VLIW, Vector Processors, and Accelerators

Ryan Lee

Adapted from prior course offerings

6.823 Fall 2021

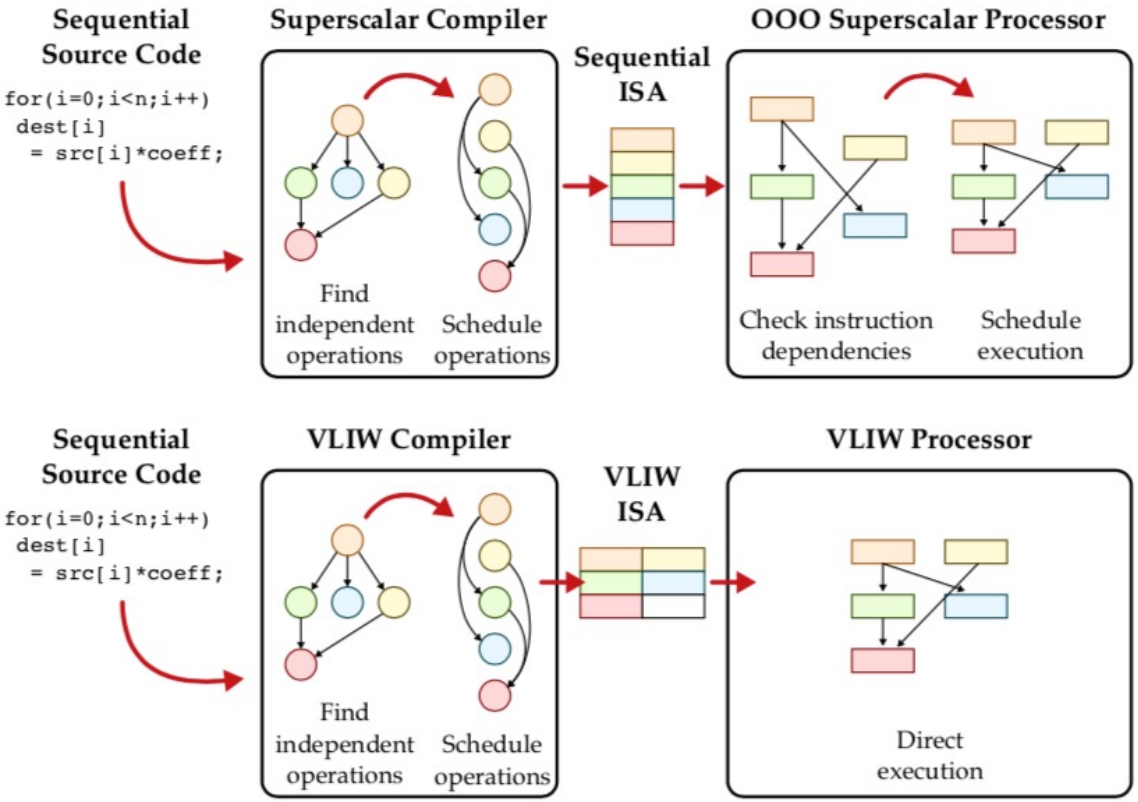
VLIW

- Motivation: OoO processors introduce complex, inefficient hardware for uncovering ILP

- The Compiler
  - Guarantees intra-instruction parallelism
  - Schedules (reorders) to maximize parallel execution

- The Architecture:
  - Allows parallelism between operations within an instructions (No dependency checks)
  - Provide deterministic latency for all operations (no bypasses)

VLIW Motivation



From Cornell University ECE 4750 Handout #15, Courtesy Chris Batten and course staff

