Computer Science 146
Computer Architecture

Fall 2019
Harvard University

Instructor: Prof. David Brooks
dbrooks@eecs.harvard.edu

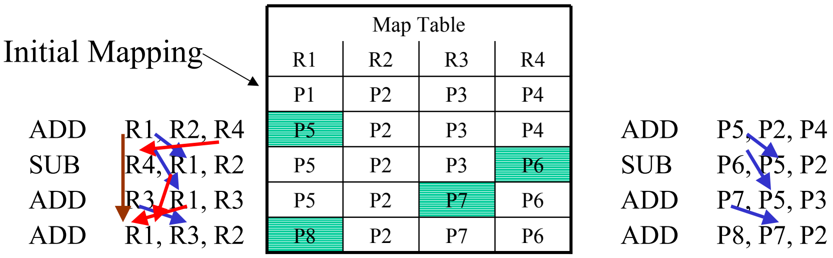
Lecture 10: Static Scheduling, Loop Unrolling, and Software Pipelining

---

Lecture Outline

- Finish Pentium Pro and Pentium 4 Case Studies
- Loop Unrolling and Static Scheduling
  - Section 4.1
- Software Pipelining
  - Section 4.4 (pages 329-332)

MIPS R10K: Register Map Table



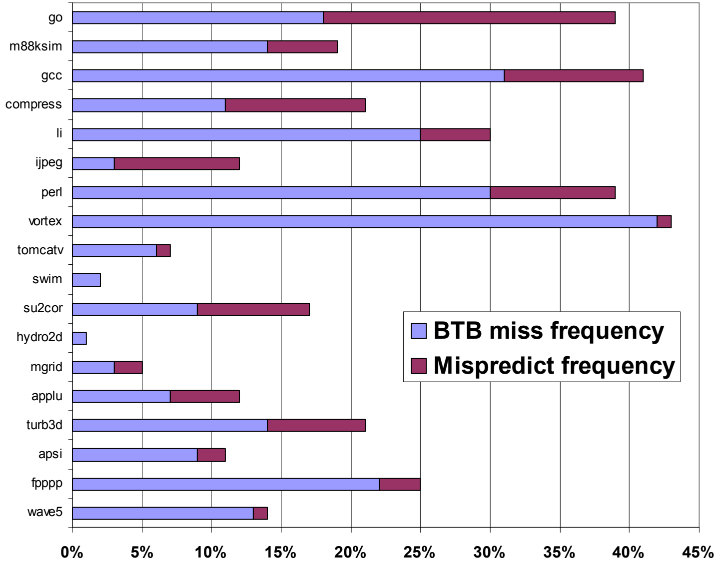
Map Table

		R1	R2	R3	R4		
Initial Mapping		P1	P2	P3	P4		
ADD	R1, R2, R4	P5	P2	P3	P4	ADD	P5, P2, P4
SUB	R4, R1, R2	P5	P2	P3	P6	SUB	P6, P5, P2
ADD	R3, R1, R3	P5	P2	P7	P6	ADD	P7, P5, P3
ADD	R1, R3, R2	P8	P2	P7	P6	ADD	P8, P7, P2

MIPS R10K:
How to free registers?

- Old Method (Tomasulo + Reorder Buffer)
  - Don’t free speculative storage explicitly
  - At Retire:
    - Copy value from ROB to register file, free ROB entry
- MIPS R10K
  - Can’t free physical register when instructions retire
    - There is no architectural register to copy to
  - Free physical register previously mapped to same logical register
  - All instructions that will read it have retired already

P6 Performance: Branch Mispredict Rate



10% to 40% Miss/Mispredict ratio: 20% avg. (29% integer)

