Microcoding, VLIW and Vector Computers

Suvinay Subramanian

6.823 Spring 2016

Microcoding

- » Abstraction layer between hardware and architecture of computer
  - i.e. separates ISA from actual hardware implementation details.
- » Layer of hardware-level instructions that implement higher-level (i.e. ISA) instructions

Benefits of Microcoding

» Originally developed as a simpler method for developing control logic
  - Direct combinational logic for powerful instruction set can be complex, prone to bugs/difficult to debug
  - Multi-step addressing modes, varied length instructions etc.

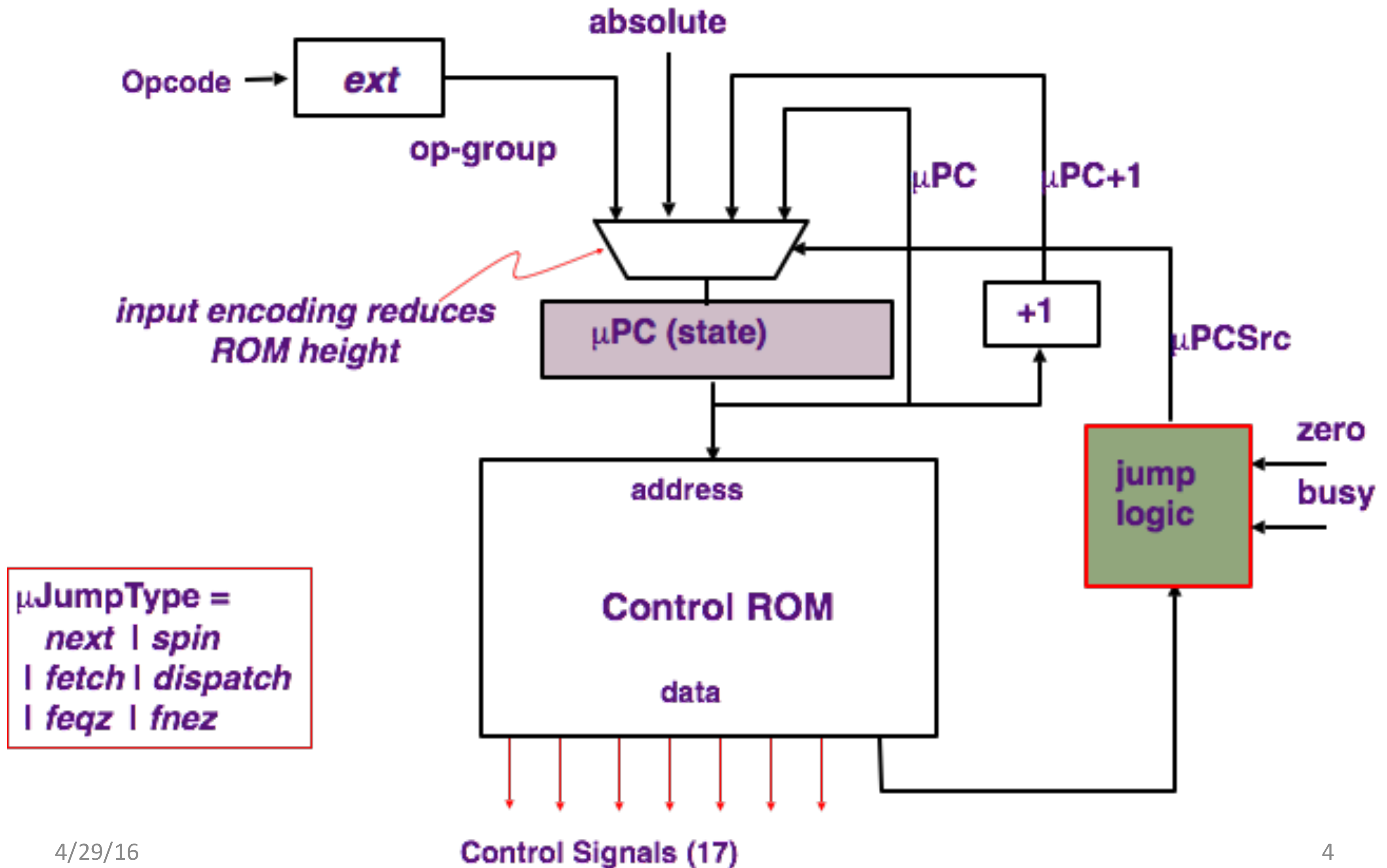
» Enables complex ISAs
  - Reduced instruction fetch bandwidth; smaller code footprint

