ECE 4750 Computer Architecture, Fall 2016

T15 Advanced Processors: VLIW Processors

School of Electrical and Computer Engineering
Cornell University

revision: 2016-11-28-17-10

1. Motivating VLIW Processors

We will use a simple vector-scalar multiply (VSMUL) kernel to explore both VLIW and SIMD processors, so we first need to establish the baseline performance of this kernel on our canonical TinyRV1 in-order single-issue processor and TinyRV1 out-of-order superscalar processor.

```
for ( int i = 0; i < n; i++ )
  dest[i] = src[i] * coeff;
```

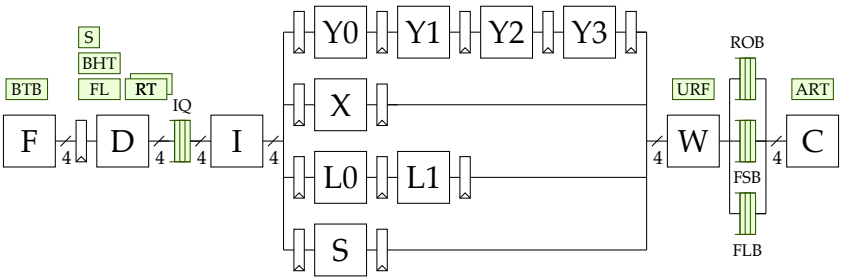
For all problems
we will assume `n` is 64.

```
loop:
  lw   x1, 0(x2)
  mul  x3, x1, x4
  sw   x3, 0(x5)
  addi x2, x2, 4
  addi x5, x5, 4
  addi x7, x7, -1
  bne  x7, x0, loop
```

Calculate the performance of this loop on the canonical TinyRV1 in-order single-issue processor with a four-cycle iterative multiplier.

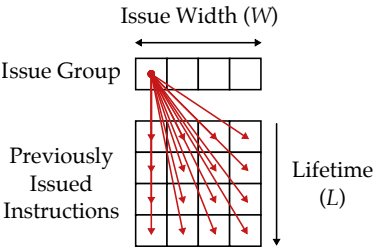
`lw`																			
`mul`																			
`sw`																			
`addi`																			
`addi`																			
`addi`																			
`bne`																			
`opA`																			
`opB`																			
`lw`																			

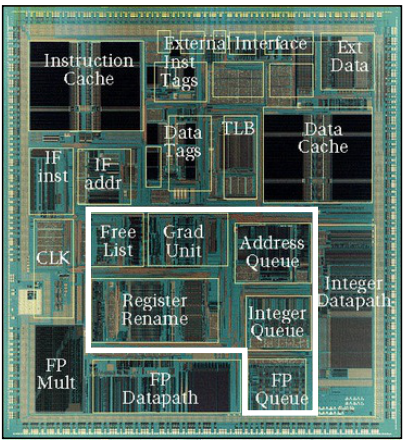
Calculate performance of VSMUL on the canonical TinyRV1 OOO quad-issue processor with register renaming, OOO memory disambiguation, perfect branch prediction, and speculative execution.



`lw x1, 0(x2)`																
`mul x3, x1, x4`																
`sw x3, 0(x5)`																
`addi x2, x2, 4`																
`addi x5, x5, 4`																
`addi x7, x7, -1`																
`bne x7, x0, loop`																
`opA`																
`lw x1, 0(x2)`																
`mul x3, x1, x4`																
`sw x3, 0(x5)`																
`addi x2, x2, 4`																
`addi x5, x5, 4`																
`addi x7, x7, -1`																
`bne x7, x0, loop`																
`opA`																

