Computer Science 146
Computer Architecture

Fall 2019

Harvard University

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

Lecture 3: CISC/RISC, Multimedia ISA, Implementation Review

Lecture Outline

- CISC vs. RISC
- Multimedia ISAs
  - Review of the PA-RISC, MAX-2
  - Examples
- Compiler Interactions
- Implementation Review

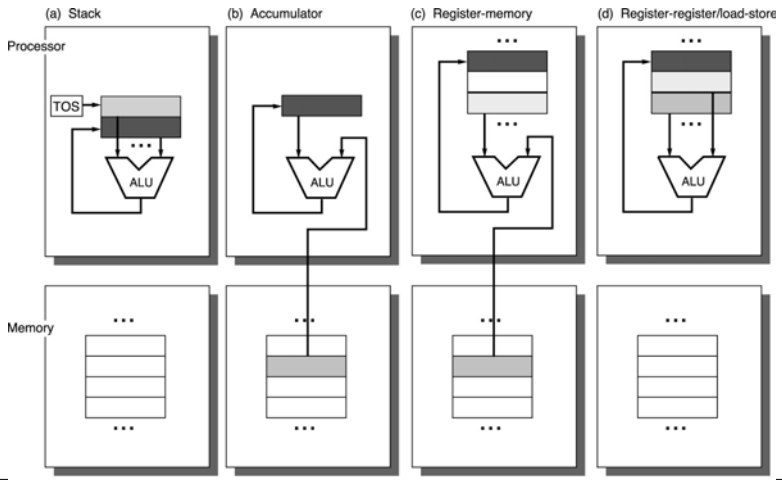
Instruction Set Architecture

"Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine."

IBM, Introducing the IBM 360 (1964)

- The ISA defines:
  - Operations that the processor can execute
  - Data Transfer mechanisms + how to access data
  - Control Mechanisms (branch, jump, etc)
  - "Contract" between programmer/compiler + HW

Classifying ISAs



Stack

- Architectures with implicit "stack"
  - Acts as source(s) and/or destination, TOS is implicit
  - Push and Pop operations have 1 explicit operand
- Example: C = A + B
  - Push A // S[++TOS] = Mem[A]
  - Push B // S[++TOS] = Mem[B]
  - Add // Tem1 = S[TOS--], Tem2 = S[TOS--] , S[++TOS] = Tem1 + Tem2
  - Pop C // Mem[C] = S[TOS--]
- x86 FP uses stack (complicates pipelining)

Accumulator

- Architectures with one implicit register
  - Acts as source and/or destination
  - One other source explicit
- Example: C = A + B
  - Load A // (Acc)umulator <= A
  - Add B // Acc <= Acc + B
  - Store C // C <= Acc
- Accumulator implicit, bottleneck?
- x86 uses accumulator concepts for integer

Register

- Most common approach
  - Fast, temporary storage (small)
  - Explicit operands (register IDs)
- Example: C = A + B

Register-memory	load/store
Load R1, A	Load R1, A
Add R3, R1, B	Load R2, B
Store R3, C	Add R3, R1, R2
	Store R3, C

- All RISC ISAs are load/store
- IBM 360, Intel x86, Moto 68K are register-memory

Common Addressing Modes

Base/Displacement	Load R4, 100(R1)
Register Indirect	Load R4, (R1)
Indexed	Load R4, (R1+R2)
Direct	Load R4, (1001)
Memory Indirect	Load R4, @(R3)
Autoincrement	Load R4, (R2)+
Scaled	Load R4, 100(R2)[R3]

What leads to a good/bad ISA?

- Ease of Implementation (Job of Architect/Designer)
  - Does the ISA lead itself to efficient implementations?
- Ease of Programming (Job of Programmer/Compiler)
  - Can the compiler use the ISA effectively?
- Future Compatibility
  - ISAs may last 30+yrs
  - Special Features, Address range, etc. need to be thought out

Implementation Concerns

- Simple Decoding (fixed length)
- Compactness (variable length)
- Simple Instructions (no load/update)
  - Things that get microcoded these days
  - Deterministic Latencies are key!
  - Instructions with multiple exceptions are difficult
- More/Less registers?
  - Slower register files, decoding, better compilers
- Condition codes/Flags (scheduling!)

