Quiz 2 Handout

Figure 1 shows the pipeline of an out-of-order machine. Flip flops represent stage boundaries. Blocks in parallel to each other represent parallel operations occurring within the same stage.

The processor consists of the following stages:

1. Instruction Fetch: The instruction at PC is fetched from the instruction cache/memory.
    - The PC is also fed into a branch target buffer (BTB), which stores mappings from source PC to target PC. On a hit in the BTB, the next PC to be fetched is updated as the target PC indicated in the BTB.
2. Instruction Decode: The instruction is decoded.
    - If the decoded instruction was a conditional branch, its direction is predicted by a branch predictor. The branch predictor is described in the next page.
      Note: Direct jumps (J/JAL) are always taken, so no prediction is needed.
    - For direct branches (BEQZ/BNE/J/JAL), the branch target is calculated by a branch target calculator and updates the next PC to be fetched according to the prediction, if required.
3. Pre-Dispatch Check:
    - The reorder buffer (ROB) is checked for available slots.
    - The free list is checked for free rename registers.
    - For store instructions, the store buffer is checked for available slots.
    - For load instructions, the load buffer is checked for available slots.
4. Dispatch: The instruction is inserted into the ROB only if *all* the checks in the previous cycle (Pre-Dispatch Check) pass.
   This design uses a Unified Register File. The ROB only stores tags to the register names, and does not store data.
5. Execute: The ROB issues an instruction that is ready for execute. The register file is read to obtain any required operands. ALU operations are sent to the appropriate functional units. Cache/memory accesses also take place in this stage.
6. Register Write: The output from the functional units, or memory access are written to the register file, and the ROB is notified.
7. Commit: Instructions are committed **in-order** and the architectural register states are updated.

Note that not all sources, and not all control logic for next PC are shown in Figure 1 for simplicity.

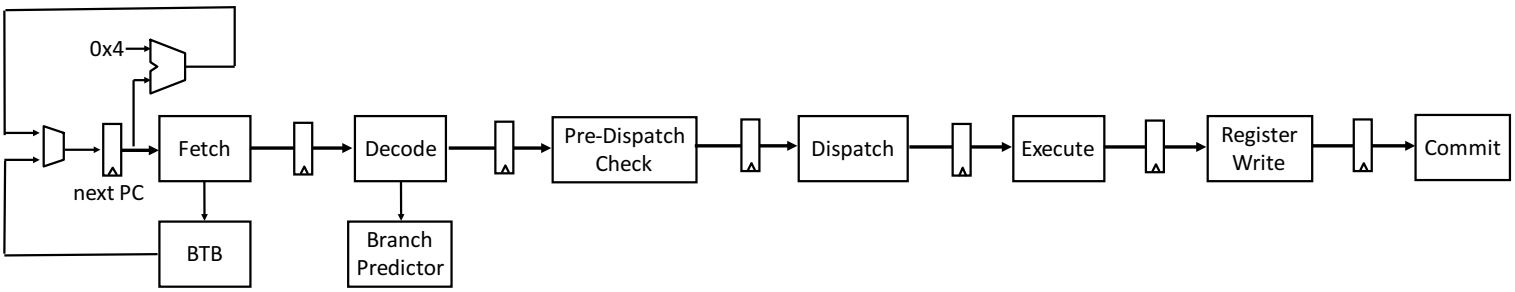

Figure 1: Out-of-order pipeline

***gshare* Branch Predictor:**

The Branch Predictor used in this processor is called ***gshare***, which uses **exclusive OR (XOR)** to combine the global history and the PC. The gshare branch predictor takes the lower three bits from the global history and the lower three bits from the PC (excluding the last 2 bits which are always 00 for aligned instructions), and calculates an index into an array of the two-bit counters by exclusive OR-ing them (Figure 2).

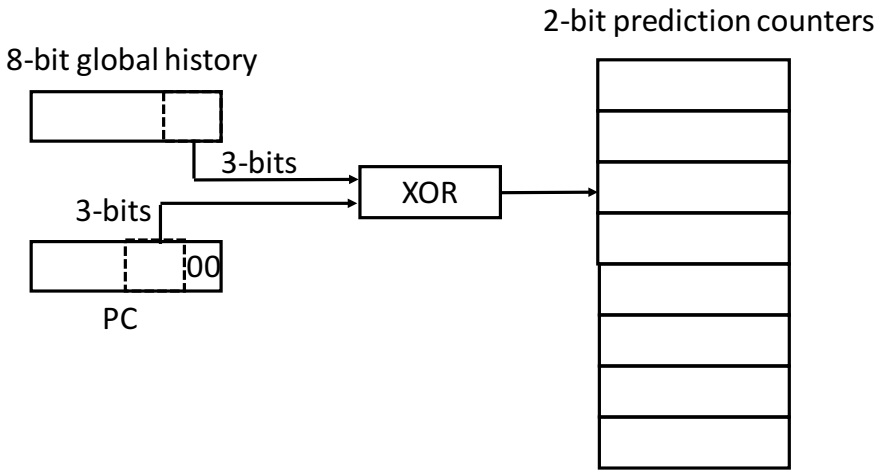

Figure 2: gshare branch predictor

In the global history, 1 represents **Taken** and 0 represents **Not-Taken**. The 2-bit counters in this design follow the state-diagram shown in Figure 3. In state **1X**, we will guess **Taken**; in state **0X**, we will guess **Not-Taken**.