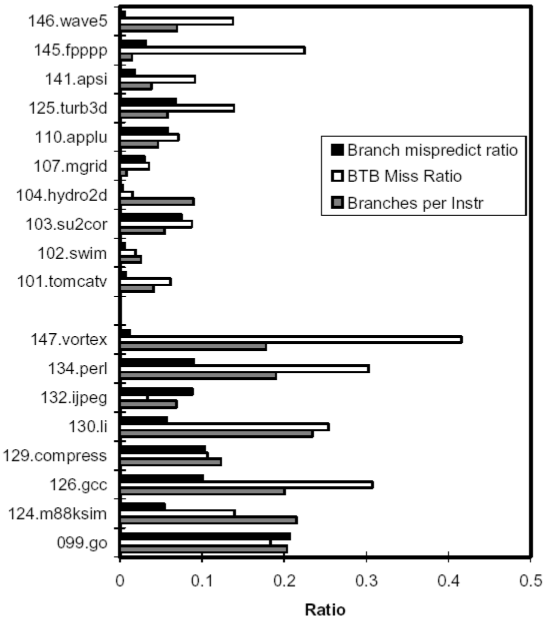
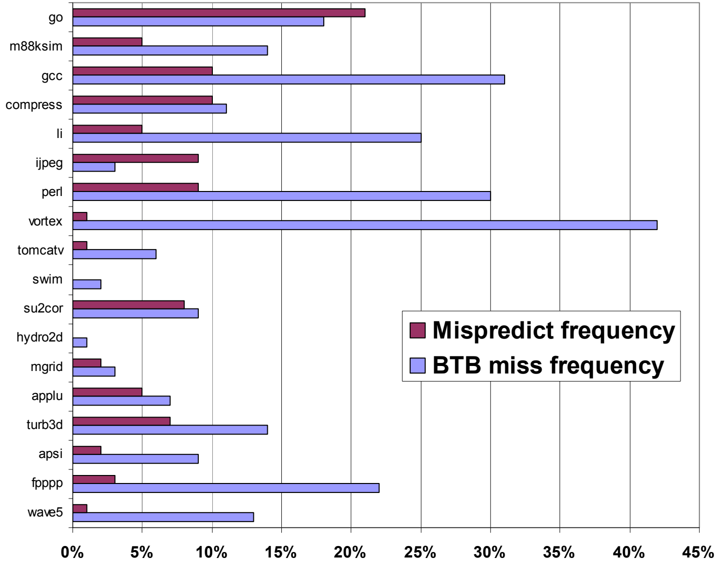


Figure 10
[Bhandarkar and Ding]

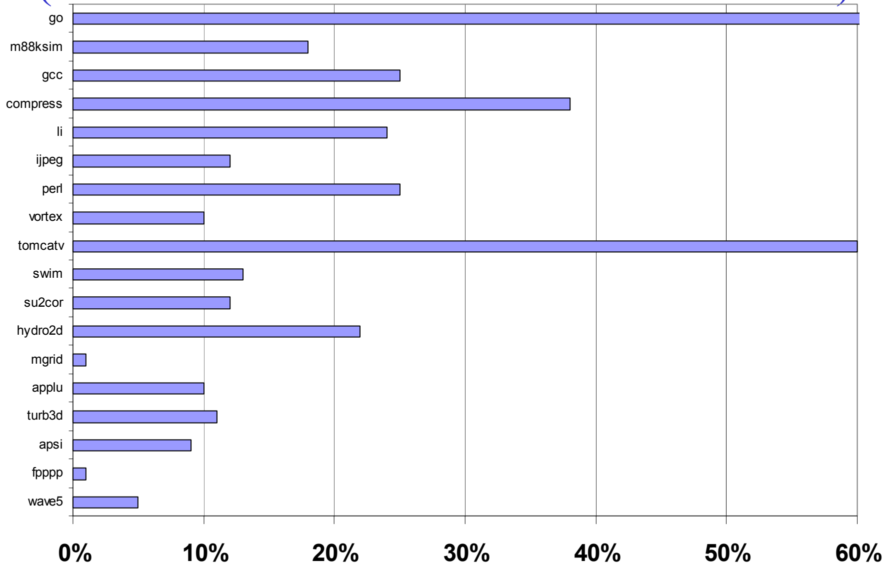
P6 Performance: Branch Mispredict Rate



10% to 40% Miss/Mispredict ratio: 20% avg. (29% integer)