Programmability

- 1960s, early 70s
  - Code was mostly hand-coded
- Late 70s, Early 80s
  - Most code was compiled, but hand-coded was better
- Mid-80s to Present
  - Most code is compiled and almost as good as assembly
- Why?

Programmability: 70s, Early 80s "Closing the Semantic Gap"

- High-level languages match assembly languages
- Efforts for computers to execute HLL directly
  - e.g. LISP Machine
    - Hardware Type Checking. Special type bits let the type be checked efficiently at run-time
    - Hardware Garbage Collection
    - Fast Function Calls
    - Efficient Representation of Lists
- Never worked out…"Semantic Clash"
  - Too many HLLs? C was more popular?
  - Is this coming back with Java? (Sun's picoJava)

Programmability: 1980s … 2000s "In the Compiler We Trust"

- Wulf: Primitives not Solutions
  - Compilers cannot effectively use complex instructions
  - Synthesize programs from primitives
- Regularity: same behavior in all contexts
  - No odd cases – things should be intuitive
- Orthogonality:
  - Data type independent of addressing mode
  - Addressing mode independent of operation performed

ISA Compatibility

"In Computer Architecture, no good idea ever goes unpunished."
Marty Hopkins, IBM Fellow

- Never abandon existing code base
- Extremely difficult to introduce a new ISA
  - Alpha failed, IA64 is struggling, best solution may not win
- x86 most popular, is the least liked!
- Hard to think ahead, but…
  - ISA tweak may buy 5-10% today
  - 10 years later it may buy nothing, but must be implemented
    - Register windows, delay branches

CISC vs. RISC

- Debate raged from early 80s through 90s
- Now it is fairly irrelevant
- Despite this Intel (x86 => Itanium) and DEC/Compaq (VAX => Alpha) have tried to switch
- Research in the late 70s/early 80s led to RISC
  - IBM 801 -- John Cocke – mid 70s
  - Berkeley RISC-1 (Patterson)
  - Stanford MIPS (Hennessy)

VAX

- 32-bit ISA, instructions could be huge (up to 321 bytes), 16 GPRs
- Operated on data types from 8 to 128-bits, decimals, strings
- Orthogonal, memory-to-memory, all operand modes supported
- Hundreds of special instructions
- Simple compiler, hand-coding was common
- CPI was over 10!

x86

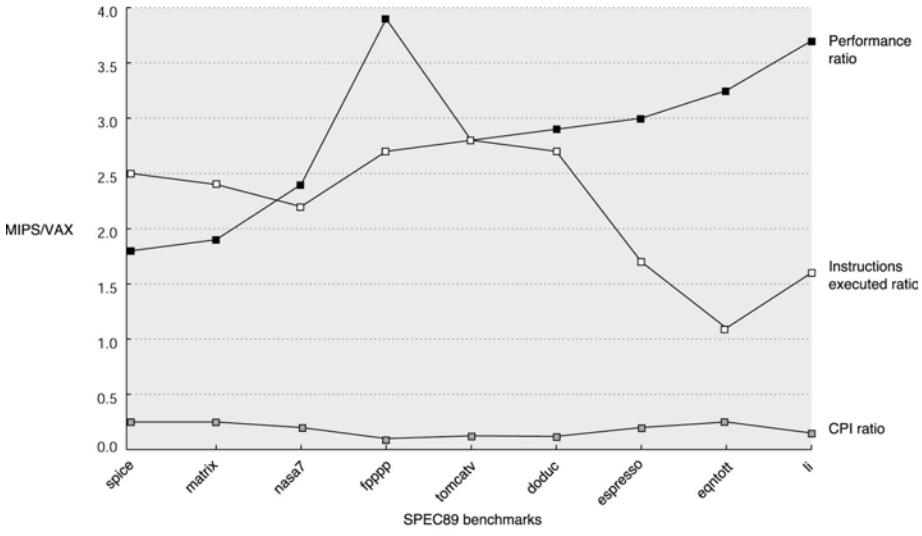
- Variable length ISA (1-16 bytes)
- FP Operand Stack
- 2 operand instructions (extended accumulator)
  - Register-register and register-memory support
- Scaled addressing modes