» Same ISA, different implementations
  - Flexibility in how data paths, micro-arch. blocks designed
  - IBM 360: Model 30, Model 40 etc.

» Allows for "patching" in the field

» New instructions supported without modifying datapath

Microcode Implementation



Microcode Fragments

State	Control points	next-state
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
…		
ALU_0	A ← Reg[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

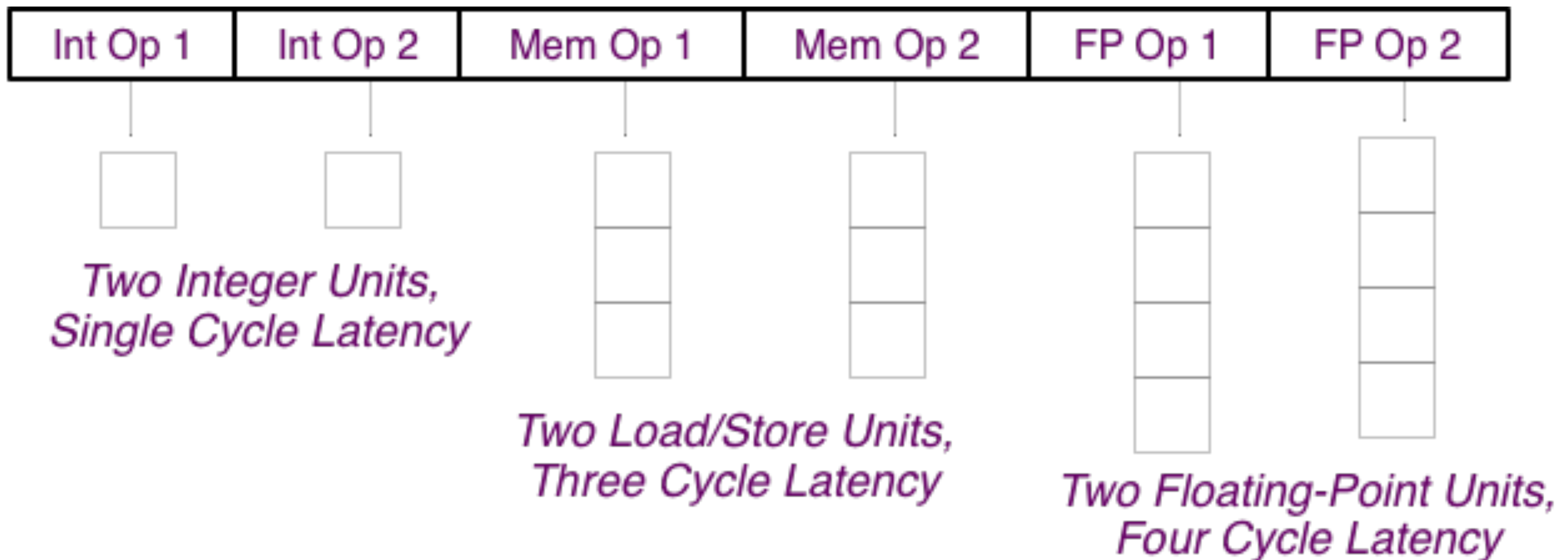
VLIW

» Premise: Static instruction scheduling + super-scalar execution to extract ILP.