Superscalar Control Logic Scaling



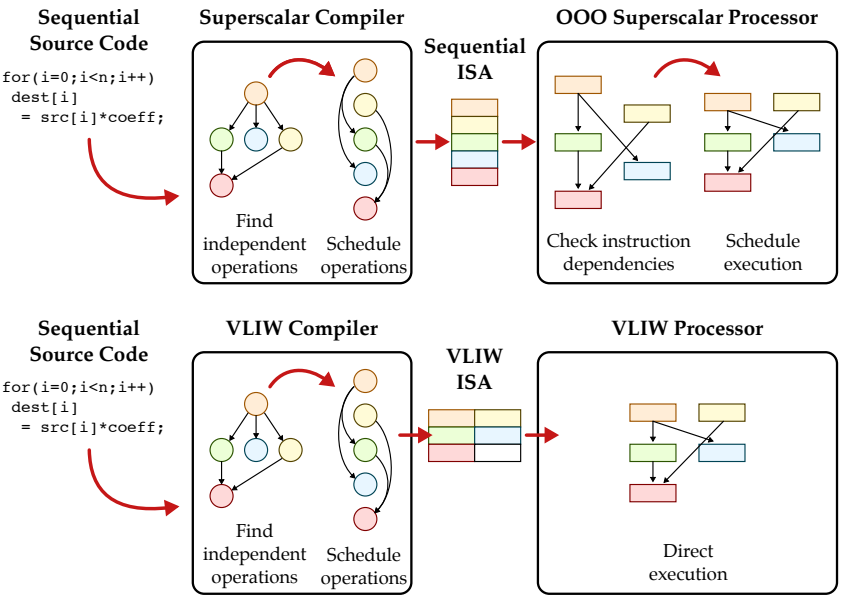


MIPS R10K Out-of-Order ~Quad-Issue Processor

- Each issued instruction must somehow check against $W \times L$ other instructions, i.e., hardware scaling $\propto W \times (W \times L)$
- For in-order issue
  - L is related to pipeline latencies
  - Check is done in the I stage (potentially using a scoreboard)
- For out-of-order issue
  - L is related to pipeline latencies and time spent waiting in the IQ/ROB
  - Check is done by broadcasting tags to waiting instructions at completion
  - As W increases, we need more instructions in-flight (i.e., waiting in IQ/ROB) to find enough ILP to keep functional units busy
  - Out-of-order control logic scales faster than W^2 (more like W^3)

VLIW = Very Long Instruction Word

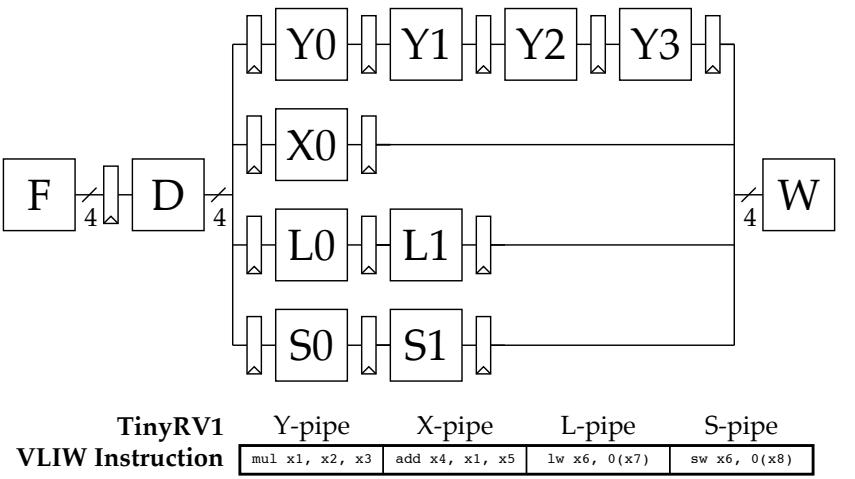
Key Idea: Replace a traditional sequential ISA with a new ISA that enables the compiler to encode instruction-level parallelism (ILP) directly in the hardware/software interface



- Multiple "sub-instructions" packed into one long instruction
- Each "slot" in a VLIW instruction for a specific functional unit
- Sub-instructions within a long instruction *must* be independent

- Compiler is responsible for avoiding all hazards!
  - Must use nops to avoid RAW hazards
  - Must use branch delay slots to avoid control hazards
  - Must use "slots" in VLIW instruction to avoid structural hazards
  - Must use extra architectural registers to avoid WAW/WAR name hazards

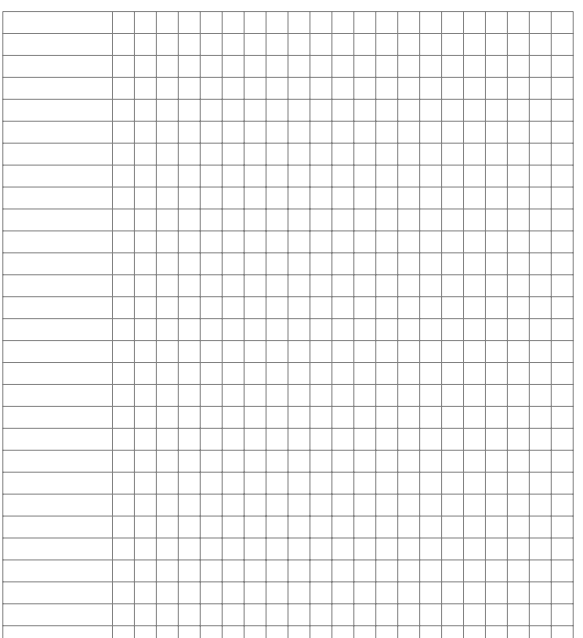
2. TinyRV1 VLIW Processor



- TinyRV1 VLIW ISA
  - Sub-instructions in VLIW instruction must be independent
  - 4-cycle Y-pipe, 1-cycle X-pipe, 2-cycle L-pipe, 2-cycle S-pipe
  - No hazard checking, assume ISA enables setting bypass muxing
  - Branch delay slot with two VLIW instructions
  - Early commit point in D
  - Stores go into memory system in S1

```
loop:
 lw   x1, 0(x2)
 mul  x3, x1, x4
 sw   x3, 0(x5)
 addi x2, x2, 4
 addi x5, x5, 4
 addi x7, x7, -1
 bne  x7, x0, loop
```