- Has been extended many times (as AMD has recently done with x86-64)
- Intel, instead (?) went to IA64

RISC vs. CISC Arguments

- RISC
  - Simple Implementation
    - Load/store, fixed-format 32-bit instructions, efficient pipelines
  - Lower CPI
  - Compilers do a lot of the hard work
    - MIPS = Microprocessor without Interlocked Pipelined Stages
- CISC
  - Simple Compilers (assists hand-coding, many addressing modes, many instructions)
  - Code Density

MIPS/VAX Comparison



After the dust settled

- Turns out it doesn't matter much
- Can decode CISC instructions into internal "micro-ISA"
  - This takes a couple of extra cycles (PLA implementation) and a few hundred thousand transistors
  - In 20 stage pipelines, 55M tx processors this is minimal
  - Pentium 4 caches these micro-Ops
- Actually may have some advantages
  - External ISA for compatibility, internal ISA can be tweaked each generation (Transmeta)

Multimedia ISAs

- Motivation
  - Human perception does not need 64-bit precision
  - Single-instruction, Multiple-data (SIMD) parallelism
- Initially introduced in workstations
  - HP MAX-1 (’94), MAX-2 (’96)
  - SPARC VIS-1 (’95)
- Quickly migrated to desktops/laptops
  - Intel MMX (’97), SSE (’99), SSE2 (’00), SSE3 (’04)
- Future will focus on security ISAs

Apps suitable to MM-ISAs

- Tons of parallelism
  - Ideally, parallelism exists at many levels
    - Frame, color components, blocks, pixels, etc
- Low precision data available
  - 8-bits per color component per pixel (RGB)
  - Sometimes 12-bits (medical apps)
- Computationally intensive apps
  - Lots of adds, subtracts, shift-and-add, etc
- Examples: MPEG encode/decode, jpeg, mp3

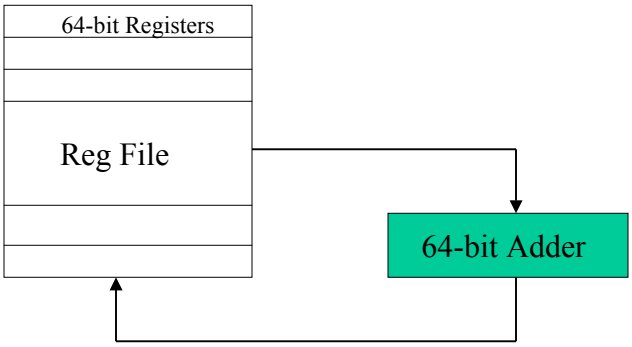
Subword Parallelism Techniques

- Loop vectorization
  - Multiple iterations can be performed in parallel
- Parallel accumulation
- Saturating arithmetic
  - In-line Conditional Execution!
- Data rearrangment
  - Critical for matrix transpose
- Multiplication by constants

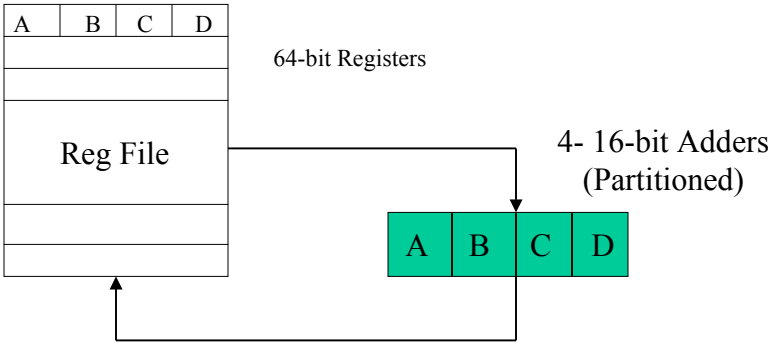
Types of Ops

- Parallel Add/Subtract
  - Modulo Arithmetic, Signed/Unsigned Saturating
- Parallel Shift-and-Add
  - Shift Left and Right
  - Equivalent to Multiply-by-Constant
- Parallel Average
- Mix, Permute
  - Subword Rearrangment
- MADD, Max/Min, SAD

