Static Scheduling

- basic pipeline: single, in-order issue
- first extension: multiple issue (superscalar)
- second extension: scheduling instructions for more ILP
  - option #1: dynamic scheduling
  - option #2: static scheduling

VLIW: Very Long Instruction Word

- problems with superscalar implementation
  - wide fetch+branch prediction (can partially fix w/ trace cache)
  - N^2 bypass (can partially fix with clustering)
  - N^2 dependence cross-check (stall+bypass logic)

one alternative: VLIW (very long instruction word)

- single-issue pipe, but unit is N-instruction group (VLIW)
  - instructions in VLIW are guaranteed to be independent (by compiler)
  + processor does not have to dependence-check within a VLIW
  - VLIW travels down pipe as a unit
  - typically “slotted” (i.e., 1st must be ALU, 2nd must be load, etc.)

Readings

H+P
- chapter 4

Readings in Computer Architecture
- Rau and Fisher: “Instruction-Level Parallel Processing”

Recent Research Papers
- EPIC/IA-64

VLIW History

- started with microcode (“horizontal microcode”)
- academic projects
  - ELI-512 [Fisher, ’85]
  - Illinois IMPACT [Hwu, ’91]
- commercial machines
  - MultiFlow [Colwell+Fisher, ’85] ⇒ failed
  - Cydrome [Rau, ’85] ⇒ failed
  - EPIC (IA-64, Itanium) [Colwell,Fisher+Rau, ’97] ⇒ ??
  - Transmeta [Ditzel, ’99]: translates x86 to VLIW ⇒ ??
  - many embedded controllers (TI, Motorola) are VLIW ⇒ success
Pure VLIW

- **pure VLIW**: no hardware dependence-checks at all
  - not even between VLIW groups
- compiler responsible for scheduling entire pipeline
  - including stall cycles
  - possible if you know structure of pipeline and latencies exactly
- problem 1: pipe & latencies vary across implementations
  - recompile for new implementations (or risk missing a stall)?
  - TransMeta solves this problem by recompiling on-the-fly
- problem 2: latencies are NOT fixed within implementation
  - don’t use caches? (forget it)
  - schedule assuming cache miss? (no point to having caches)

A VLIW Compromise

**compromise**: EPIC (Explicitly Parallel Instruction Computing)
- less rigid than VLIW (not really VLIW at all)
- variable width instruction words
  - implemented as “bundles” with dependence bits
  - makes code compatible with different width machines
- assumes inter-bundle stall logic provided by hardware
  - makes code compatible with different pipeline depths, op latencies
  - enables stalls on cache misses (actually, out-of-order too)
- exploits any information on parallelism compiler can give
- compatible with multiple implementations of same arch
  - e.g., IA64, Itanium

not many VLIW purists left

ILP and Scheduling

no point to having an N-wide pipeline if, on average, many fewer than N independent instructions per cycle
- performance is important
- but utilization (actual/peak performance) is also

Code Example: SAXPY

- **SAXPY** (single-precision A*X+Y)
  - linear algebra routine (used in solving systems of equations)
  - part of famous “Livermore Loops” kernel (early benchmark)
  
  ```
  for (I=0;I<N;I++)
  Z[I] = A*X[I] + Y[I]
  ```
  
  ```
  ldf f0, X(r1) // loop:    
  mulf f4,f0,f2    // assume A in f2
  ldf f6, Y(r1)    // X,Y,Z are constant addresses
  addf f8,f6,f4
  stf f8, Z(r1)
  add r1,r1, #4    // assume I in r1
  ble r1,r2,loop   // assume N*4 in r2
  ```
Default SAXPY Performance

- scalar, pipelined processor (for illustration)
  - 5 cycle FP mult, 2 cycle FP add, both pipelined
  - full bypassing, branches predicted taken
- single iteration (7 instructions) latency: 15 cycles
- performance: 7 instructions / 15 cycles ⇒ IPC = 0.47
- utilization: 0.47 actual IPC / 2 peak IPC ⇒ 24%

Performance and Utilization

- superscalar pipeline
  - same configuration, just two at a time
  - performance: still 15 cycles - not any better (why?)
  - utilization: 0.47 actual IPC / 2 peak IPC ⇒ 24%
  - notice: more hazards → stalls (why?)
  - notice: each stall more expensive (why?)