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

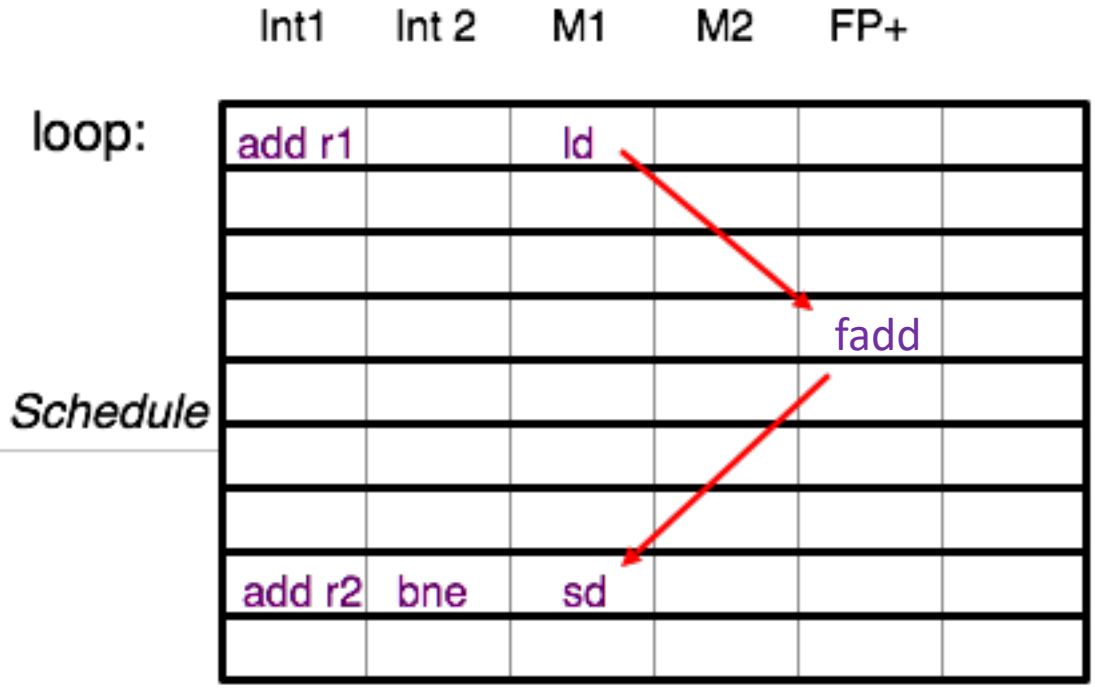
Loop Unrolling

```
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

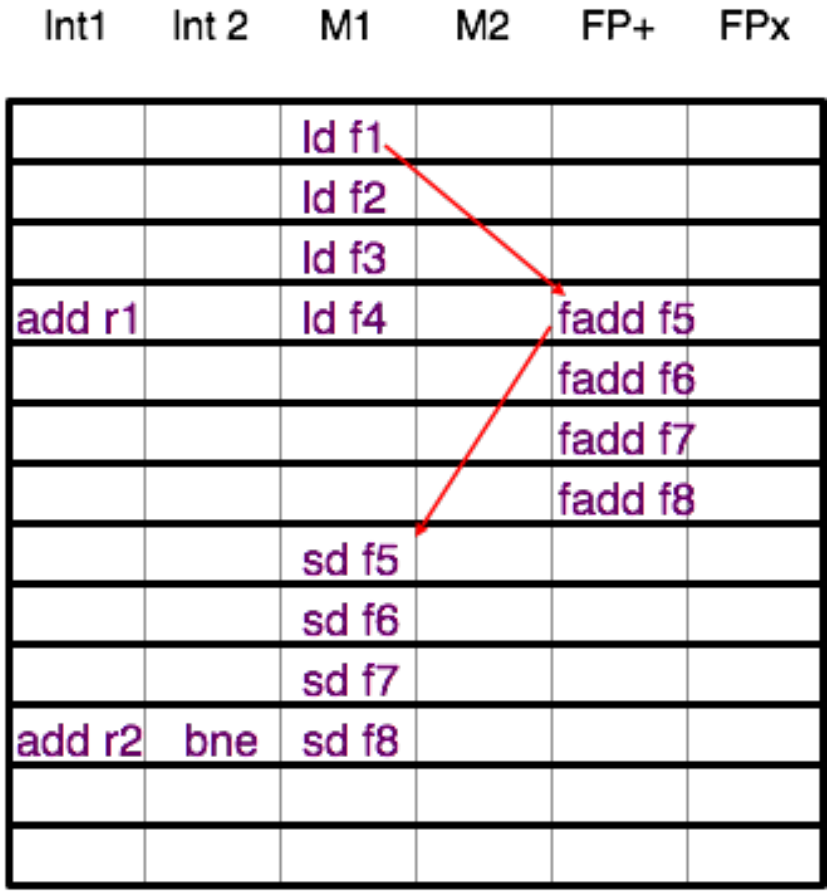