Y-pipe	X-pipe	L-pipe	S-pipe



3. VLIW Compilation Techniques

We will explore several compiler techniques that are critical for achieving high-performance on VLIW processors (note that some of these techniques can help improve performance on traditional processors too!):

- Loop unrolling
- Software pipelining
- Loop unrolling + software pipelining
- Other techniques: trace scheduling, instruction encoding

3.1. Loop Unrolling

```
int j = 0;
for ( int i = 0; i < n; i += 4 ) {
  dest[i+0] = src[i+0] * coeff;
  dest[i+1] = src[i+1] * coeff;
  dest[i+2] = src[i+2] * coeff;
  dest[i+3] = src[i+3] * coeff;
  j += 4;
}
for ( ; j < n; j++ )
  dest[j] = src[j] * coeff;
```

```
loop:
 lw   x1, 0(x2)
 lw   xA, 4(x2)
 lw   xB, 8(x2)
 lw   xC, c(x2)
 mul  x3, x1, x4
 mul  xD, xA, x4
 mul  xE, xB, x4
 mul  xF, xC, x4
 sw   x3, 0(x5)
 sw   xD, 4(x5)
 sw   xE, 8(x5)
 sw   xF, c(x5)
 addi x2, x2, 16
 addi x5, x5, 16
 addi x7, x7, -4
 bne  x7, x0, loop
```

- Loop unrolling amortizes loop overhead
- Loop unrolling requires many arch regs for sw renaming; increases static code size
- Need to carefully handle cases where `n` is not divisible by 4; add extra "fix-up code" after unrolled loop

```
loop:
 lw   x1, 0(x2)
 lw   xA, 4(x2)
 lw   xB, 8(x2)
 lw   xC, c(x2)
 mul  x3, x1, x4
 mul  xD, xA, x4
 mul  xE, xB, x4
 mul  xF, xC, x4
 sw   x3, 0(x5)
 sw   xD, 4(x5)
 sw   xE, 8(x5)
 sw   xF, c(x5)
 addi x2, x2, 16
 addi x5, x5, 16
 addi x7, x7, -4
 bne  x7, x0, loop
```

Y-pipe	X-pipe	L-pipe	S-pipe

Unroll by factor of eight?

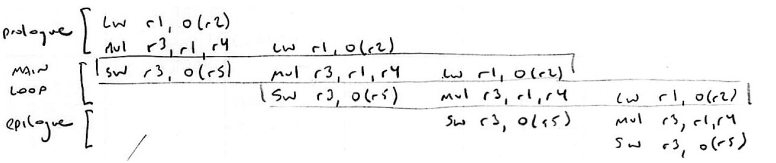
Y-pipe	X-pipe	L-pipe	S-pipe

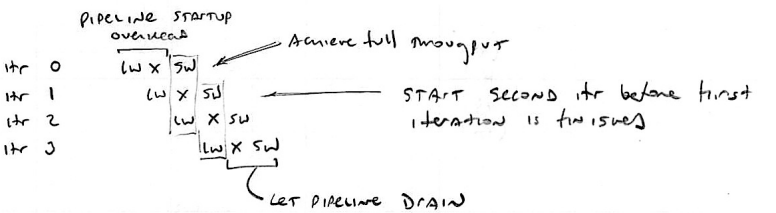
Calculate performance of VSMUL on the canonical TinyRV1 OOO quad-issue processor with register renaming, OOO memory disambiguation, perfect branch prediction, and speculative execution. Assume we unroll the loop by a factor of four.

`lw x1, 0(x2)`																
`lw xA, 4(x2)`																
`lw xB, 8(x2)`																
`lw xC, c(x2)`																
`mul x3, x1, x4`																
`mul xD, xA, x4`																
`mul xE, xB, x4`																
`mul xF, xC, x4`																
`sw x3, 0(x5)`																
`sw xD, 4(x5)`																
`sw xE, 8(x5)`																
`sw xF, c(x5)`																
`addi x2, x2, 16`																
`addi x5, x5, 16`																
`addi x7, x7, -4`																
`bne x7, x0, loop`																

3.2. Software Pipelining

Take instructions from multiple iterations to create new loop that can run at higher peak throughput.