» Tradeoff: Complex hardware vs Complex compiler "Conservation of complexity"

- OoO processors do dynamic scheduling
  Figure out independent instructions on-the-fly
- VLIW machines: Compilers figure out independent instructions and schedules them suitably

VLIW Hardware



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified

VLIW Software

» Key Questions:
- How do we find independent instructions to fetch/execute?
- How to enable more compiler optimizations?

» Key Ideas:
- Get rid of control flow
  - Predicated execution, loop unrolling
- Optimize frequently executed code-paths
  - Trace scheduling
- Others: Software pipelining, speculative execution

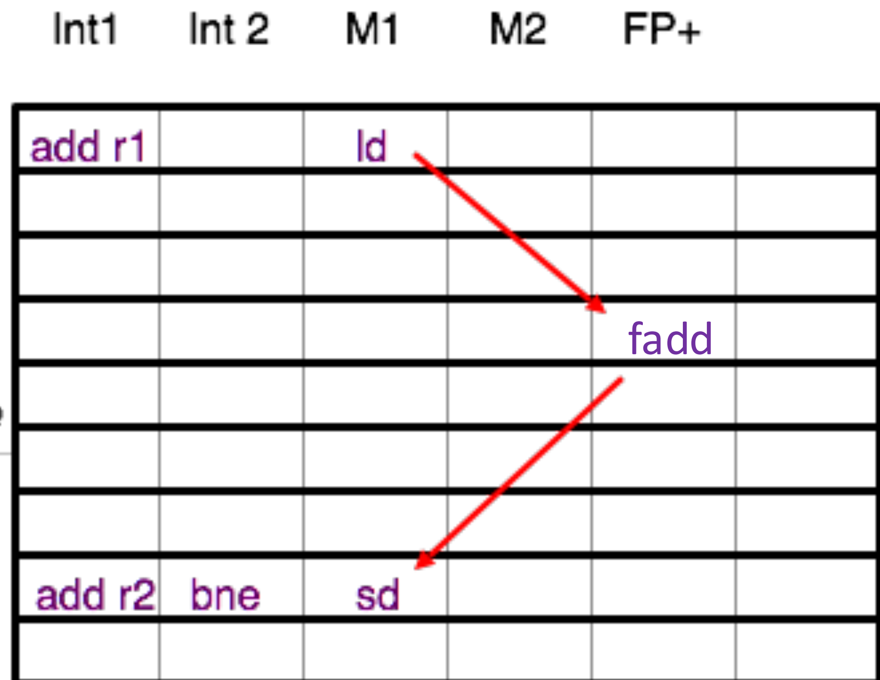
Loop Unrolling

```
for (i=0; i<N; i++)
   B[i] = A[i] + C;
```

Compile

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Schedule



How many FP ops/cycle?

1 fadd / 8 cycles = 0.125

Loop Unrolling

- Unroll loop to perform M iterations at once
  - Get more independent instructions
  - Need to be careful about case where M is not a multiple of number of loop iterations

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Loop Unrolling

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule

loop:

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

4 fadds / 11 cycles = 0.36

Loop Unrolling

1. Combine M iterations of loop
2. Pipeline schedule to reduce RAW stalls
   - In the example above, notice that we move (re-order) loads to the top
3. Rename registers
   - f1, f2, f3, f4

