CS4100: Computer System Design

Exploiting ILP: Superscalar Processors

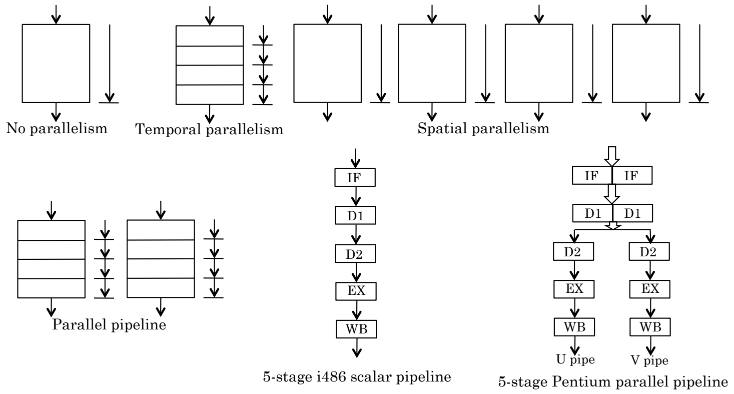
Madhu Mutyam
PACE Laboratory
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Sept 25-Oct 1, 2015

Limitations of Scalar Pipelines

- Single *k*-stage instruction pipelines
  - All instructions, regardless of type, traverse through the same set of pipeline stages
  - At most one instruction can be resident in each pipeline stage at any time
  - Instructions advance through the pipeline stages in a lockstep fashion
- Fundamental limitations
  - The maximum throughput for a scalar pipeline is bounded by one instruction per cycle
  - The unification of all instruction types into one pipeline can yield an inefficient design
  - The stalling of a lockstep or rigid scalar pipeline induces unnecessary pipeline bubbles

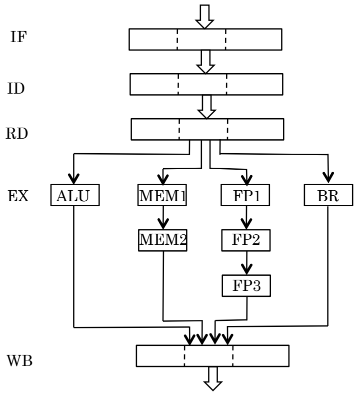
From Scalar to Superscalar Pipelines



- Parallel Pipelines
  - Speedup of a scalar pipeline is determined by the *depth* of the pipeline
  - Speedup of a parallel pipeline is determined by the *width* of the pipeline

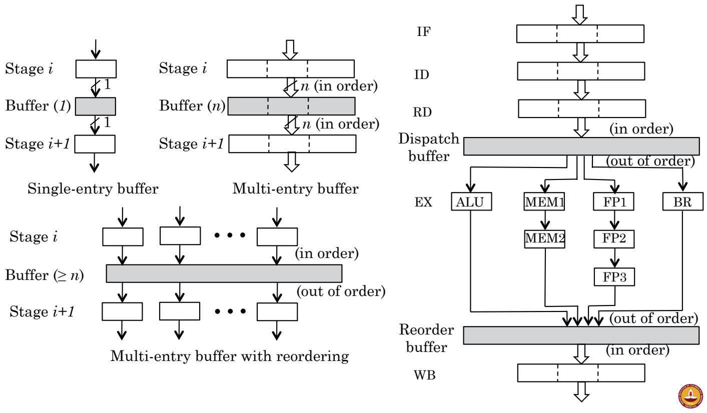
From Scalar To Superscalar Pipelines (Contd)

- Diversified pipelines



From Scalar to Superscalar Pipelines (Contd)

- Dynamic pipelines

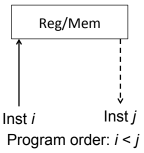


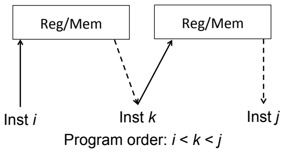
Instruction Level Parallelism (ILP)