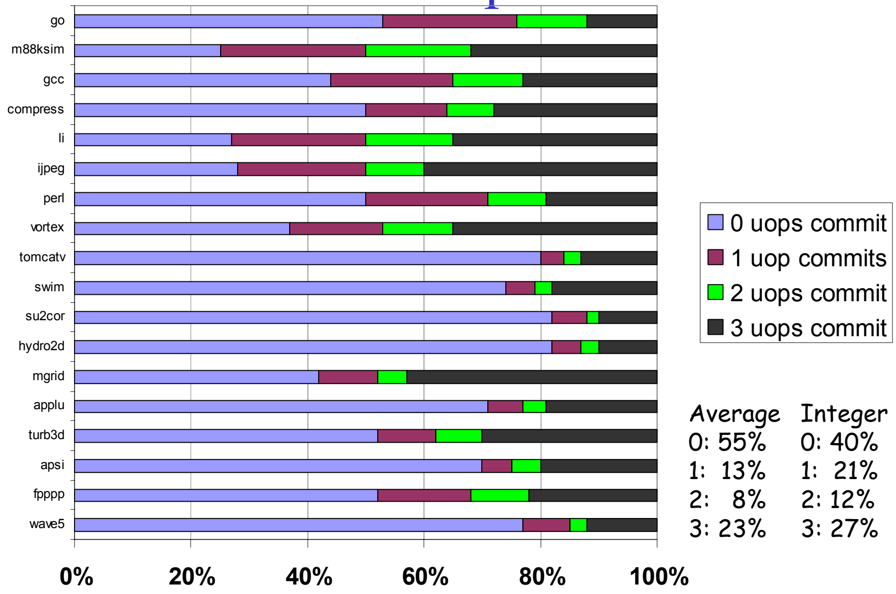
P6 Performance: Speculation rate

(% instructions issued that do not commit)



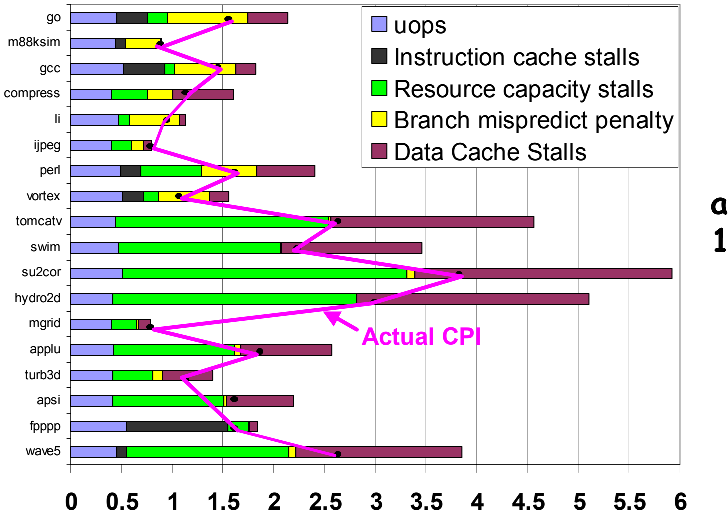
1% to 60% instructions do not commit: 20% avg (30% integer)

P6 Performance: uops commit/clock



Average	Integer
0: 55%	0: 40%
1: 13%	1: 21%
2: 8%	2: 12%
3: 23%	3: 27%

P6 Dynamic Benefit?
Sum of parts CPI vs. Actual CPI



0.8 to 3.8 Clock cycles per instruction: 1.68 avg (1.16 integer)

Ratio of sum of parts vs. actual CPI: 1.38X avg. (1.29X integer)