Software Pipelining

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
```

	Int1	Int 2	M1	M2	FP+	FPx
prolog			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
iterate (loop:)			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
	add r1	bne	ld f4	sd f8	fadd f8	
epilog				sd f5	fadd f5	
				sd f6	fadd f6	
		add r2		sd f7	fadd f7	
		bne		sd f8	fadd f8	
				sd f5		

4 fadds / 4 cycles = 1

Loop Unrolling Limitations

» Code growth

» Does not handle inter-iteration dependences well

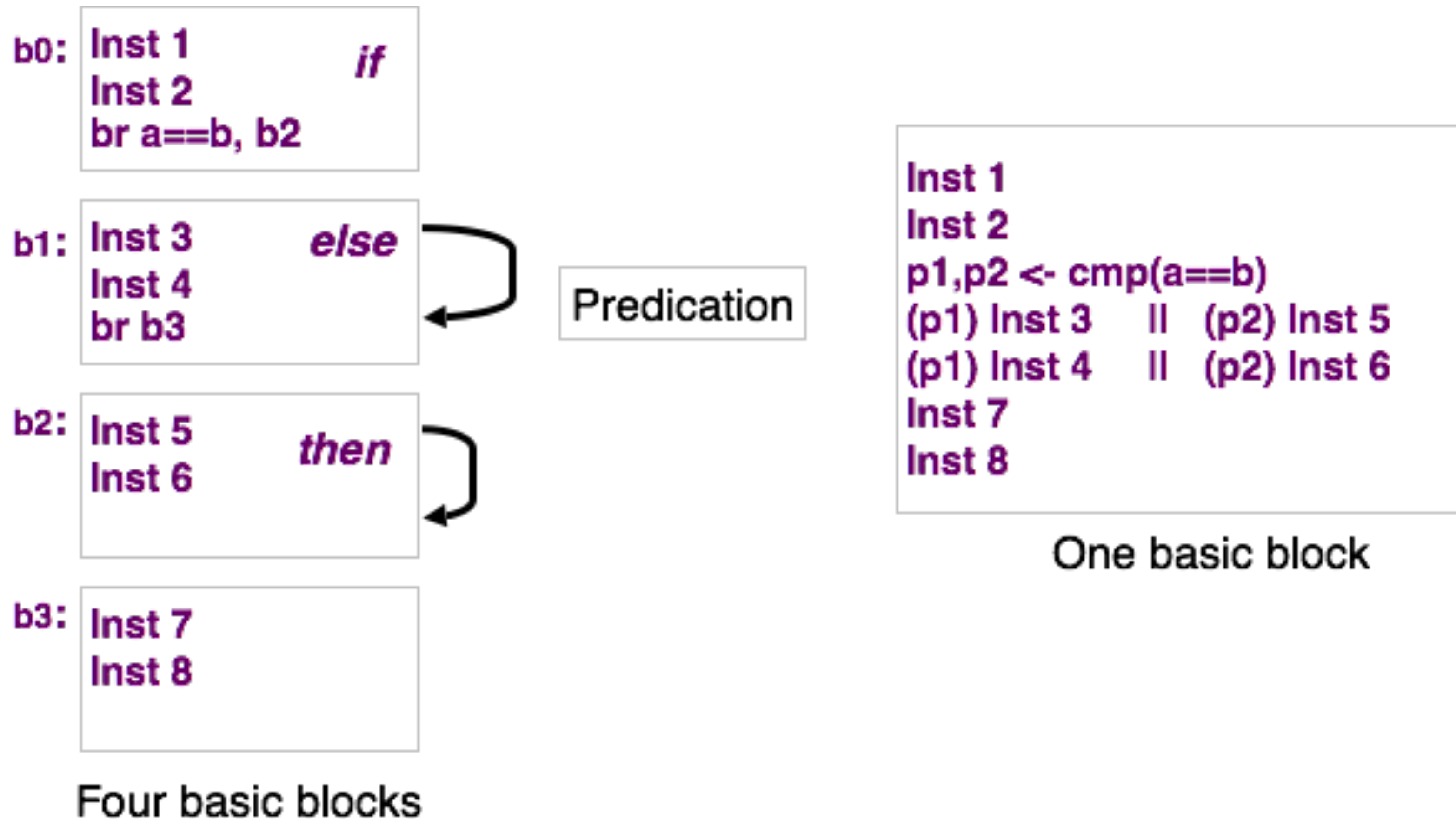
Predicated Execution

» Limited ILP within a basic-block; branches limit available ILP

» Idea: Eliminate hard-to-predict branches by converting control dependence to data dependence

