CS510 Computer Architecture

Lecture 09: Dynamic Scheduling

Soontae Kim

Spring 2016

School of Computing, KAIST

Correlating Branches

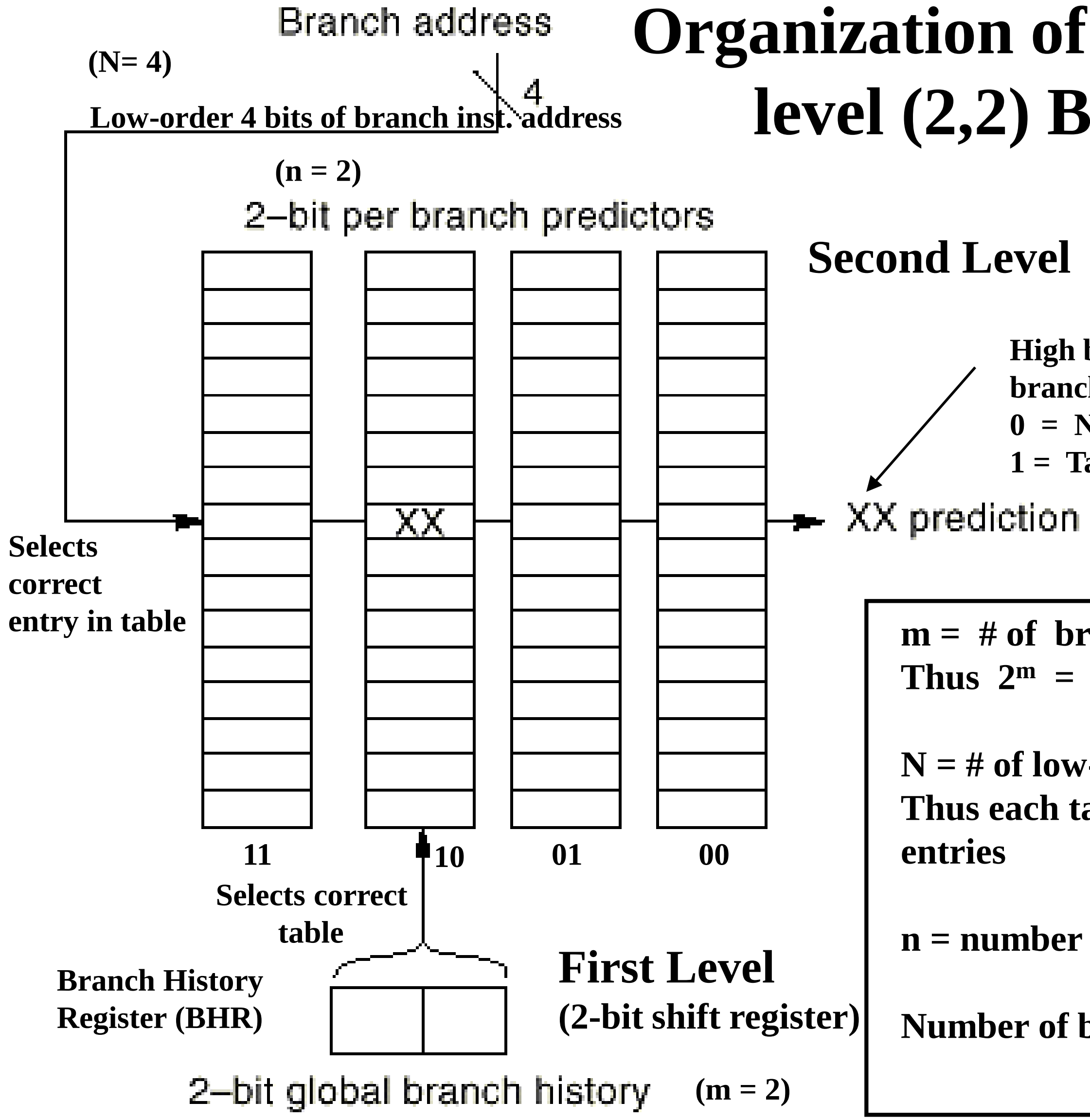
Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

Example:

			DSUBUI	R3, R1, #2	
B1	if (aa==2)		BNEZ	R3, L1	; b1 (aa!=2)
	aa=0; (not taken)		DADD	R1, R0, R0	; aa==0
		L1:	DSUBUI	R3, R2, #2	
B2	if (bb==2)		BNEZ	R3, L2	; b2 (bb!=2)
	bb=0; (not taken)		DADD	R2, R0, R0	; bb==0
		L2:	DSUBU	R3, R1, R2	; R3=aa-bb
B3	if (aa!==bb){ taken)		BEQZ	R3, L3	; b3 (aa==bb)

Branch <u>B3</u> is <u>correlated</u> with branches B1, B2. If <u>B1, B2 are both not taken, then B3 will be taken</u>. Using only the behavior of one branch cannot detect this behavior.