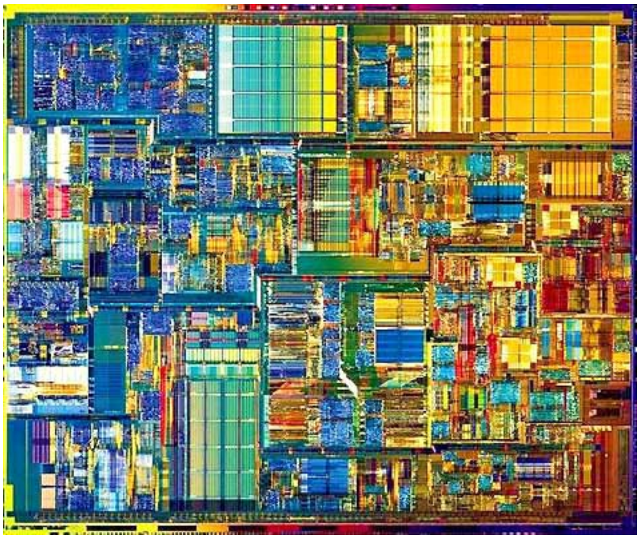
Pentium 4

- Still translate from 80x86 to micro-ops
- P4 has better branch predictor, more FUs
- Instruction Cache holds micro-operations vs. 80x86 instructions
  - no decode stages of 80x86 on cache hit ("Trace Cache")
- Faster memory bus: 400 MHz v. 133 MHz
- Caches
  - Pentium III: L1I 16KB, L1D 16KB, L2 256 KB
  - Pentium 4: L1I 12K uops, L1D 8 KB, L2 256 KB
  - Block size: PIII 32B v. P4 128B; 128 v. 256 bits/clock
- Clock rates:
  - Pentium III 1 GHz v. Pentium IV 1.5 GHz
  - 14 stage pipeline vs. 24 stage pipeline

Trace Cache

- IA-32 instructions are difficult to decode
- Conventional Instruction Cache
  - Provides instructions up to and including taken branch
- Trace cache, records uOps instead of x86 Ops
- Builds them into groups of six sequentially ordered uOps per line
  - Allows more ops per line
  - Avoids clock cycle to get to target of branch

Pentium 4 Die Photo



- 42M xistors
  - PIII: 26M
- 217 mm^2
  - PIII: 106 mm^2
- L1 Execution Cache
  - Buffer 12,000 Micro-Ops
- 8KB data cache
- 256KB L2$

Pentium 4 features

- Multimedia instructions 128 bits wide vs. 64 bits wide => 144 new instructions
  - When used by programs??
  - Faster Floating Point: execute 2 64-bit Fl. Pt. Per clock
  - Memory FU: 1 128-bit load, 1 128-store /clock to MMX regs
- ALUs operate at 2X clock rate for many ops
- Pipeline doesn’t stall at this clock rate: uops replay
- Rename registers: 40 vs. 128; Window: 40 v. 126
- BTB: 512 vs. 4096 entries (Intel: 1/3 improvement)