Scheduling and Issue

- instruction scheduling: decide on instruction execution order
- important tool for improving utilization and performance
- related to instruction issue (when instructions execute)
  - pure VLIW: static scheduling with static issue
  - in-order superscalar, EPIC: static scheduling with dynamic issue
  - well, not completely dynamic...
  - in-order pipeline relies on compiler to schedule well

Instruction Scheduling

- idea: independent instructions between slow ops and uses
  - otherwise pipeline sits idle waiting for RAW to resolve
  - we have already seen dynamic pipeline scheduling
- to do this we need independent instructions
- scheduling scope: code region we are scheduling
  - the bigger the better (more independent instructions to choose from)
  - once scope is defined, schedule is pretty obvious
  - trick is making a large scope (schedule across branches???)
- compiler scheduling techniques (more about these later)
  - loop unrolling (for loops)
  - software pipelining (also for loops)
  - trace scheduling (for general control-flow)
Scheduling: Compiler or Hardware?

- compiler
  + large scheduling scope (full program), large “lookahead”
  + enables simple hardware with fast clock
  - low branch prediction accuracy (profiling?)
  - no information on latencies like cache misses (profiling?)
  - pain to speculate and recover from mis-speculation (h/w support?)

- hardware
  + better branch prediction accuracy
  + dynamic information on latencies (cache misses) and dependences
  + easy to speculate & recover from mis-speculation
  - finite on-chip instruction buffering limits scheduling scope
  - more complicated hardware (more power? tougher to verify?)
  - slow clock

Aside: Profiling

profile: (statistical) information about program tendencies
- run program once with a test input and see how it behaves
- hope that other inputs lead to similar behaviors
- compiler can use this info for scheduling
- profiling can be a useful technique
  - must be used carefully - else, can harm performance
- popular research topic
  - gaining importance

Loop Unrolling SAXPY

we want to separate dependent operations from one another
- but not enough flexibility within single iteration of loop
- longest chain of operations is 9 cycles
  - load result (1 cycle)
  - forward to multiply (5 cycles)
  - forward to add (2 cycles)
  - forward to store (1 cycle)
  - can’t hide 9 cycles of latency using 7 instructions
  - how about 9 cycles of latency twice in 14 instructions?
- loop unrolling: schedule 2 loop iterations together

Unrolling Part 1: Fuse Iterations

• combine two (in general, N) iterations of loop
  • fuse loop control (induction increment + backward branch)
  • adjust implicit uses of internal induction variables ($r1$ in example)

```assembly
ldf f0,X(r1)
mulf f4,f0,f2
ldf f6,Y(r1)
addf f8,f6,f4
stf f8,%r1
add r1,r1,#4
ble r1,r2,loop
```

```assembly
ble r1,r2,loop
```

```assembly
ldf f0,X(r1)
mulf f4,f0,f2
ldf f6,Y(r1)
addf f8,f6,f4
stf f8,%r1
add r1,r1,#4
ble r1,r2,loop
```

```assembly
beq r1,r2,loop
```

```assembly
```
### Unrolling Part 2: Pipeline Schedule

- **pipeline schedule to reduce RAW stalls**
- have seen this already (as done dynamically by hardware)

```plaintext
ld f0, X(r1)
mulf f4, f0, f2
ld f6, Y(r1)
addf f8, f6, f4
st f8, Z(r1)
ld f0, X+4(r1)
mulf f4, f0, f2
ld f6, Y+4(r1)
addf f8, f6, f4
st f8, Z+4(r1)
add r1, r1, #8
ble r1, r2, loop
```

### Unrolling Part 3: Rename Registers

- **pipeline scheduling caused WAR hazards**
- so we rename registers to solve this problem (similar to w/hardware)

```plaintext
ld f0, X(r1)
ld f0, X+4(r1)
mulf f4, f0, f2
ld f6, Y(r1)
mulf f4, f0, f2
ld f6, Y+4(r1)
addf f8, f6, f4
st f8, Z(r1)
addf f8, f6, f4
stf f8, Z+4(r1)
add r1, r1, #8
ble r1, r2, loop
```