- Pipeline CPI = Ideal pipeline CPI + stalls due to pipeline hazards
- Exploit ILP to minimize pipeline stalls
- Identify independent instructions and schedule them appropriately
- Compiler optimizations for exposing ILP
- Dynamic scheduling

Data Dependences

- Instruction *j* is data dependent on instruction *i* if either

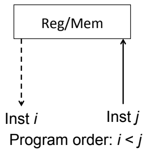


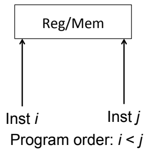


- A data dependence conveys:
  - the possibility of a hazard
  - the order in which results must be calculated
  - an upper bound on the # of instructions that can be executed parallely
- Dependences that flow through memory locations are difficult to detect

Name Dependences

- Occur when two instructions use the same register or memory location, but no flow of information
  - Anti-dependence
  - Output dependence

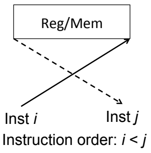


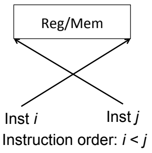


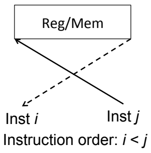
- *Register renaming* can be used to resolve the name dependences

Data Hazards

- A hazard exists whenever there is a *dependence* between instructions, which are at a close distance
- Consider instructions *i* and *j* such that *i* proceeds *j* in program order:
  - RAW hazard
  - WAW hazard
  - WAR hazard







  - RAW corresponds to true dependence
  - WAW corresponds to output dependence
  - WAR corresponds to anti-dependence
- Goal of compiler or hardware techniques is to exploit parallelism by preserving program order only where it affects the program outcome

Control Dependences

- Ordering of an instruction w.r.t. a branch instruction
  - An instruction that is *control dependent* on a branch cannot be moved *before* the branch
  - An instruction that is *not control dependent* on a branch cannot be moved *after* the branch
- As long as program correctness is maintained, control dependence can be violated
  - Preserve *exception behaviour* and *data flow* for program correctness

```
      DADDU   R2, R3, R4
      BEQZ    R2, L1
      LW      R1, 0(R2)
L1:
```

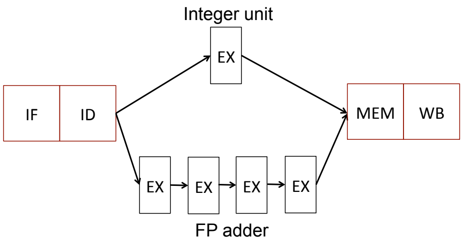
```
      DADDU   R1, R2, R3
      BEQZ    R4, L1
      DSUBU   R1, R5, R6
L1:   . . .
      OR      R7, R1, R8
```

```
      DADDU   R1, R2, R3
      BEQZ    R10, L1
      DSUBU   R4, R5, R6
      DADDU   R5, R4, R9
L1:   OR      R7, R8, R9
```

Compiler Optimizations for Exposing ILP

- Pipeline scheduling
  - Separate the execution of a dependent instruction from the source instruction by the pipeline latency of the source instruction
- Loop unrolling
  - Replicate the loop body multiple times and adjust the loop terminating condition
  - Code size increases

Multi-Cycle Functional Units



Instruction pipeline

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Example #1

```
for (i=999; i ≥ 0; i − −)
    x[i] = x[i] + s;
```

```
Loop:  L.D       F0, 0(R1)
       stall
       ADD.D     F4, F0, F2
       stall
       stall
       S.D       F4, 0(R1)
       DADDUI    R1, R1, #-8
       stall
       BNE       R1, R2, Loop
```

without pipeline scheduling

```
Loop:  L.D       F0, 0(R1)
       ADD.D     F4, F0, F2
       S.D       F4, 0(R1)
       DADDUI    R1, R1, #-8
       BNE       R1, R2, Loop
```

```
Loop:  L.D       F0, 0(R1)
       DADDUI    R1, R1, #-8
       ADD.D     F4, F0, F2
       stall
       stall
       S.D       F4, 8(R1)
       BNE       R1, R2, Loop
```