Pentium, Pentium Pro, P4 Pipeline

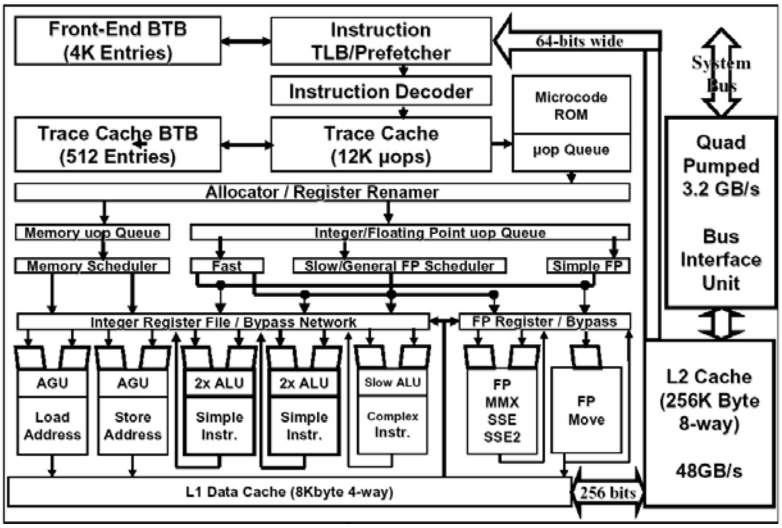
P5 Microarchitecture

P6 Microarchitecture

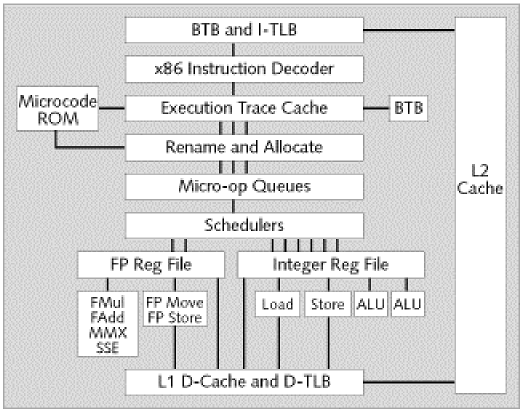
NetBurst Microarchitecture

- Pentium (P5) = 5 stages
  Pentium Pro, II, III (P6) = 10 stages (1 cycle ex)
  Pentium 4 (NetBurst) = 20 stages (no decode)

Pentium 4 Block Diagram



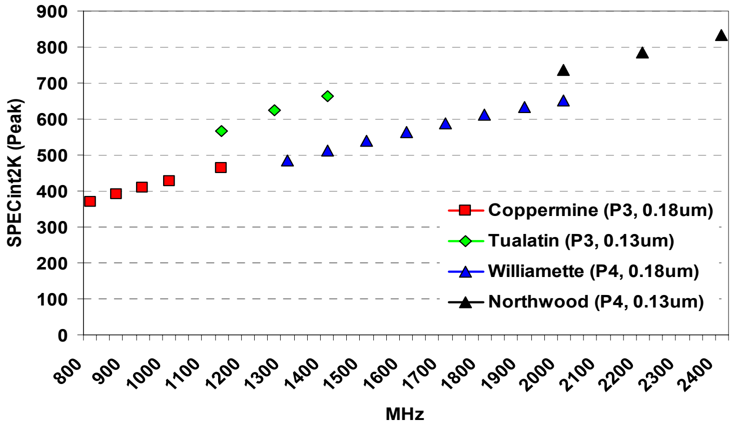
Block Diagram of Pentium 4 Microarchitecture



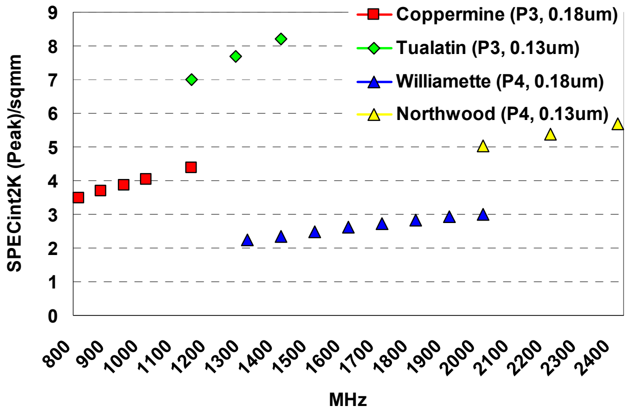
- BTB = Branch Target Buffer (branch predictor)
- I-TLB = Instruction TLB, Trace Cache = Instruction cache
- RF = Register File; AGU = Address Generation Unit
- "Double pumped ALU" means ALU clock rate 2X => 2X ALU F.U.s

---

Pentium III vs. Pentium 4: Performance



Pentium III vs. Pentium 4: Performance / mm²



Williamette: 217mm², Northwood: 146mm², Tualatin: 81mm², Coppermine: 106mm²

Static ILP Overview

- Have discussed methods to extract ILP from hardware
- Why can't some of these things be done at compile-time?
  - Tomasulo scheduling in software (loopy code)
  - ISA changes needed?

Same loop example

- Add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Assume following latency

Instruction producing result	*Instruction using result*	*Execution in cycles*	*Latency in cycles*
FP ALU op	**Another FP ALU op**	**4**	**3**
FP ALU op	**Store double**	**3**	**2**
Load double	**FP ALU op**	**1**	**1**
Load double	**Store double**	**1**	**0**
Integer op	**Integer op**	**1**	**0**

Loop in RISC code: Stalls?