Organization of A Correlating Two-level (2,2) Branch Predictor



하위 4bit가 동일하면 같은 index의 branch prediction을 share한다. (flush 개념 x)

m = # of branches tracked at first level = 2
Thus $2^m = 2^2 = 4$ tables at second level

N = # of low-order bits of branch address = 4
Thus each table at 2nd level has $2^N = 2^4 = 16$ entries

n = number of bits of 2nd level table entry = 2

Number of bits for 2nd level = $2^m \times n \times 2^N$
$= 4 \times 2 \times 16 = 128$ bits

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.

Dynamic Branch Prediction: Example (continued)

```
if  (d==0)
       d=1;
if  (d==1)
```

```
       BNEZ     R1, L1        ;  branch b1 (d!=0)
       DADDIU   R1, R0, #1    ;  d==0, so d=1
L1:    DADDIU   R3, R1, # -1
       BNEZ     R3, L2        ;  branch b2 (d!=1)
. . .
L2:
```

Combinations and meaning of the taken/not taken prediction bits.

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

correlation is set to not taken initially

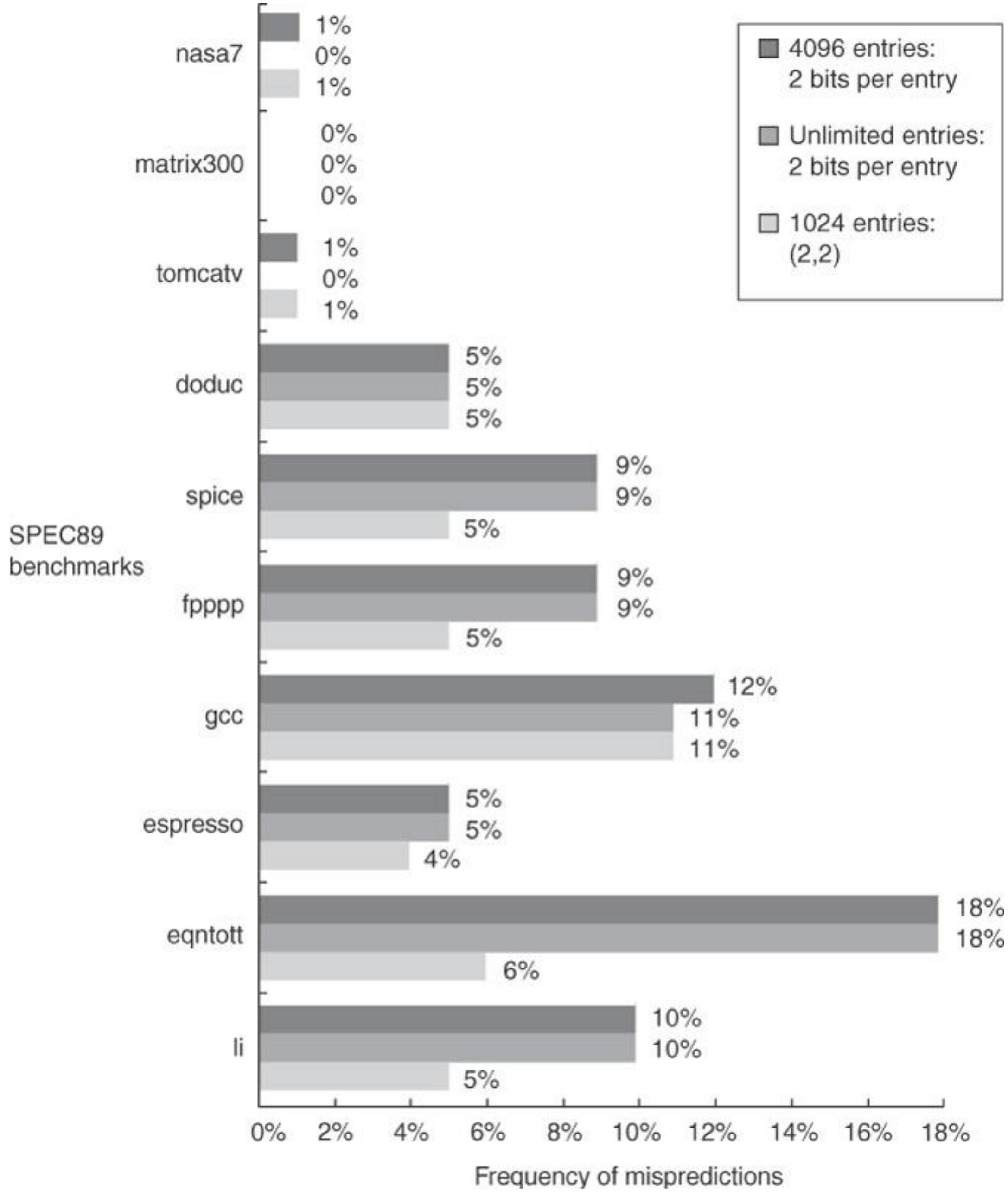
The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
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