```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

```
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



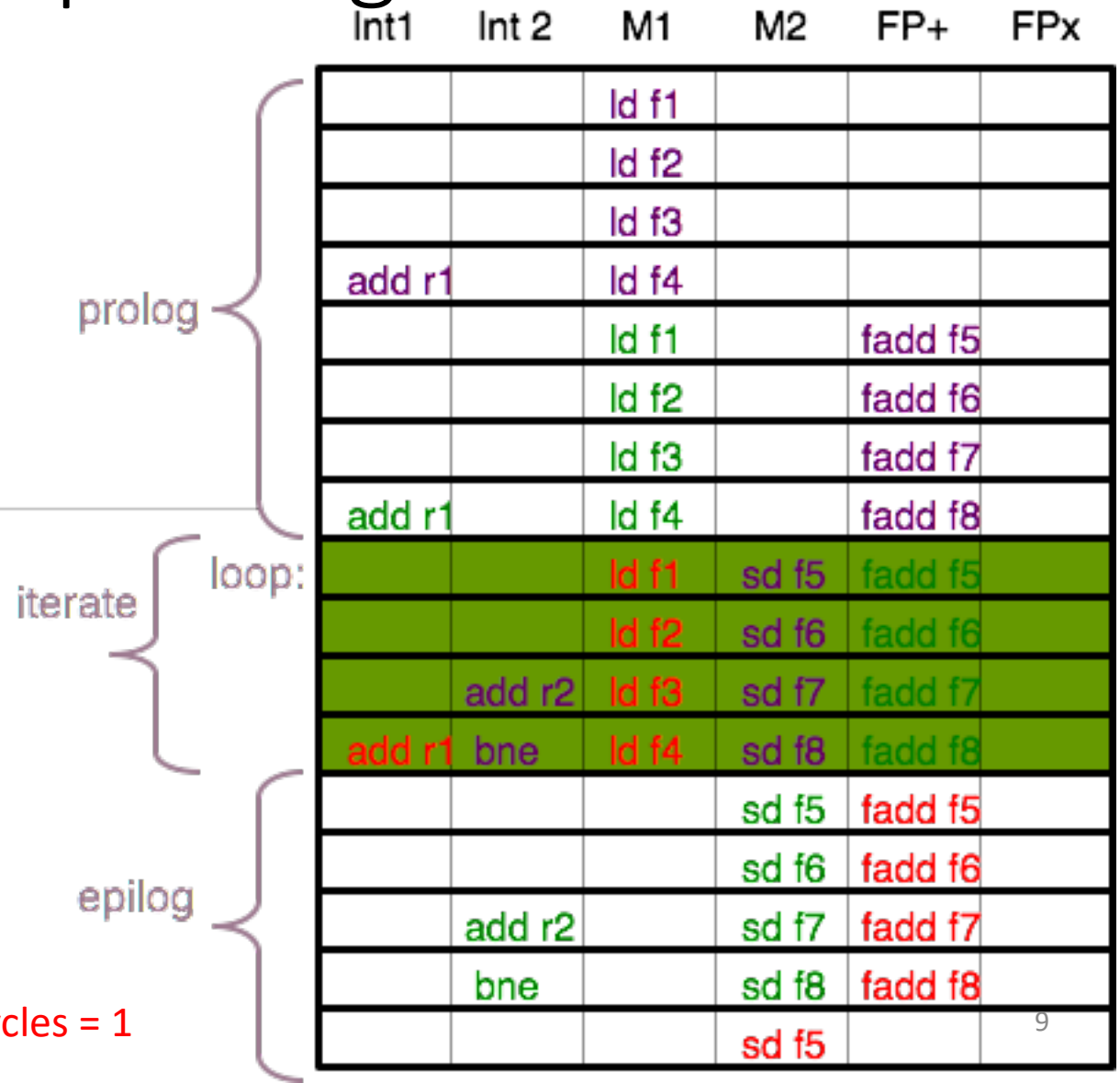
4 fadds / 11 cycles = 0.36

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
```



