Lecture #13

****
Review -- 1 min
****
Branch prediction
Branch Target Buffer
 1-cycle or 0-cycle branches
 combine with branch pred. algorithms
Multiple issue: superscalar v. VLIW

****
Outline - 1 min
****
Multiple Issue
  pressure ILP -- pipeline depth * issue rate
  7-deep times 5 issue = 35 parallel instructions needed
   (v. branch per 5 instructions)
Improving ILP
 Compiler -- trace scheduling
  ~loop unrolling
  (harder b/c harder taken/not taken decision
  and less regular code)
 not talk about -- SW pipelining

HW for Improved ILP
 Annulling instructions
 Speculative Execution
Limits to ILP

****
Lecture - 20 min
****
Superscaler

Fetch and Issue Multiple Instructions per Cycle

 IF ID EX MEM WB
 IF ID EX MEM WB

IF	ID	EX	MEM	WB	
IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
		IF			
		...			

Typicaly restrict issue --
instead of 5 arbitrary instructions in parallel:
e.g. 2 integer
2 FP
1 branch

Observations
1. ILP -- #issue * pipeline depth
e.g. in above instruction need 20 instructions
that can execute at once
2. Superscaler == Superpipelined

<viewgraph -- get picture from limits of ILP paper>

IF	IF	IF	IF	ID	ID	ID	...
	IF	IF	IF	IF	ID	ID	...
		IF	IF	IF	IF		

-- Still need 20 instructions executing at once

3. Complex issue logic
Examine n opcodes, 3*n registers
hazard cases proportional to n^2

pipelining -- limited by clock skew, latch delay
superscaler limited by issue logic (which may slow
clock)
both limited by ILP

4. Increased Branch Hazard

May slow issue (2 or more cycles)
--> more cycles lost
Multiple insructions per clock
--> each clock cycle lost costs multiple instructions
(notice -- same problem for superpipelined)

5. increase load delay
same reasons

6. Use lots of hardware
memory BW for instructions
register ports
memory ports
hazard detection

Notice: both VLIW and Superscalar/superpipelined pressure ILP

Increasing ILP in SW and HW

SW: Trace Scheduling
HW: Annulling instructions
HW: Speculative Execution

Trace Scheduling

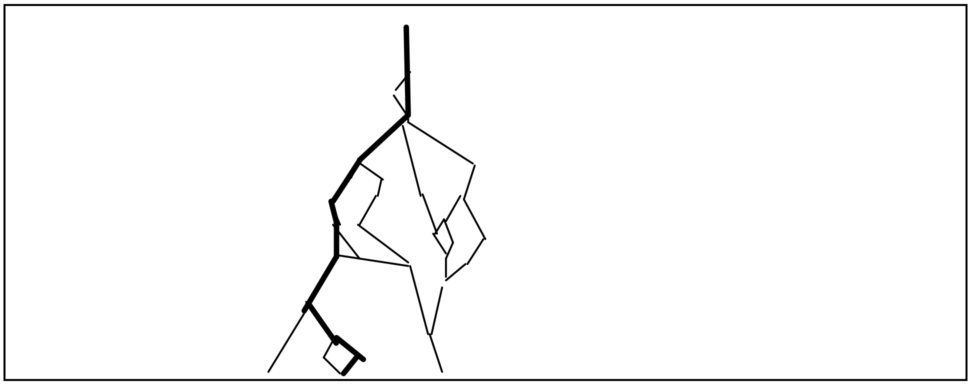
Loop unrolling says -- common case is loop is taken
so copy code from after conditional branch
to before it

Can do same thing for non-loop conditional branches

called *Trace Scheduling*

picture: tree/graph of execution paths

Trace sceduling -- optimize for most likely path



Like loop unrolling, 2 steps

1. trace selection (~unrolling)

--> create straight line code down likely path

example

```
            if A==0
                    // A1
                    <insr>; <insr>;...
            else
                    //A2
-->likely           <instr>; <instr>; ...
                    if B==0
                            // B1
-->likely                   <instr>...
                    else
                            //B2
                            <instr>...
            //C
            <instr>...


            BEQ R1, A1
```

```
A2:     <instr>
        <instr>
        ...
        BNEZ R2, B2
B1:     <instr>
        <instr>
        ...
C:      <instr>
        ...


A1:     <instr>
        ...
        j C

B2:     <instr>
        ...
        j C
```

example: if untaken branch faster than taken branch
make the straight line

2. trace compaction (~scheduling)
-->avoid hazards by moving instructions as early as
possible