history가 not taken / taken

prediction이 틀렸을때 update, 맞았을때 그대로 stay

Two level (1,1)



4096 entries:
2 bits per entry
Unlimited entries:
2 bits per entry
1024 entries:
(2,2)
SPEC89
benchmarks
nasa7
matrix300
tomcatv
doduc
spice
fpppp
gcc
espresso
eqntott
li
1%
0%
1%
0%
0%
0%
1%
0%
1%
5%
5%
5%
9%
9%
5%
9%
9%
5%
12%
11%
11%
5%
5%
4%
18%
18%
6%
10%
10%
5%
0% 2% 4% 6% 8% 10% 12% 14% 16% 18%
Frequency of mispredictions

Reduction of Data Hazards Stalls with Dynamic Scheduling

- So far we have dealt with data hazards in instruction pipelines by:
  - Data forwarding (register bypassing) to reduce or eliminate stalls needed to prevent RAW hazards as a result of true data dependence.
  - Hazard detection hardware to stall the pipeline starting with the instruction that uses the result.
  - Compiler-based static pipeline scheduling to separate the dependent instructions minimizing actual hazard-prevention stalls in scheduled code.
    - Loop unrolling to increase basic block size: More ILP.
- Dynamic scheduling:
  - Uses a hardware-based mechanism to reorder or rearrange instruction execution order to reduce stalls dynamically at runtime.
    - Better dynamic exploitation of instruction-level parallelism (ILP).

    some dependency는 static time때 detection되지 않고, dynamic time때 detection 되기도 한다.
  - Enables handling some cases where instruction dependencies are unknown at compile time (ambiguous dependencies).

    memory 주소의 경우 register에 그 값이 담겨 있는 경우가 있어서 static time때 알수 없는 경우가 있다.
  - Similar to the other pipeline optimizations above, a dynamically scheduled processor cannot remove true data dependencies, but tries to avoid or reduce stalls.

Dynamic Pipeline Scheduling: *The Concept*

(Out-of-order execution)

- **Dynamic pipeline scheduling overcomes the limitations of in-order pipelined execution by allowing out-of-order instruction execution.**
- **Instructions are allowed to start executing out-of-order as soon as their operands are available.**
  - **Better dynamic exploitation of instruction-level parallelism (ILP).**

Example:

True Data Dependency

```
DIV.D   F0, F2, F4
ADD.D   F10, F0, F8
SUB.D   F12, F8, F14
```

Does not depend on DIV.D or ADD.D

In the case of in-order pipelined execution, SUB.D must wait for DIV.D to complete which stalled ADD.D before starting execution.
In out-of-order execution SUB.D can start as soon as its operands F8, F14 are available.

- **This implies allowing out-of-order instruction completion.**
- **May lead to imprecise exceptions if an instruction issued earlier raises an exception.**
  - **This is similar to pipelines with multi-cycle floating point units.**

Dynamic Pipeline Scheduling

- Dynamic instruction scheduling is accomplished by:
  - Dividing the Instruction Decode stage into two stages:

Always done in program order

- **Issue**: Decode instructions, check for structural hazards.
  - A record of data dependencies is constructed as instructions are issued
  - This creates a dynamically-constructed dependence graph for the window of instructions being processed in the CPU.

Can be done out of program order

- **Read operands**: Wait until data hazard conditions, if any, are resolved, then read operands when available (then start execution)

(All instructions pass through the issue stage in order but can be stalled or pass each other in the read operands stage).