4 fadds / 4 cycles = 1

Loop Unrolling Limitations

- Code growth
- Does not handle inter-iteration dependences well

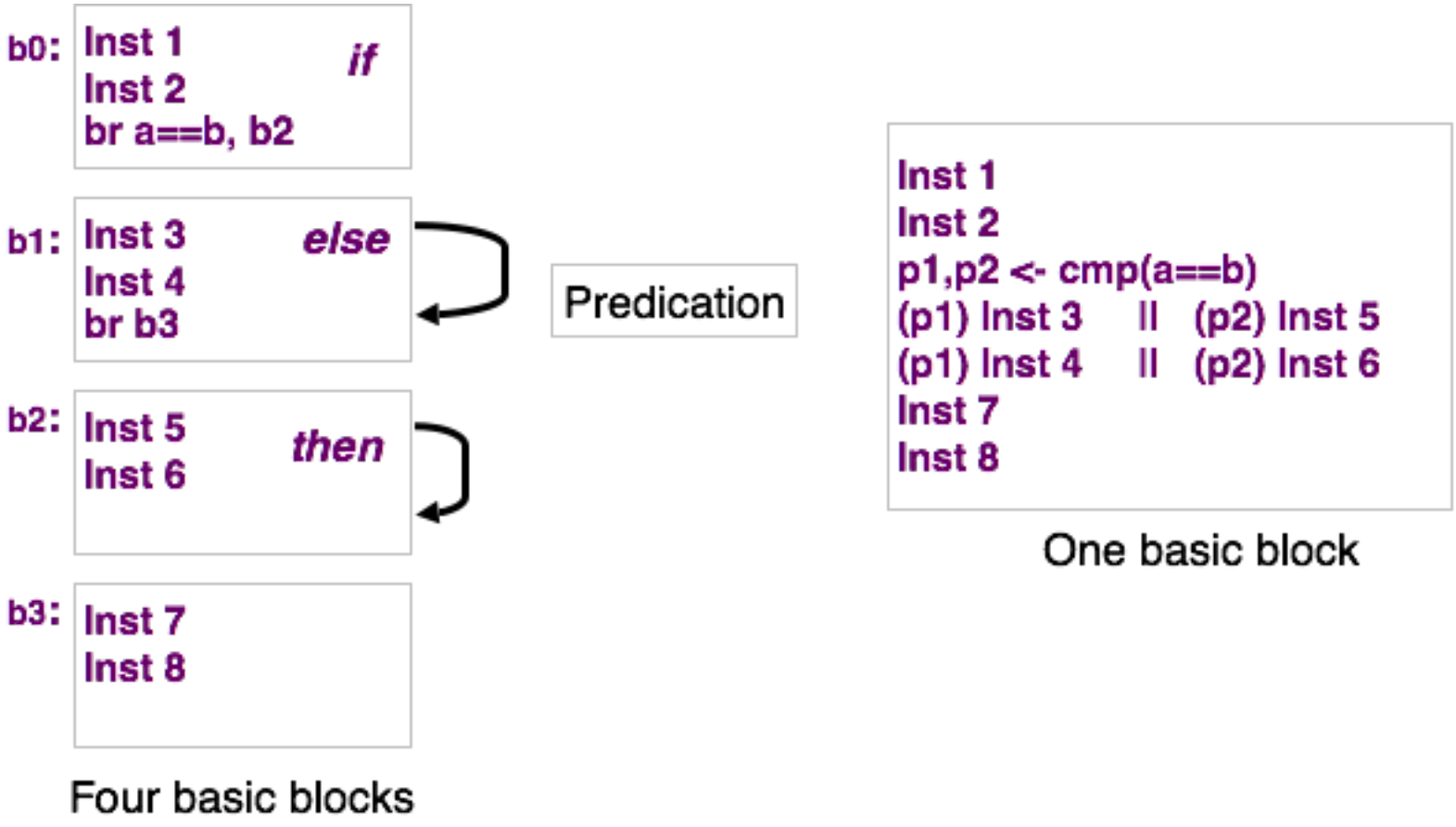
Predicated Execution

» 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



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

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
  - Amortized cost of instruction fetch and decode
  - (Sometimes) regular memory access pattern
  - No need to explicitly code loops
- Pitfalls:
  - Only works if code sequence (or parallelism) is regular