- Start with original loop and focus on core computation
- Create "software" pipeline diagram
- Create prologue to fill the pipeline
- Create main loop body
- Create epilogue to drain the pipeline

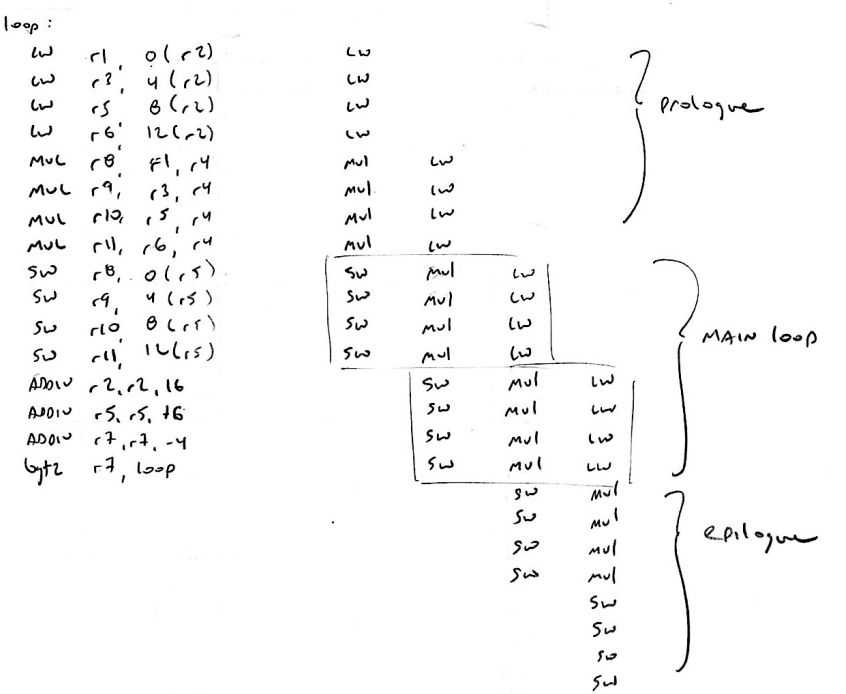
```
 lw   x1, 0(x2)
 mul  x3, x1, x4
 addi x2, x2, 4
 lw   x1, 0(x2)
 addi x2, x2, 4

loop:
 sw   x3, 0(x5)
 mul  x3, x1, x4
 lw   x1, 0(x2)
 addi x2, x2, 4
 addi x5, x5, 4
 addi x7, x7, -1
 bne  x7, x0, loop

 sw   x3, 0(x5)
 addi x5, x5, 4
 mul  x3, x1, x4
 sw   x3, 0(x5)
```

Y-pipe	X-pipe	L-pipe	S-pipe

- Software Pipelining vs Loop Unrolling
  - Produces more compact code
  - Uses less registers
  - Can better handle irregularly sized input arrays
  - Quickly get up to peak throughput, one epilogue/prologue per loop
  - Software pipelining does *not* reduce loop overhead

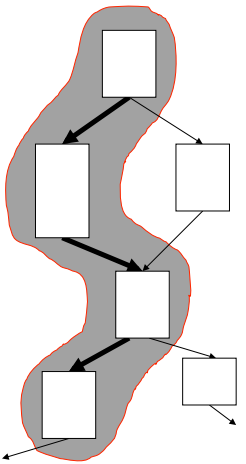
3.3. Loop Unrolling and Software Pipelining

- Use loop unrolling to amortize loop overhead
- Use software pipelining to quickly reach full throughput (and also reduce code size, register pressure)



Y-pipe	X-pipe	L-pipe	S-pipe

3.4. Other Compiler Techniques



- Problem: What if there are no loops?
  - Branches limit basic block size in control-flow intensive irregular code
  - Only one branch per VLIW instruction
  - Difficult to find ILP in individual basic blocks
- Trace scheduling
  - Use predication to create larger basic blocks
  - Use profiling feedback or compiler heuristics to find common branch paths
  - Pick trace of basic blocks along common path
  - Schedule whole trace at once
  - Add "fix-up code" to cope with branches jumping out of trace

VLIW Instruction Encoding

- Problem: VLIW encodings require many NOPs which waste space
- Compressed format in memory, expand on instruction cache refill
- Provide a single-op VLIW instruction
- Create variable sized "instruction bundles"

Memory Latency Register (MLR)

- Problem: Loads have variable latency
- Compiler schedules code for maximum load-use distance
- Compiler sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late