- **Unscheduled MIPS code:**
  - **-To simplify, assume 8 is lowest address**

```
Loop: L.D    F0,0(R1);F0=vector element
      ADD.D  F4,F0,F2;add scalar from F2
      S.D    0(R1),F4;store result
      DSUBUI R1,R1,8 ;decrement pointer 8B (DW)
      BNEZ   R1,Loop ;branch R1!=zero
      NOP            ;delayed branch slot
```

Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1)  ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2  ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4  ;store result
7       DSUBUI R1,R1,8   ;decrement pointer 8B (DW)
8       stall
9       BNEZ   R1,Loop   ;branch R1!=zero
10      stall            ;delayed branch slot
```

Unscheduled 10 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       DSUBUI R1,R1,8
3       ADD.D  F4,F0,F2
4       stall
5       BNEZ   R1,Loop   ;delayed branch
6       S.D    8(R1),F4  ;altered when move past DSUBUI
```

Swap BNEZ and S.D by changing address of S.D

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

6 clocks, but just 3 for execution, 3 for loop overhead; How to make it faster?

Unroll Loop Four Times (unscheduled code)

```
1  Loop:L.D    F0,0(R1)
2       ADD.D  F4,F0,F2
3       S.D    0(R1),F4      ;drop DSUBUI & BNEZ
4       L.D    F0,-8(R1)
5       ADD.D  F4,F0,F2
6       S.D    -8(R1),F4     ;drop DSUBUI & BNEZ
7       L.D    F0,-16(R1)
8       ADD.D  F4,F0,F2
9       S.D    -16(R1),F4    ;drop DSUBUI & BNEZ
10      L.D    F0,-24(R1)
11      ADD.D  F4,F0,F2
12      S.D    -24(R1),F4
13      DSUBUI R1,R1,#32     ;alter to 4*8
14      BNEZ   R1,LOOP
15      NOP
```

How can remove them?

Removing the name dependencies?

```
1  Loop:L.D    F0,0(R1)
2       ADD.D  F4,F0,F2
3       S.D    0(R1),F4      ;drop DSUBUI & BNEZ
4       L.D    F6,-8(R1)
5       ADD.D  F8,F6,F2
6       S.D    -8(R1),F8     ;drop DSUBUI & BNEZ
7       L.D    F10,-16(R1)
8       ADD.D  F12,F10,F2
9       S.D    -16(R1),F12   ;drop DSUBUI & BNEZ
10      L.D    F14,-24(R1)
11      ADD.D  F16,F14,F2
12      S.D    -24(R1),F16
13      DSUBUI R1,R1,#32     ;alter to 4*8
14      BNEZ   R1,LOOP
15      NOP
```

1 cycle stall

2 cycles stall

1 cycle stall

This is why we call it register renaming!

Rewrite loop to minimize stalls?

15 + 4 x (1+2) + 1 = 28 clock cycles, or 7 per iteration

Assumes R1 is multiple of 4

Loop Unrolling Problem

- Do not know loop iteration counts…
- Suppose it is *n*, and we would like to unroll the loop to make *k* copies of the body
- Generate a pair of consecutive loops:
  - 1st executes (*n* mod *k*) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times
  - For large values of *n*, most of the execution time will be spent in the unrolled loop

---

Unrolled Loop That Minimizes Stalls

```
1  Loop:L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    0(R1),F4
10      S.D    -8(R1),F8
11      DSUBUI R1,R1,#32
12      S.D    16(R1),F12
13      BNEZ   R1,LOOP
14      S.D    8(R1),F16 ; 8-32 = -24
```

- Scheduling Assumptions?
  - Move store past DSUBUI even though it changes register (must change offset)
  - Alias analysis: move loads before stores
  - Easy for humans to see this, what about compilers?