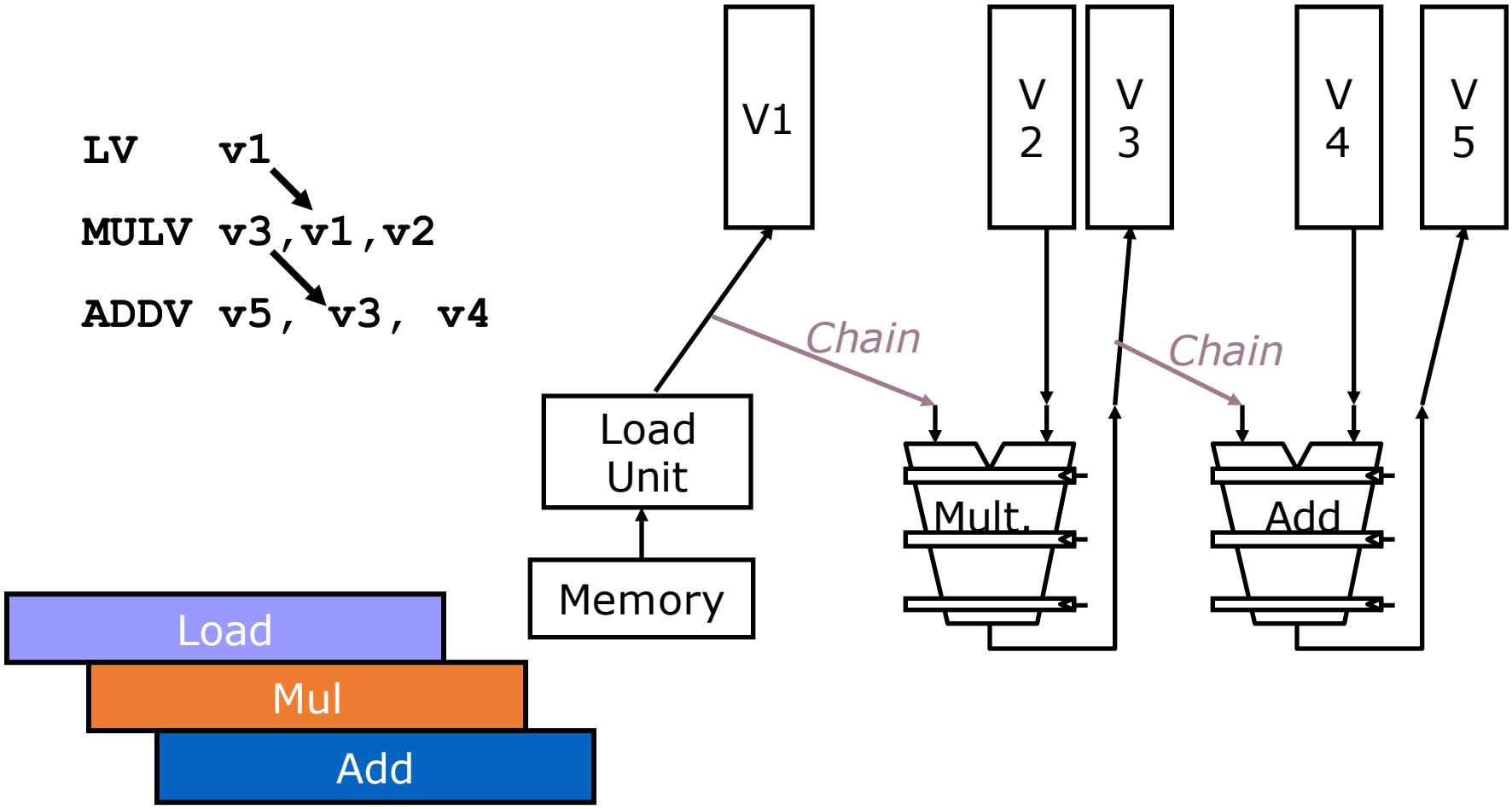
Vector Computers

Terminology:

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

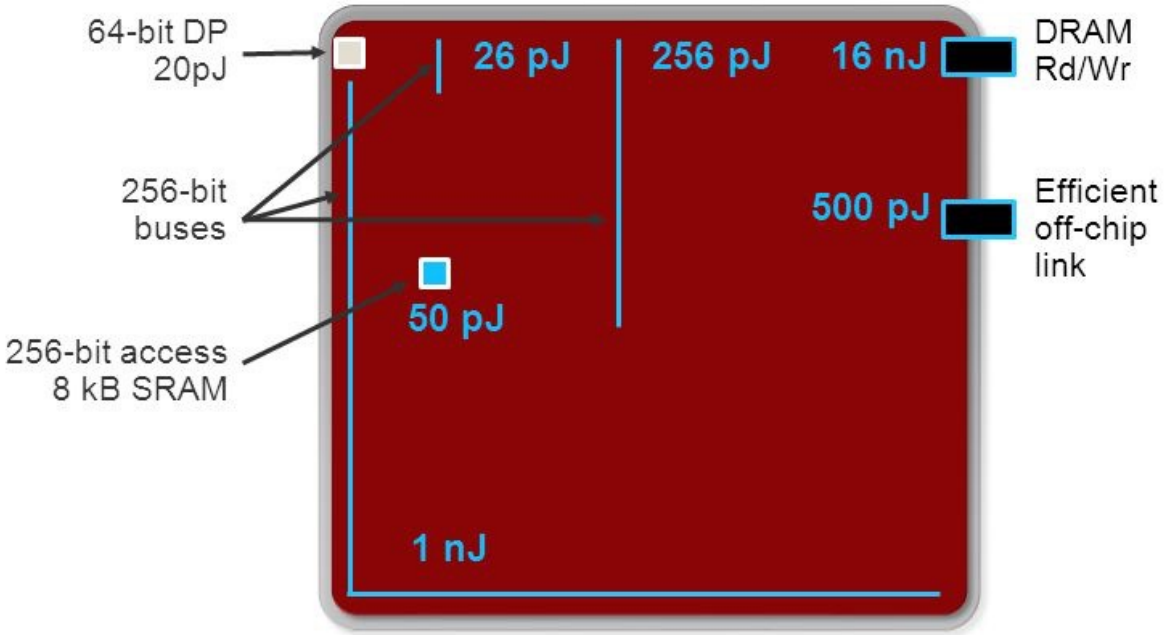
Vector Computers

```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Accelerators

» Motivation in lecture: Cost of Data movement
  - Using the limited number of transistors more efficiently.

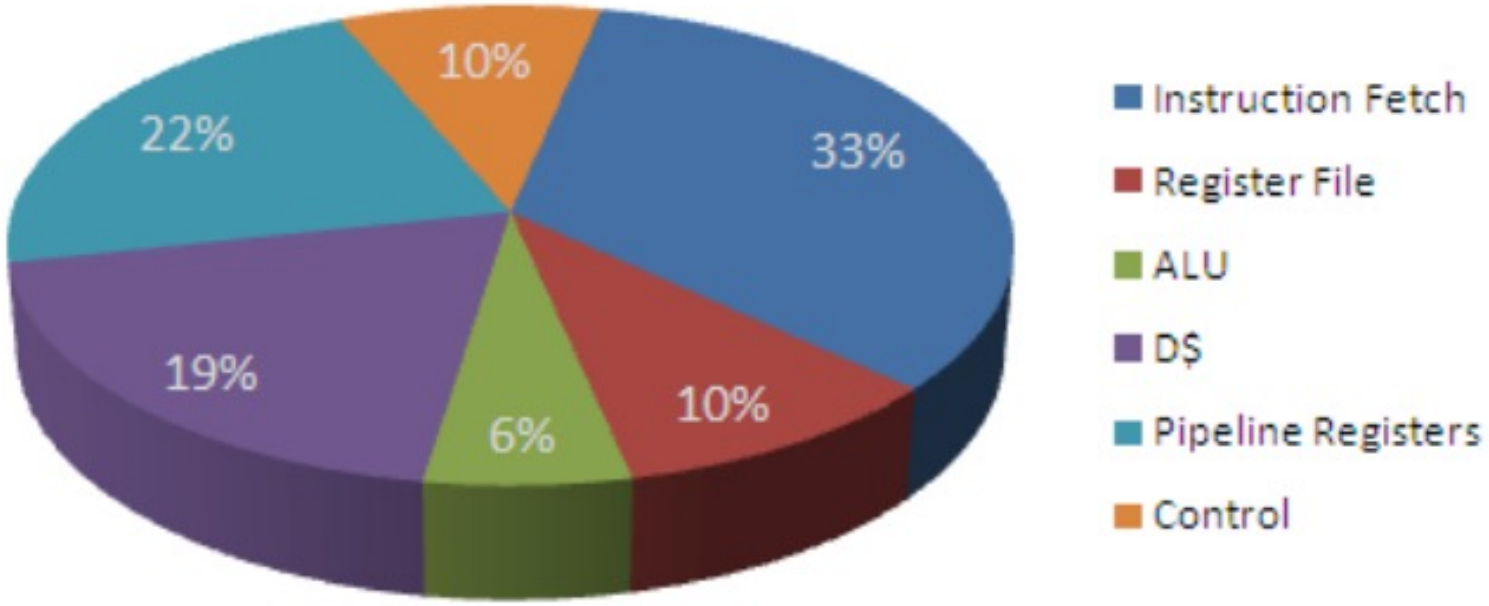


Accelerators

- Another motivation: Inefficiency from control overheads in general-purpose processors
  - Sequential stream of *instructions*
  - Allows processors to be a generalist: able to handle every task
  - But what is the actual energy cost of an operation?

Case Study: H.264 Encoder

» CMP energy breakdown:



We can get away with much less energy/op

- Remove/Amortize overhead of instruction fetch, decode
  - Fixed control flow
  - Custom datapaths
  - Dataflow execution
- Custom hardware for low bit-width operations
  - Similar to SIMD implementations
- Reuse data as much as possible