Trace scheduling harder, less common than loop unrolling
o harder to tell by inspection if branch taken/not taken
(need high prob. branch to make it worth while)
o less regular code -- harder to move around

Annulling Instructions

Tech trends:
branches getting more expensive v. regular instructions
-- superscaler/pipelined --> more cycles delay
-- superscaler --> more instructions per cycle lost

Idea -- change control dependence into a data dependence

Conditional instruction evaluates condition (e.g. register === 0)
    if true -- do instruction
    if false -- no effect (no register update; no exception)

example: conditional move

```
	BNEZ R1, L
	MOV R2, R3
    L: ....
```

becomes

```
	CMOVZ R2, R3, R1
```

Most vendors have added to new instruction
    CMOV -- Alpha, MIPS, PowerPC, SPARC
    annul any following instr -- PA-RISC
(-->probably a good idea)

Speculative Execution (HW Support)

Ties together
- o Dynamic branch prediction
- o register renaming
- o in-order commit

Advantages of HW v. Software
1. Disambiguate memory references
2. better branch prediction
3. binary compatibility

Implementation:
1. Branch Target Buffer

--> don't stall on instructions
(even if condition depends on pending instructions)

2. renaming + in-order commit via Reorder Buffer
   --> don't commit until branch direction known

Key point: look at reorder buffer

	State	Value
1. ADDD ...		
2. SUB ...		
3. BNEZ R1, LOOP	T	NT
4. MULTD ...		
5. ADD ...		

QUESTION: need to restore PC after mispredicted branch (or exception)
How?

1. Add PC field to reorder buffer (DA: doubles size)
2. keep reorderPC -- increment on every committed instruction; update on taken branch

****
Admin - 3 min
****

****
Lecture - 24 min
****

Limits to ILP

Motivation: in first lecture -- electrical engineers --> 35% performance per yer via better processes
Past few years -- 55% per year -- the 20% comes from taking advantage of better ILP

pipelining (SPARC1, MIPS R2000)
superpipelining (Alpha 21064, Mips R4000)
superscaler (IBM Power, Ultrasparc, Alpha)

...

What will it take to get more ILP?
Where does it end?

ILP Studies

Goal: FInd out how much parallelism
Difficulty: depends on

1. Benchmarks used
2. hardware assumptions
3. compiler assumptions
   (e.g. no conditional execution instruction)

This study

Aggressive HW model

1. Infinite Functional Units (1 cycle latency for all)
2. Infinite 1 cycle cache
3. Inifinite renaming registers
   -- not WAR, WAW hazards
4. Perfect branch and jump prediction
5. Infinite instruction issue
6. perfect alias analysis in memory

Picture: you have trace of entire program
(you know whether all branches are taken/not taken before program starts running)
load the trace into an infinite tomasulo engine
at every cycle, execute all instructions that no longer have data hazards

Notice, this is *really* aggressive, but, surprizingly, we can imagine a machine that does even better – HW shows a technique that executes instructions before data dependency clears.

Also, standard MIPS compiler (could imagine a compiler that does high-level optimizations to get rid of unnecessary calculations, etc.)
As we restrict hardware, compiler limitations may become more important

Start with agressive hardware then make more realistic assumptions.

SLIDES:

- upper limit to ILP (fig 4.38 p 319)
    - ideal processor
- window (fig 4.40 p 323)
    - dependencies (issue)^2

      Think about logic for renaming registers
      This is part of why machines will issue "2FP, 2 integers, and a branch, but not 5 integer instructions – limit dependencies to check)

- branch and jump prediction (fig 4.42)
    - "ideal processor" – results of all branches known before first instruction executed!

      For restricted cases, rule is that if you mispredict you have to stop until dependency resolved (no other penalty, just restricts parallelism)

- register impact (fig 4.44 p328)
    - limit number of renaming registers

      surprizingly much impact from name dependencies – need to look at lots of live variables to get full parallelism
      (today's processors <32)

alias (4.46 p 330)

conclude: need either sophisticated compiler analysis or dynamic HW

realistic (vy aggressive) (4.48 p 332)

- <= 64 instructions issued per clock
    - of any type (e.g. all Loads? OK)
- selective predictor with 1K entries; 16-entry return predictor
- perfect disambiguation of memory refs (HW)
- register renaming w/ 64 integer and 64 FP
- vary window size
- perfect cache; 1-cycle latencies for all instructions (XXX not realistic)
-

Bottom line:

FP limited by how many instructions we can issue

Integer: limited by branch prediction

Case study: powerPC604 and PPro

<slides>

****

Summay - 1 min

****