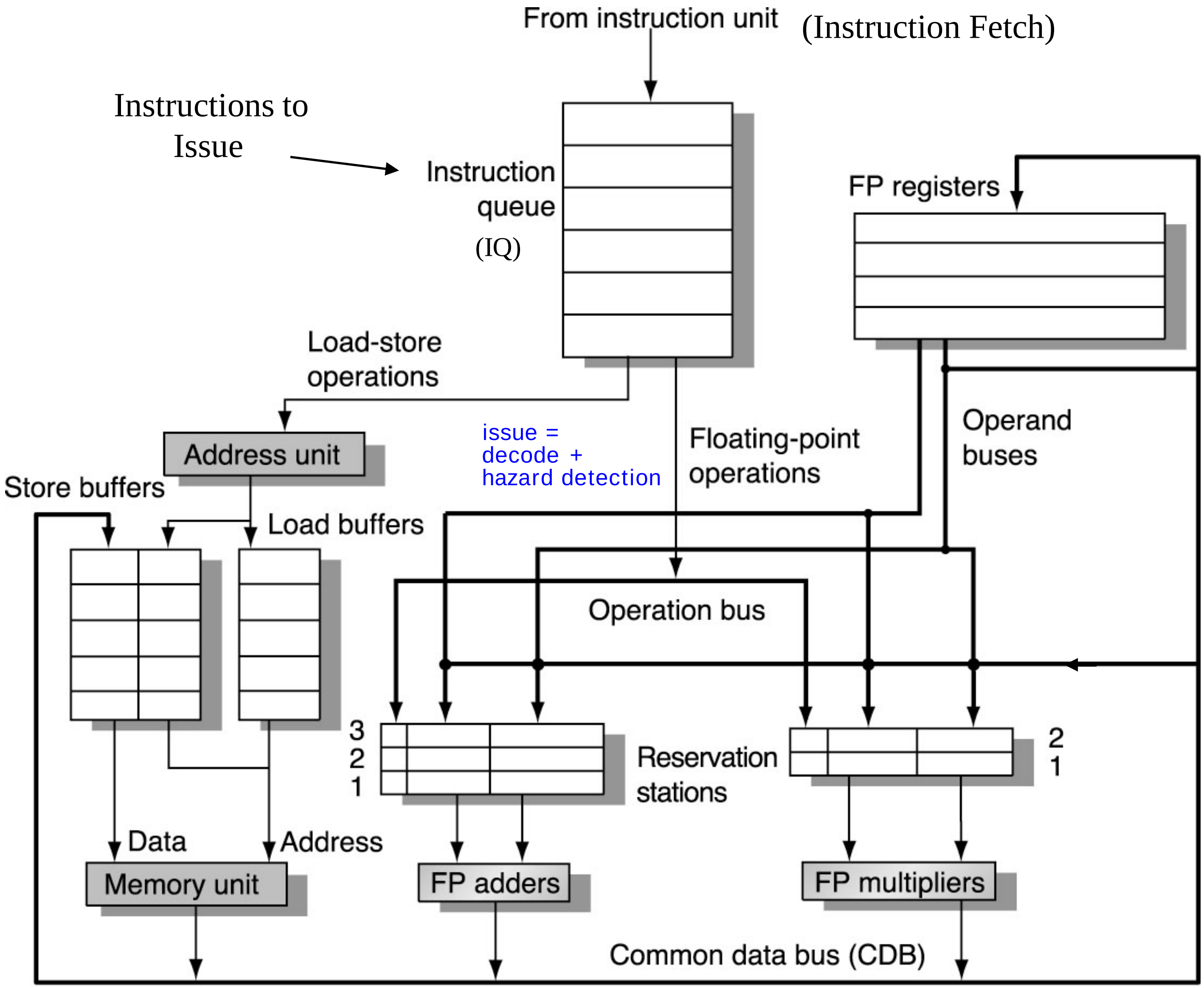
  - In the instruction fetch stage, fetch an additional instruction every cycle into a latch or several instructions into an instruction queue.
  - Increase the number of functional units to meet the demands of the additional instructions in their EX stage.
- Two approaches to dynamic scheduling
  - Scoreboard
  - Tomasulo