### Unrolled SAXPY Performance

- **2 iterations (12 instructions) → 17 cycles (fewer stalls)**
- before unrolling, it took 15 cycles for 1 iteration!

```plaintext
ld f0, A(r1)
ld f10, A+4(r1)
mulf f4, f0, f2
mulf f14, f10, f2
ld f6, B(r1)
ld f16, B+4(r1)
addf f8, f6, f4
addf f18, f16, f14
st f8, C(r1)
st f18, C+4(r1)
add r1, r1, #8
ble r1, r2, loop
```

### Shortcomings of Loop Unrolling

- **code growth**
- poor scheduling along “seams” of unrolled copies
- doesn’t handle inter-iteration dependences (recurrences)

```plaintext
for (I=0; I<N; I++)
X[I] = A*X[I-1]; // each iteration depends on prior
```

```plaintext
ld f2, X-4(r1)
mulf f4, f2, f0
st f4, X(r1)
add r1, r1, #4
ble r1, r2, loop
```

1 dependence chain → can’t schedule
Software Pipelining

**software pipelining:** deals with these problems
- also called symbolic loop unrolling
- reinvented a few times under different guises
  - microcode [Charlesworth '81]
  - polycyclic scheduling [Rau '85]
  - general loop unrolling [Lam '88]

The Pipeline Analogy

- **hardware pipeline**
  - any cycle contains:
    - stage 3 of inst i, stage 2 of inst i+1, stage 1 of inst i+2
- **software pipeline**
  - cycle → software pipelined (physical) iteration
  - instruction → logical (original) iteration
  - stage → instruction

  - single physical iteration contains...
  - ...instructions from multiple original iterations:
    - inst 3 of iteration i, inst 2 of iteration i+1, inst 1 of iteration i+2

Software Pipelining Example

- **physical iteration** (box) contains:
  - $\text{stf}$ from original iteration i
  - $\text{ldf,mulf}$ from original iteration i+1
  - **prologue:** get pipeline started ($\text{ldf,mulf}$ from iteration 0)
  - **epilogue:** finish up leftovers ($\text{stf}$ from iteration N)

```
ldf f2,X-4(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
```

```
ldf f2,X-4(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
```

Software Pipelining Pipeline Diagrams

- **same diagram, new terminology**
  - cycles → physical iterations (across)
  - instructions → logical iterations (down)
  - stages → instructions ($\text{LM = ldf, mufl,S = stf}$)

```
1 2 3 4 5
L M S
L M S
L M S
```

- **NOTICE,** within physical iteration, instruction groups are in reverse order
- that's OK, groups are unrelated (parallel)
- perfect for VLIW!!
Software Pipelining Example II

• vary software pipelining structure to tolerate more latency
  • e.g., ldf, mulf, stf from 3 different iterations

```assembly
ldf f2,X(r1)  ldf f2,X-8(r1)
ldf f4,f2,f0  mulf f4,f2,f0
stf f4,X(r1)  ldf f2,X-4(r1)
add r1,r1,#4  mulf f4,f2,f0
ble r1,r2,loop stf f4,X-8(r1)
add r1,r1,#4  mulf f4,f2,f0
ble r1,r2,loop stf f4,X(r1)
```

```
1 2 3 4 5 6
1 L M S
2 L M S
3 L M S
4 L M S
```

Software Pipelining

+ doesn’t increase code size (much)
+ good scheduling at iteration “seams”
+ can vary degree of pipelining to tolerate longer latencies
  • “software superpipelining”
  • one physical iteration: instructions from logical iterations i,i+2,i+4
    – hard to do conditionals within loops
    – tricky register allocation sometimes

Trace Scheduling

problem: not everything is a loop
idea: for general non-loop situations
• find common paths in program
  • realign basic blocks to form straight-line trace
    • basic block: single-entry, single-exit instruction sequence
    • trace (aka superblock, hyperblock): fused basic block sequence
• schedule instructions within trace
• create fixup code outside trace in case trace != actual path
  – nasty
• trace scheduling
  • [Ellis,’85]