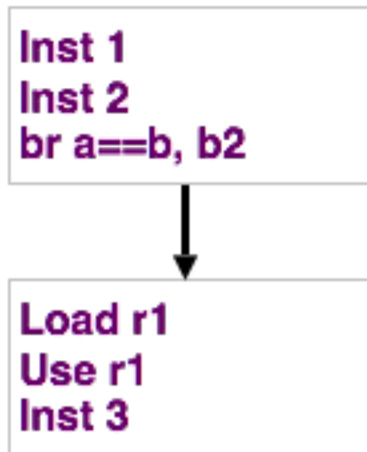
- Each instruction (within the branch basic block) has a predicate bit set
- Only instructions with true predicates are executed and committed. Others are treated as nops.

Predicated Execution

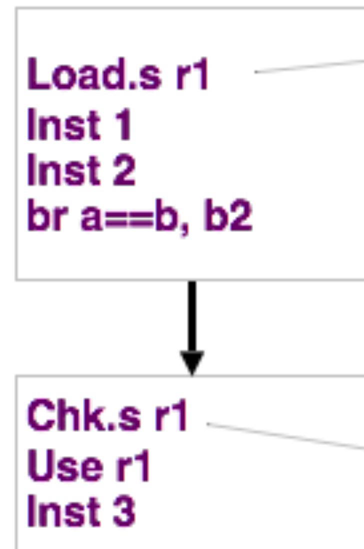


Speculative Execution

» Move instructions above branches to explore more ILP options



Can't move load above branch because might cause spurious exception



Speculative load never causes exception, but sets "poison" bit on destination register

Check for exception in original home block jumps to fixup code if exception detected

Speculative Execution