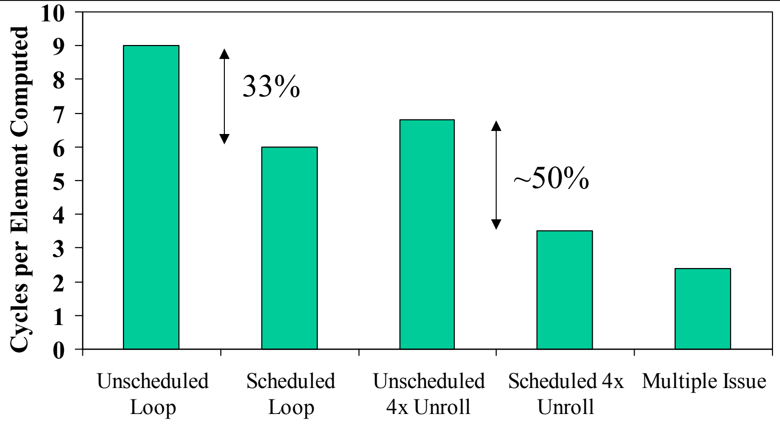
14 clock cycles, or 3.5 per iteration

Multiple Issue:Loop Unrolling and Static Scheduling

```
Cycle  Integer Pipe            Float Pipe
1      L.D    F0,0(R1)
2      L.D    F6,-8(R1)        ADD.D  F4,F0,F2
3      L.D    F10,-16(R1)      ADD.D  F8,F6,F2
4      L.D    F14,-24(R1)      ADD.D  F12,F10,F2
5      L.D    F18,-32(R1)      ADD.D  F16,F14,F2
6      S.D    0(R1),F4         ADD.D  F20,F18,F2
7      S.D    -8(R1),F8
8      S.D    -16(R1),F12
9      DSUBUI R1,R1,#40
10     S.D    16(R1), F16             ;40-24 = 16
11     BNEZ   R1,LOOP
12     S.D    8(R1),F16               ; 40-32 = 8
```

12 clock cycles, or 2.4 per iteration

Loop Performance



Get 1.7x from unrolling (6->3.5) and 1.5x (3.5 -> 2.5)from Dual Issue

Compiler Scheduling Requirements

- Compiler concerned about *dependencies* in program
  - Pipeline determines if these become *hazards*
- Obviously we want to avoid this when possible (stalls)
- Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- Dependencies limit ILP
- Dependency analysis
  - Easy to determine for registers (fixed names)
  - Hard for memory ("memory disambiguation" problem):

Compiler Scheduling: Memory Disambiguation

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Compiler knows that if R1 doesn't change then:

`0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)`

Guarantees that there were no dependencies between loads and stores so they could be moved by each other

Compiler Loop Unrolling

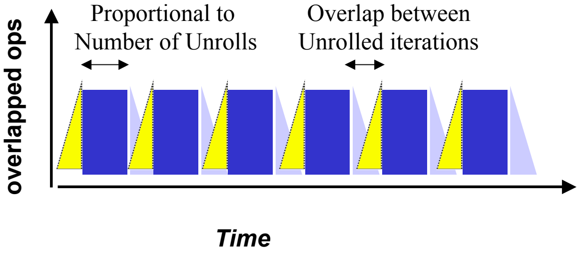
1. Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
2. Determine unrolling the loop would be useful by finding that the loop iterations were independent
3. Rename registers to avoid name dependencies
4. Eliminate extra test and branch instructions and adjust the loop termination and iteration code
5. Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
    - requires analyzing memory addresses and finding that they do not refer to the same address.
6. Schedule the code, preserving any dependences needed to yield same result as the original code

Loop Unrolling Limitations

- Decrease in amount of overhead amortized per unroll
    - Diminishing returns in reducing loop overheads
- Growth in code size
    - Can hurt instruction-fetch performance
- Register Pressure
    - Aggressive unrolling/scheduling can exhaust 32 register machines

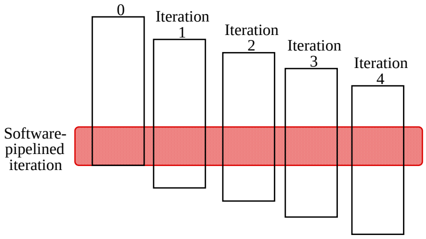
Loop Unrolling Problem

- Every loop unrolling iteration requires pipeline to fill and drain
- Occurs every *m*/*n* times if loop has *m* iterations and is unrolled *n* times