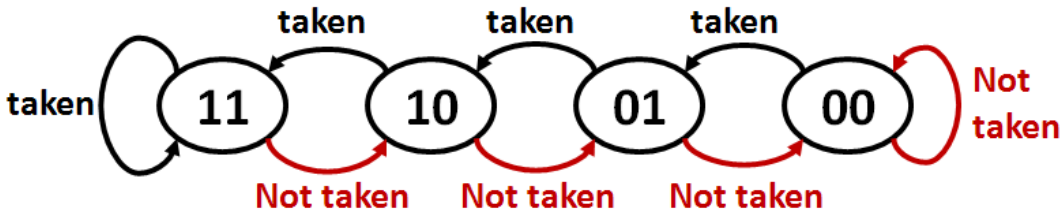

Figure 3: State Diagram of 2-bit counters

Processor State

Prediction Counter	
Index	Value
000	00
001	01
010	10
011	11
100	00
101	11
110	10
111	01

Global History
00010110

Branch Target Buffer		
Entry	PC	Target
1	0xcc	0x2c
2		
3		
4		

Fetched Inst. Queue
I18: 0xcc

Decoded Inst. Queue
I19: 0xc8

Next PC to Fetch
I20:

Store Buffer					
Entry	Valid	Speculative	Inum	Addr	Data
1	1	0	I3	8004	73
2	1	1	I7	8100	0401
3	1	1	I10	8198	8000
4			I14		
5					

Load Buffer			
Entry	Valid	Inum	Addr.
1	1	I1	1008
2	1	I5	1004
3	1	I12	8004
4	1	I16	1000
5			

Free List	P8								

Physical Registers		
Reg	Value	Valid
P1	1331	
P2	8000	1
P3	1008	1
P4	1000	1
P5		
P6		
P7	1729	1
P8		
P9	73	1
P10		
P11		
P12	0401	1

Reorder Buffer											
Inum	PC	Use	Ex	Op	p1	PR1	p2	PR2	Rd	LPRd	PRd
…	…	…	…	…	…	…	…	…	…	…	…
I7	…	…	…	…	…	…	…	…	…	…	…
I8	0x10	1	1	addi	1	P3			R2	P12	P2
I9	0x14	1	1	subi	1	P1			R4	P4	P5
I10	0x18	1	1	sw	1	P2	1	P2			
I11	0x1c	1		beqz		P5					
I12	0xb0	1		lw	1	P2			R1	P1	P6
I13	0xb4	1		mul	1	P2		P6	R5	P9	P10
I14	0xb8	1		sw		P6	1	P2			
I15	0xbc	1		beqz		P6					
I16	0xc0	1	1	lw	1	P3			R1	P6	P11
I17	0xc4	1	1	mul	1	P5	1	P2	R5	P10	P7

Next to commit

Next available

Rename Table	
Register	Value
R1	P11
R2	P2
R3	P3
R4	P5
R5	P7
R6	

Figure 4: Processor State

A snapshot of the processor state is shown in Figure 4. It consists of the following components:

- **Fetched Instruction Queue**: Holds the fetched instructions.
- **Decoded Instruction Queue**: Holds the decoded instructions.
- **Next PC to be fetched**: See Figure 1.
- **Branch Target Buffer (BTB):** Holds map of source PC to target PC. If a fetched instruction PC hits in the BTB, the next PC to fetch is the corresponding target PC.
- **Prediction Counter**: 2-bit counters for branch prediction.
- **Branch Global History**: 8-bit global branch history.
- **Physical Registers**: The processor holds all data values in a **unified physical register file.**
- **Free List**: Tracks which physical registers are available for use.

- **Rename Table:** A map from architectural to physical register names.
- **Reorder Buffer (ROB)**: Contains the bookkeeping information for managing the out-of-order execution and register renaming (but, it does not contain any register data values).
- **Store Buffer**: The address and data from an executed SW instruction are temporarily kept in a store buffer, and then moved to the cache **after** the instruction commits or cleared if the instruction is aborted.
- **Load Buffer**: The address from an executed LW instruction is temporarily kept in the load buffer, and cleared **after** the instruction commits, or is aborted.

For SW instructions, assume the first operand (PR1) provides the base register for the store address, and the second operand (PR2) provides the data source for the store.

We provide a list of actions below. Study them carefully and relate them to the concepts covered in the lectures. You will be required to associate events in the processor to one of these actions.

A. Satisfy a dependence by stalling
B. Satisfy a dependence by bypassing a speculative value
C. Satisfy a dependence by bypassing a committed value
D. Satisfy a dependence by speculation using a static prediction
E. Satisfy a dependence by speculation using a dynamic prediction
F. Write a speculative value using lazy data management
G. Write a speculative value using greedy data management
H. Speculatively update a prediction using lazy value management
I. Speculatively update a prediction using greedy value management
J. Non-speculatively update a prediction
K. Check the correctness of a speculation and find a correct speculation
L. Check the correctness of a speculation and find an incorrect speculation
M. Abort speculative action and cleanup lazily managed values
N. Abort speculative action and cleanup greedily managed values
O. Commit correctly speculated instruction, where there was no value management
P. Commit correctly speculated instruction, and mark lazily updated values as non-speculative
Q. Commit correctly speculated instruction, and free log associated with greedily updated values
R. Illegal or broken action