```
int32_t calculateSomething(int32_t *a, int32_t *b) {
    int32_t result;
    if (m_p > m_q + 1) {
        result = a[*b];
    } else {
        result = defaultValue;
    }
    return result;
}
```

Speculative execute load of `a[*b]` before branch condition is resolved.
Say: `m_p, m_q` are bound checks.

1. What if `m_p < m_q + 1`? Say, `b = nullptr`.
   Exception!
2. But exceptions can arise in other ways. What if a[*b] is valid, but there is a page-fault? Exception! Trap to OS routine.

Trace Scheduling

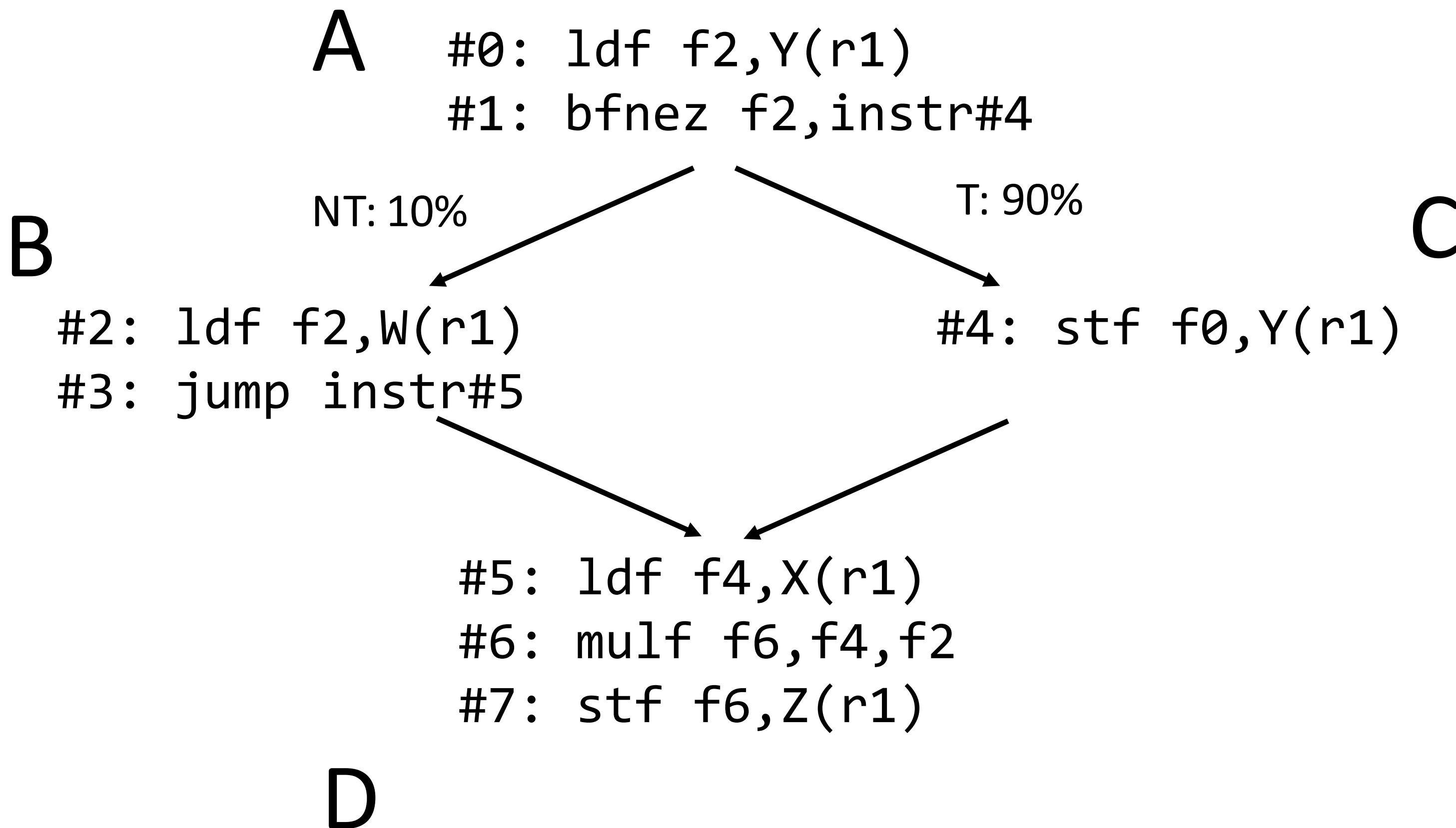
» Idea: For non-loop situations:
  - Find common path in program trace
  - Re-align basic blocks to form straight-line trace
    - Trace: Fused basic-block sequence
  - Schedule trace
  - Create fixup code in case trace != actual path
    - Can be nasty

Trace Scheduling

