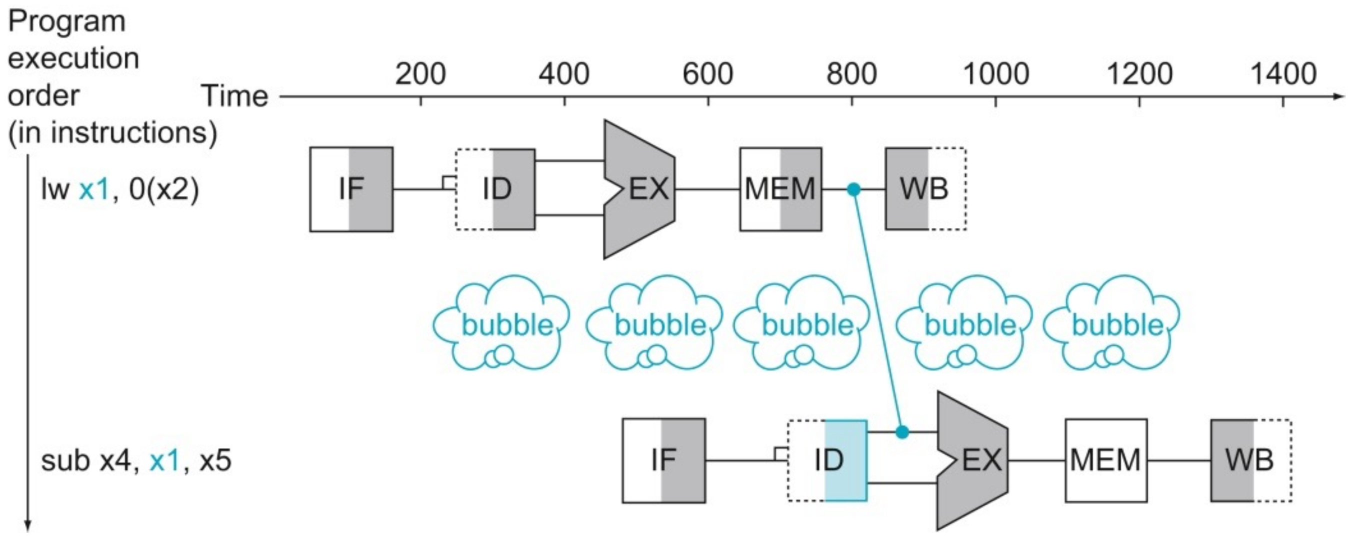
- Ideally two-issue processor can improve performance by up to a factor of two

- Would require more instructions executing in parallel
  - This increases the relative performance loss from data and control hazards

- E.g., data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  x10, x0, x1
      lw   x2, 0(x10)
      ```
    - Split into two packets, effectively a stall
      ```

Hazards in the Dual-Issue RISC-V (cont.)

- Another example: load-use hazard
  - A load has a use latency of one clock cycle, which means an instruction that can use the result of the load has to be one cycle later
  - Thus, in a two-issue processor, next two instructions cannot use the load result without stalling
  - In general, the speedup is less than 2 for a two-issue processor
  - More aggressive scheduling required



Load-use data hazard

Compiler Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: lw   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sw   x31,0(x20)     // store result
      addi x20,x20,-4     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

```
for (i=99; i>=0; i--)
  a[i] = a[i] + s;
```

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

CPI = 4 cycles/5 instructions = 0.8 (v.s. an ideal case of 0.5)

Or, IPC = 5/4 = 1.25 (v.s. a peak IPC = 2)

Loop Unrolling and Scheduling

- Replicate loop body to expose more parallelism
- Loop unrolling example

```
for (i=99; i>=0; i=i-4){
    a[i] = a[i] + s;
    a[i-1] = a[i-1] + s;
    a[i-2] = a[i-2] + s;
    a[i-3] = a[i-3] + s;
    }
```

Since x20 is decremented by 16 (unrolling 4 iterations), these addresses are the original value of x20 minus 4, minus 8, and minus 12

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-16	lw x28, 0(x20)	1
	nop	lw x29, 12(x20)	2
	add x28,x28,x21	lw x30, 8(x20)	3
	add x29,x29,x21	lw x31, 4(x20)	4
	add x30,x30,x21	sw x28, 16(x20)	5
	add x31,x31,x21	sw x29, 12(x20)	6
	nop	sw x30, 8(x20)	7
	blt x22,x20,Loop	sw x31, 4(x20)	8

- IPC?

  IPC = 14/8 = 1.75

  Closer to 2, but at cost of registers and code size

Loop Unrolling and Scheduling (cont.)

- These sequences of instructions in this loop are actually completely independent (adding a scalar value to each element of an array)

```
Loop: lw   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sw   x31,0(x20)      // store result
      addi x20,x20,-4      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

- Except using x31 register in each loop iteration, which causes an enforced ordering by the reuse of a name (x31 register)
  - Store followed by a load of the same register, i.e. WAR or antidependence (a name dependence)
- Loop unrolling uses register renaming, i.e. 4 temporary registers rather than one, to resolve antidependence
  - Significant increase in code size, but delivers better performance (IPC = 1.75 or CPI = 0.57)

Readings

- Chapter 3, 3.3, 3.7