Lecture: Static ILP

- Topics: predication, speculation (Sections C.5, 3.2)
Loop:  
L.D  F0, 0(R1)  
ADD.D  F4, F0, F2  
S.D  F4, 0(R1)  
DADDUI  R1, R1,# -8  
BNE  R1, R2, Loop  

Software Pipeline?!
Software Pipeline

Original iter 1
Original iter 2
Original iter 3
Original iter 4
New iter 1
New iter 2
New iter 3
New iter 4
Software Pipelining

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers
Problem 4

for (i=1000; i>0; i--)
    x[i] = y[i] * s;

Source code

Loop:    L.D         F0, 0(R1)          ; F0 = array element
         MUL.D    F4, F0, F2        ; multiply scalar
         S.D         F4, 0(R2)          ; store result
         DADDUI  R1, R1,# -8      ; decrement address pointer
         DADDUI  R2, R2,#-8       ; decrement address pointer
         BNE        R1, R3, Loop    ; branch if R1 != R3
         NOP

Assembly code

- Show the SW pipelined version of the code and does it cause stalls?
Problem 4

for (i=1000; i>0; i--)
    x[i] = y[i] * s;

Loop:
    L.D F0, 0(R1) ; F0 = array element
    MUL.D F4, F0, F2 ; multiply scalar
    S.D F4, 0(R2) ; store result
    DADDUI R1, R1, # -8 ; decrement address pointer
    DADDUI R2, R2, # -8 ; decrement address pointer
    BNE R1, R3, Loop ; branch if R1 != R3
    NOP

• Show the SW pipelined version of the code and does it cause stalls?

Loop:
    S.D F4, 0(R2)
    MUL F4, F0, F2
    L.D F0, 0(R1)
    DADDUI R2, R2, # -8
    DADDUI R1, R1, # -8
    BNE R1, R3, Loop ; branch if R1 != R3
    There will be no stalls
Predication

• A branch within a loop can be problematic to schedule

• Control dependences are a problem because of the need to re-fetch on a mispredict

• For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions
Predicated or Conditional Instructions

if (R1 == 0)
    R2 = R2 + R4
else
    R6 = R3 + R5
    R4 = R2 + R3

R7 = !R1
R8 = R2
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
Predicated or Conditional Instructions

• The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

• Example: lwc  R1, 0(R2), R3  (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

• Replaces a control dependence with a data dependence (branches disappear) ; may need register copies for the condition or for values used by both directions

```plaintext
if (R1 == 0)
    R2 = R2 + R4
else
    R6 = R3 + R5
R4 = R2 + R3

R7 = !R1 ;  R8 = R2 ;
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
```
Problem 1

• Use predication to remove control hazards in this code

```c
if (R1 == 0)
    R2 = R5 + R4
    R3 = R2 + R4
else
    R6 = R3 + R2
```
Problem 1

• Use predication to remove control hazards in this code

```
if (R1 == 0)
    R2 = R5 + R4
    R3 = R2 + R4
else
    R6 = R3 + R2
R7 = !R1 ;
R6 = R3 + R2   (predicated on R1)
R2 = R5 + R4   (predicated on R7)
R3 = R2 + R4   (predicated on R7)
```
Complications

• Each instruction has one more input operand – more register ports/bypassing

• If the branch condition is not known, the instruction stalls (remember, these are in-order processors)

• Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work

• Increases register pressure, activity on functional units

• Does not help if the br-condition takes a while to evaluate
Support for Speculation

• In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences

• However, we need hardware support
  ➢ to ensure that an exception is raised at the correct point
  ➢ to ensure that we do not violate memory dependences
Detecting Exceptions

• Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)

• For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss

• In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative

• Note that a speculative instruction needs a special opcode to indicate that it is speculative
Program-Terminate Exceptions

• When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register

• If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the segfault happens two procedures later)

• Alternatively, an instruction (the sentinel) in the speculative instruction’s original location checks the register value and initiates recovery
Memory Dependence Detection

• If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

• When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

• If a store finds its address in the ALAT, it indicates that a violation occurred for that address

• A special instruction (the sentinel) in the load’s original location checks to see if the address had a violation and re-executes the load if necessary
Problem 2

• For the example code snippet below, show the code after the load is hoisted:

Instr-A
Instr-B
ST R2 → [R3]
Instr-C
BEZ R7, foo
Instr-D
LD R8 ← [R4]
Instr-E
Problem 2

• For the example code snippet below, show the code after the load is hoisted:

  Instr-A
  Instr-B
  ST  R2 → [R3]
  Instr-C
  BEZ R7, foo
  Instr-D
  LD R8 ← [R4]
  Instr-E

  LD.S R8 ← [R4]
  Instr-A
  Instr-B
  ST R2 → [R3]
  Instr-C
  BEZ R7, foo
  Instr-D
  LD.R8 ← [R4]
  Instr-E

  rec-code: LD R8 ← [R4]
Amdahl’s Law

• Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power

• Amdahl’s Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play

• Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl’s Law states that maximum execution time reduction is 40% (max speedup of 1.66)
Principle of Locality

- Most programs are predictable in terms of instructions executed and data accessed

- The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code

- Temporal locality: a program will shortly re-visit $X$

- Spatial locality: a program will shortly visit $X+1$
Problem 1

• What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?
Problem 1

- What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?

The index is 12 bits wide, so the table has $2^{12}$ saturating counters. Each counter is 3 bits wide. So total storage = $3 \times 4096 = 12$ Kb or 1.5 KB
Problem 2

• What is the storage requirement for a tournament predictor that uses the following structures:
  ▪ a “selector” that has 4K entries and 2-bit counters
  ▪ a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  ▪ a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.
Problem 2

• What is the storage requirement for a tournament predictor that uses the following structures:
  - a “selector” that has 4K entries and 2-bit counters
  - a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  - a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.

Selector = 4K * 2b = 8 Kb
Global = 3b * 2^14 = 48 Kb
Local = (12b * 2^8) + (2b * 2^12) = 3 Kb + 8 Kb = 11 Kb
Total = 67 Kb
Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

do {
    for (i=0; i<4; i++) {
        increment something
    }
    for (j=0; j<8; j++) {
        increment something
    }
    k++;
} while (k < some large number)
Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

```c
do {
    for (i=0; i<4; i++) {
        increment something
    }
    for (j=0; j<8; j++) {
        increment something
    }
    k++;
} while (k < some large number)
```

PC+4: \( \frac{2}{13} = 15\% \)

1b Bim: \( \frac{2+6+1}{4+8+1} = \frac{9}{13} = 69\% \)

2b Bim: \( \frac{3+7+1}{13} = 11/13 = 85\% \)

Global: \( \frac{4+7+1}{13} = 12/13 = 92\% \)

(gets confused by 01111 unless you take branch-PC into account while indexing)

Local: \( \frac{4+7+1}{13} = 12/13 = 92\% \)
Title

• Bullet