```
A = Y[i]
If (A == 0)
    A = W[i]
Else:
    Y[i] = 0
Z[i] = A * X[i]
```

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

Trace Scheduling



Trace Scheduling

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

Recovery / repair code

```
#2 : ldf f2,W(r1)
#5’: ldf f4,X(r1)
#6’: mulf f6,f4,f2
#7’: stf f6,Z(r1)
```

A, C, D superblock (trace)

» Trace scheduling can be combined with other techniques:

- What if we moved #5, #6 before #4? Speculative load issue
- What if we moved #5, #6 even further, above #1?
- What if branch was biased the other way?
- What if branch was evenly biased – 50%, 50%? Predication

VLIW Summary

» Loop unrolling
  - Reduces branch frequency
  - Tighter packing of instructions
  - Dependences b/w iterations; handling “extra” iterations

» Predicated execution, speculative execution
  - Control-flow
  - Control-flow, Load-store speculation

» Trace scheduling
  - Recovery code
  - Combined with other techniques above; moving code upward/downward may provide benefits

Vector Computers

» Idea: Operate on vectors instead of scalars
  - ISA is more expressive, therefore captures more information

» Advantages:
  - No dependences within a vector
  - Reduced instruction fetch bandwidth
  - (Sometimes) regular memory access pattern
  - No need to explicitly code loops

» Pitfalls:
  - Only works if code sequence (or parallelism) is regular

Vector Computers

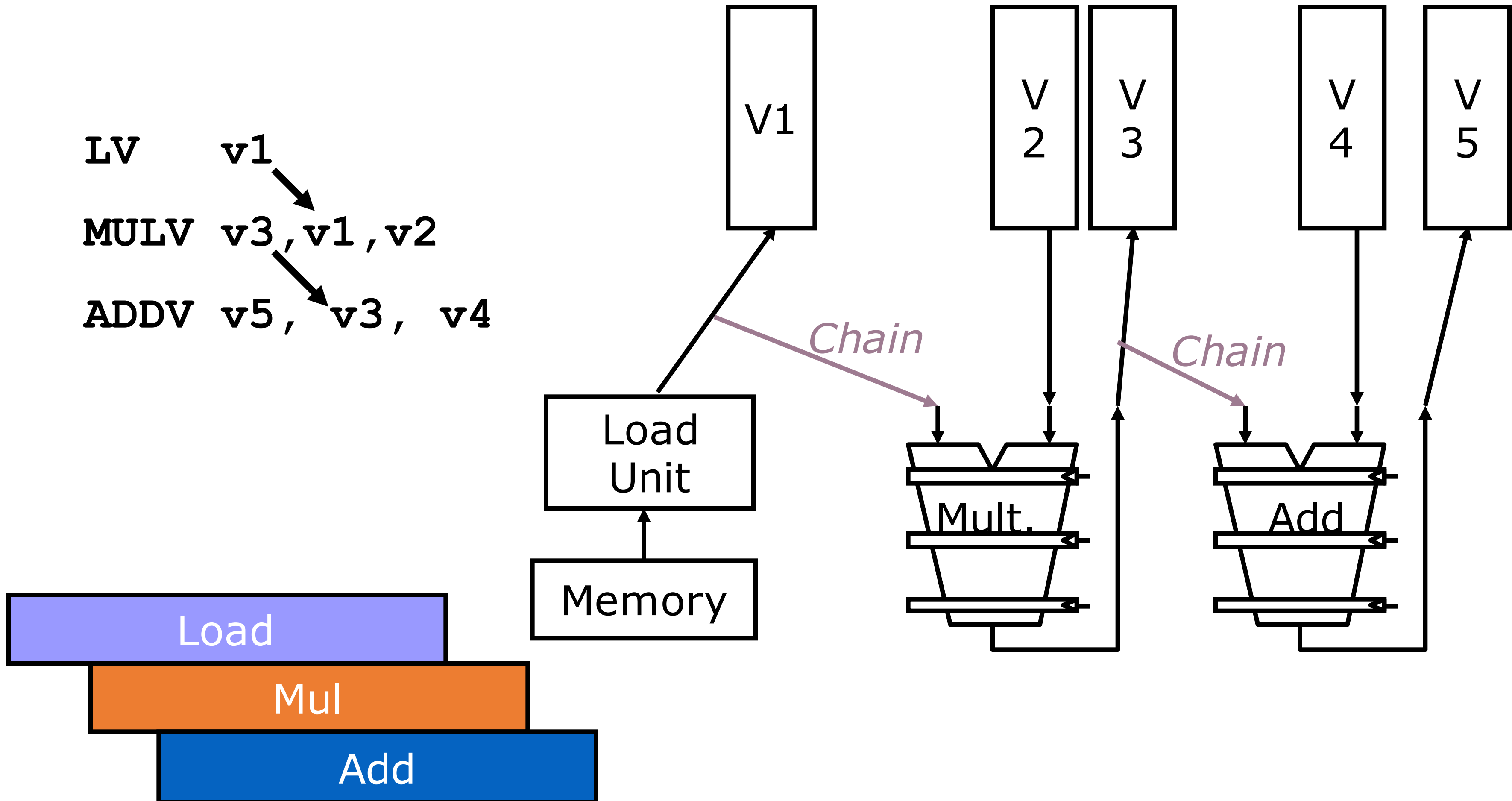
To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is unsuccessful; it's back to the drawing boards again. Many

Vector Computers

Terminology:

» Vector length register (VLR)
» Conditional execution using vector mask (VM)
» Vector lanes
» Vector chaining

Vector Computers



That’s all folks!