with pipeline scheduling

Example #1 (Contd)

```
Loop:  L.D       F0, 0(R1)
       ADD.D     F4, F0, F2
       S.D       F4, 0(R1)
       L.D       F6, -8(R1)
       ADD.D     F8, F6, F2
       S.D       F8, -8(R1)
       L.D       F10, -16(R1)
       ADD.D     F12, F10, F2
       S.D       F12, -16(R1)
       L.D       F14, -24(R1)
       ADD.D     F16, F14, F2
       S.D       F16, -24(R1)
       DADDUI    R1, R1, #-32
       BNE       R1, R2, Loop
```

with loop unrolling
(13 stalls)

```
Loop:  L.D       F0, 0(R1)
       L.D       F6, -8(R1)
       L.D       F10, -16(R1)
       L.D       F14, -24(R1)
       ADD.D     F4, F0, F2
       ADD.D     F8, F6, F2
       ADD.D     F12, F10, F2
       ADD.D     F16, F14, F2
       S.D       F4, 0(R1)
       S.D       F8, -8(R1)
       DADDUI    R1, R1, #-32
       S.D       F12, 16(R1)
       S.D       F16, 8(R1)
       BNE       R1, R2, Loop
```

with loop unrolling + pipeline scheduling
(0 stalls)

Conditional Branches Limit Performance

- Make a branch prediction and speculatively execute the instructions in the predicted path
- For each misprediction, recover the state of the processor to the point before the mispredicted branch
- Branch misprediction penalty increases as the pipelines deepen and the number of outstanding instructions increases

Branch Prediction Mechanisms

- Predict the branch direction – whether a conditional branch is taken or not taken
- Predict the branch target
- Static branch prediction techniques
  - always-not-taken, always-taken
- Dynamic branch prediction techniques
  - 2-bit dynamic branch prediction, correlating branch prediction, ...

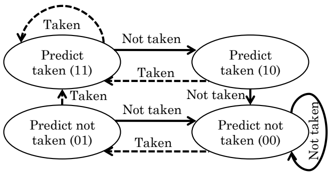
Motivation for Correlating Branch Prediction

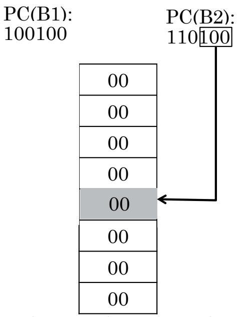
```
    a = 0;
    while (a<1000){
B1          if(a%2 == 0) { ... }
            a++;
            ...
B2          if(b == 0){ ... }
    }
```

Branch history

B1: TNTNTNTNTN....

B2: NNNNNNNNN....



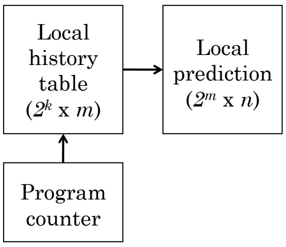


2-bit dynamic branch predictor

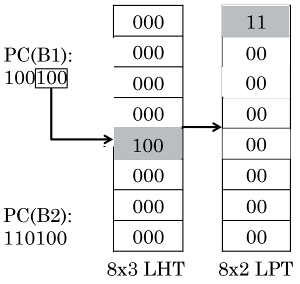
Branch prediction:
N N N N N N N N

Correlating Branch Prediction

- Prediction accuracy may be improved if the recent behavior of other branches is taken into consideration
- (m, n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch


Local history table (2^k x m)
Local prediction (2^m x n)
Program counter

Illustrating (3, 2)-Correlating Branch Prediction



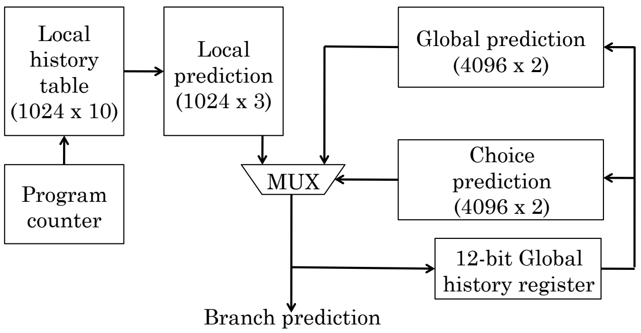
8x3 LHT	8x2 LPT
000	11
000	00
000	00
000	00
100	00
000	00
000	00
000	00

Branch Prediction: N N N N N N N N T

Branch Outcome: T N N N T N N N T

Tournament Branch Predictor

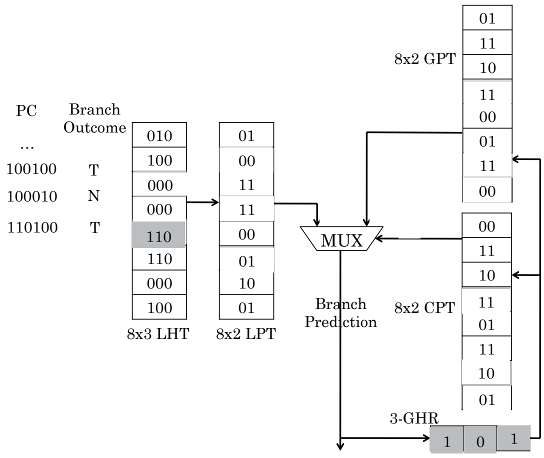
- Adaptively combines local and global branch predictor behavior
- Tournament predictor used in Alpha 21264:


Local history table (1024 x 10)
Program counter
Local prediction (1024 x 3)
Global prediction (4096 x 2)
Choice prediction (4096 x 2)
MUX
12-bit Global history register
Branch prediction

Working of the Tournament Branch Predictor

- Local prediction table: a branch is predicted to be *taken* if the value of a 3-bit saturating counter is ≥ 4
- Global prediction table: a branch is predicted to be *taken* if the value of a 2-bit saturating counter is ≥ 2
- Prediction is made if both the local and global predictions are same
- If the local and global predictions do not match, consult the choice prediction table
- Choice prediction table chooses
  - The global prediction's decision if the value of a 2-bit saturating counter is ≥ 2
  - The local prediction's decision if the value of a 2-bit saturating counter is < 2

Illustrating the Tournament Branch Prediction

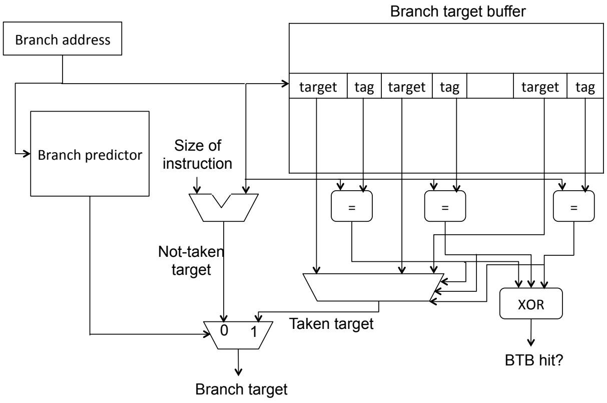


PC	Branch Outcome
...	
100100	T
100010	N
110100	T

8x3 LHT	8x2 LPT	8x2 GPT	8x2 CPT
010	01	01	00
100	00	11	11
000	11	10	10
000	11	11	11
110	00	00	01
110	01	01	11
000	10	11	10
100	01	00	01

Branch Target Buffers

- Cache-like structure to store the last seen target address for branches
- Accessed in parallel with branch prediction algorithm



Thank You