Dynamic Scheduling: The Tomasulo Algorithm

- **Developed at IBM and first implemented in IBM's 360/91 mainframe in 1966, about 3 years after the debut of the scoreboard in the CDC 6600.**
- **Dynamically schedule the pipeline in hardware to reduce stalls.**
- **Differences between IBM 360 & CDC 6600 ISA.**
  - **IBM has only 2 register specifiers/instr vs. 3 in CDC 6600.**
  - **IBM has 4 FP registers vs. 8 in CDC 6600 (part of ISA).**
- **Current CPU architectures that can be considered descendants of the IBM 360/91 and that implement and utilize a variation of the Tomasulo Algorithm include:**

RISC CPUs: Alpha 21264, HP 8600, MIPS R12000, PowerPC G4 ..
RISC-core x86 CPUs: AMD Athlon, Intel Pentium III, 4, Xeon, ….

Dynamic Scheduling: The Tomasulo Approach



The basic structure of a MIPS floating-point unit using Tomasulo's algorithm

Reservation Station Fields

- **Op** **Operation to perform in the unit (e.g., + or –)**
- **Vj, Vk** **Values of Source operands S1 and S2**
  - **Store buffers have a single V field indicating result to be stored.**

reservation stations number 저장 (example에서 0~15)

- **Qj, Qk** **Reservation stations producing source registers.**
  - **No ready flags; Qj,Qk=0 => ready.**
  - **Store buffers only have Qi for RS producing a result.**
- **A:** **Address information for loads or stores. Initially immediate field of instruction then effective address when calculated.**
- **Busy:** **Indicates reservation station is busy.**
- **Register result status:** **Qi Indicates which Reservation Station will write each register, if one exists.**
  - **Blank (or 0) when no pending instruction (i.e. RS) exist that will write to that register.**

Three Stages of Tomasulo Algorithm

Always done in program order

1 **Issue:** **Get instruction from Instruction Queue (IQ).**
- **Instruction issued to a free reservation station (RS) (no structural hazard) in FIFO order.**
- **Selected RS is marked busy.**
- **Control sends available instruction operands values (from ISA registers) to the assigned RS.**

operand를 생성할 reservation stations number로 renamed해준다.

- **Operands not available yet are renamed to RSs that will produce the operand (register renaming). (Dynamic construction of data dependence graph)**

2 **Execution (EX):** **Operate on operands.**
- **When both operands are ready then start executing on assigned FU.**
- **If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)**

내가 필요한 operand을 생성할 reservation station number을 watch하고 있어야 한다.

3 **Write result (WB):** **Finish execution.**
- **Write result on Common Data Bus (CDB) to all awaiting units (RSs)**
- **Mark reservation station as available.**

- **Normal data bus: data + destination ("go to" bus).**
- **Common Data Bus (CDB): data + source ("come from" bus):**
  - **64 bits for data + 4 bits for source (RSs and load buffers).**
  - **Write data to waiting RS if source matches expected RS (that produces result).**
  - **Do the result forwarding via broadcast to waiting RSs.**

Can be done out of program order

Including destination register

Tomasulo Algorithm

- **Control & buffers *distributed* with Functional Units (FUs)**
  - **FU buffers are called *"reservation stations"* which have pending instructions and operands and other instruction status info (including data dependences).**
  - **Reservations stations are sometimes referred to as "physical registers" or "renaming registers" as opposed to architecture or ISA registers specified by the ISA.** (logical register)
- **ISA Registers in instructions are replaced by either values (if available) or pointers (renamed) to reservation stations that will supply the value later:**
  - **This process is called *register renaming*.**
    - **Register renaming eliminates WAR, WAW hazards.**(name dependency이기 때문에 rename되서 해결된다.)
  - **More registers than those ISA supports are possible, leading to optimizations that compilers can't achieve and prevents the number of ISA registers from becoming a bottleneck.**
- **Instruction results go (forwarded) from RSs to RSs , *not through registers*, over *Common Data Bus (CDB)* that broadcasts results to all waiting RSs (dependant instructions.**
- **Loads and Stores are treated as FUs with RSs as well.**

Drawbacks of The Tomasulo Approach

- <u>**Implementation Complexity:**</u>
  - **Example: The implementation of the Tomasulo algorithm may have caused delays in the introduction of 360/91, MIPS 10000, IBM 620 among other CPUs.**
- **Many high-speed associative result stores (using CDB) are required.**
- **Performance limited by one <u>Common Data Bus</u>**
  - **Possible solution:**

    <u>***Multiple CDBs***</u> $\rightarrow$ **more Functional Units and RSs logic (ex. comparators) needed for parallel associative stores.**