Trace Scheduling Example

```plaintext
A = Y[i];
if (A == 0)
  A = W[i];
else
  Y[i] = 0;
Z[i] = A*X[i];
```

```plaintext
#0: ldf f2,Y(r1)
#1: bnez f2,#4
#2: ldf f2,W(r1)
#3: jump #5
#4: stf f0,Y(r1)
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)
```

• scheduling problem: separate #6 (3 cycles) from #7
  • but how to move `mulf` and `ldf` above if-then-else?
Basic Blocks and Superblocks

- choose most common path: A,C,D
  - assumes you know branch #1’s frequency (profiling)
- fuse into one large “superblock” & schedule
- create repair code in case real path was A,B,D

4 basic blocks: A,B,C,D

![Diagram of basic blocks and superblocks]

Repair Blocks

- change sense (test of branch)
- repair block: may need to duplicate code (block D here)
- haven’t scheduled superblock yet

Superblock Scheduling 1

- first scheduling move: move #5,#6 above #4
  - moved load (#5) above store (#4)
  - we can tell this is OK, but can the compiler?
    - if yes, fine
    - otherwise, need to do something

ISA Support for Load-Store Speculation

- change #5 to advanced load, lf.a
  - “advanced” means advanced past unknown store
- processor tracks load address, matches with other stores
- insert chk.a to check store collision. collision? repair
- called “memory conflict buffer (MCB)”, adopted by IA64
Superblock Scheduling 2

second scheduling move: move #5, (load) #6 above #1 (branch)
• that's OK, load did not depend on branch
• was going to be executed anyway

scheduling non-move: don't move #4 (store) above #1 (branch)
• why? hard (but possible) to undo a store in repair block

Superblock Scheduling 3

move #2 (load) #5, and #6 above #1 (branch)
• rename f2 to f8 to avoid name conflicts
• is this an OK thing to do?
• from a store standpoint, yes
• what about from a fault standpoint? what if #2 faults?

ISA Support for Load-Branch Speculation

• change #2 to speculative load, l.f.s
  • “speculative” means speculative above unknown branch
• processor keeps interrupt bit with register f8
• interrupt handled when f8 is used by #6
• called “poison bit” or “deferred interrupt”, adopted by IA64
Hyperblock Scheduling

what if branch #1 is not biased?
• create a large block from both paths (all 4 basic blocks)
• called a **hyperblock**
• use **predication** to conditionally execute instruction

ISA Support for Predication

`#0: ldf f2, Y(r1)`
`#1: sltip p1, f2, #0`
`#2: ldf p f2, W(r1), p1`
`#4: stf np f0, Y(r1), p1`
`#5: ldf f4, X(r1)`
`#6: mulf f6, f4, f2`
`#7: stf f6, Z(r1)`

two levels of predication
• **full predication**: can tag every instruction with predicate
  • adopted by IA64
• **conditional register moves**: (CMOVE)
  • construct appearance of full predication from one basic primitive
  
  ```
  cmoveqz r1, r2, r3  // if (r3 == 0) r1 = r2;
  ```
  • may require a lot of code duplication
  • adopted by Alpha, IA32
• “if-conversion”: converts control-flow to data-flow
  + eliminates branches
  – why can it be bad?

Static Scheduling Summary

• loop unrolling
  + reduces branch frequency
  – expands code size, have to handle “extra” iterations
• software pipelining
  + no dependences in loop body
  – does not reduce branch frequency, need prologue/epilogue blocks
• trace scheduling
  + works for non-loops
  – more complex than unrolling and software pipelining
• ISA support
  • speculative loads, advanced loads
  • predication
Multiple Issue Summary

- multiple issue problem spots
  - fetch, branch prediction ⇒ trace cache?
  - decode (N^2 dependence cross-check)
  - execute (N^2 bypass) ⇒ clustering?

- implementations
  - in-order superscalar
  - VLIW
  - EPIC (compromise)

material for mid-term ends here

next up: the memory system (caches, memory, etc.)