More advanced Technique: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)

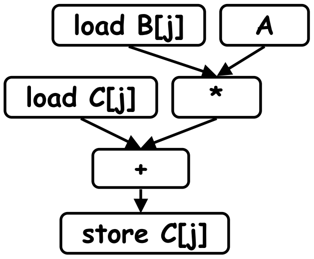


Software Pipelining

- Now must optimize inner loop
- Want to do as much work as possible in each iteration
- Keep all of the functional units busy in the processor

```
for(j = 0; j < MAX; j++)
  C[j] += A * B[j];
```

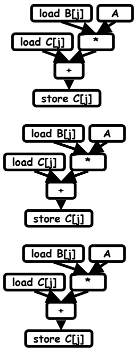
Dataflow graph:



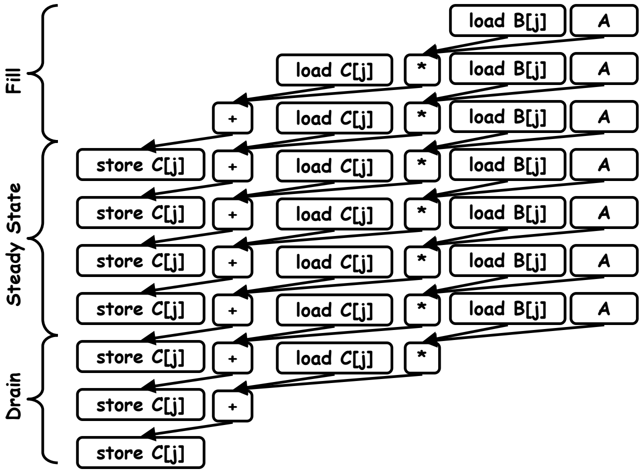
Software Pipelining Example

```
for(j = 0; j < MAX; j++)
  C[j] += A * B[j];
```

Not pipelined:



Pipelined:



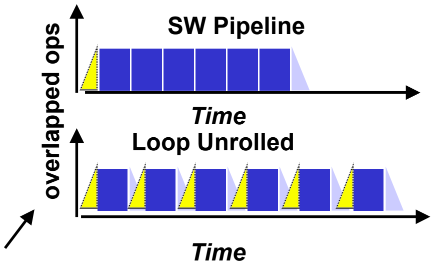
Software Pipelining Example

Before: Unrolled 3 times

```
L.D       F0,0(R1)
ADD.D     F4,F0,F2
S.D       0(R1),F4
L.D       F6,-8(R1)
ADD.D     F8,F6,F2
S.D       -8(R1),F8
L.D       F10,-16(R1)
ADD.D     F12,F10,F2
S.D       -16(R1),F12
DSUBUI    R1,R1,#24
BNEZ      R1,LOOP
```

After: Software Pipelined

```
S.D    0(R1),F4 ; Stores M[i]
ADD.D  F4,F0,F2 ; Adds to M[i-1]
L.D    F0,-16(R1);Loads M[i-2]
DSUBUI R1,R1,#8
BNEZ   R1,LOOP
```



- **Symbolic Loop Unrolling**
  - **Maximize result-use distance**
  - **Less code space than unrolling**
  - **Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling**

5 cycles per iteration

Software Pipelining vs. Loop Unrolling

- Software pipelining is *symbolic* loop unrolling
  - Consumes less code space
- Actually they are targeting different things
  - Both provide a better scheduled inner loop
  - Loop Unrolling
    - Targets loop overhead code (branch/counter update code)
  - Software Pipelining
    - Targets time when pipelining is filling and draining
  - Best performance can come from doing both

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];     /* S1 */
        B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel?

Schedule for next few lectures

- Next Time (Mar. 15th)
  - VLIW vs. Superscalar
  - Global Scheduling
    - Trace Scheduling, Superblocks
  - Hardware support for software scheduling
  - Comparison between hardware and software ILP

- Next Next Time (Mar. 17th) – HW#3 Due
  - Itanium (IA64) case study
  - Review for midterm (Mar 22nd)