Simple implementation



Simple implementation

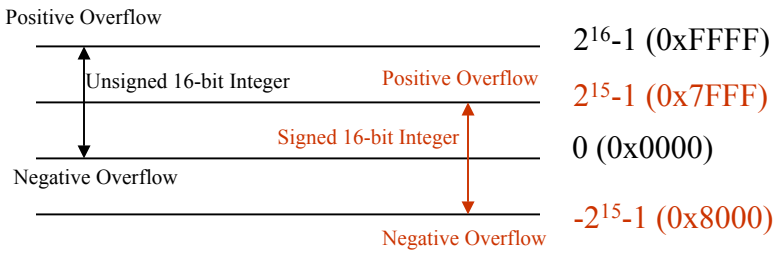


- Could also be 8x8-bit, 2x32-bit
- How would we do shift-and-add?

Overflow?

- Ignore Overflow (Modulo arithmetic)
- Set flags
- Throw an exception
- Clamp results to max/min values (Saturating arithmetic)
- Some ops never overflow (PAVG)

Saturating Arithmetic



- How would this be implemented?

In-line Conditional Execution

- Saturating arithmetic allows the following:

  If cond(Ra_i,Rb_i) Then Rt_i=Ra_i else Rt_i=Rb_i

For i=number of subwords in the word

- Takes advantage of the fact that saturating arithmetic is not commutative
  - ie. (+k)-k not same as +k(-k)
  - (0 + 20) + (0 – 20) = 0 (Standard Arithmetic)
  - (0 + 20) + (0 – 20) = 20 (With Unsigned Saturating Arith.)

Finding min(Ra,Rb) with Saturating Arithmetic

- Example: Finding min(Ra, Rb)

If Ra_i>Rb_i Then Rt_i=Rb_i else Rt_i=Ra_i

Ra=	260	60	260	60
Rb=	60	260	-60	-260
HSUB,us Ra, Rb, Rt	200	0	320	320
HSUB,ss R0, Rt, Rt	-200	0	-320	-320
HADD,ss Rt, Ra, Rt	60	60	-60	-260

- max(Ra, Rb), abs(Ra), SAD(Ra,Rb) are easy too

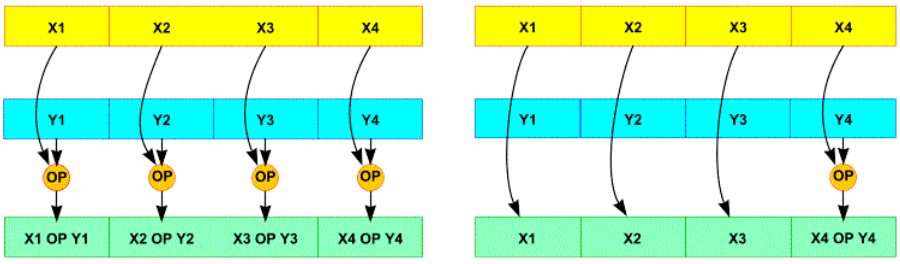
Speedups on Kernels on PA-8000 (with and without MAX2)

Programs vs. Metrics	16x16 Block Match	8x8 Matrix Transpose	3x3 Box Filter	8x8 IDCT
Instructions	420 (1307)	32 (84)	1107 (5320)	380 (1574)
Cycles	160 (426)	16 (42)	548 (2234)	173 (716)
Registers	14 (12)	18 (22)	15 (18)	17 (20)
Cycles/Element	0.63 (1.66)	0.25 (0.66)	2.80 (11.86)	2.70 (11.18)
Instructions/Cycle	2.63 (3.07)	2.00 (2.00)	2.02 (2.29)	2.20 (2.20)
Speedup	**2.66**	**2.63**	**4.24**	**4.14**

What is Intel's SSE?

- An extension of Intel's MMX (similar to MAX)
- Streaming SIMD Extensions
  - 8 new 128-bit SIMD Floating Point registers
  - PIII: SSE -> 50 new ops
    - ADD, SUB, MUL, DIV, SQRT, MAX, MIN
  - P4: SSE2 -> MMX -> 128bits, SSE-> 64-bit FP
  - Prescott New Instructions: SSE3 -> 13 new instructions (data movement, conversion, "horizontal addition")

Packed vs. Scalar



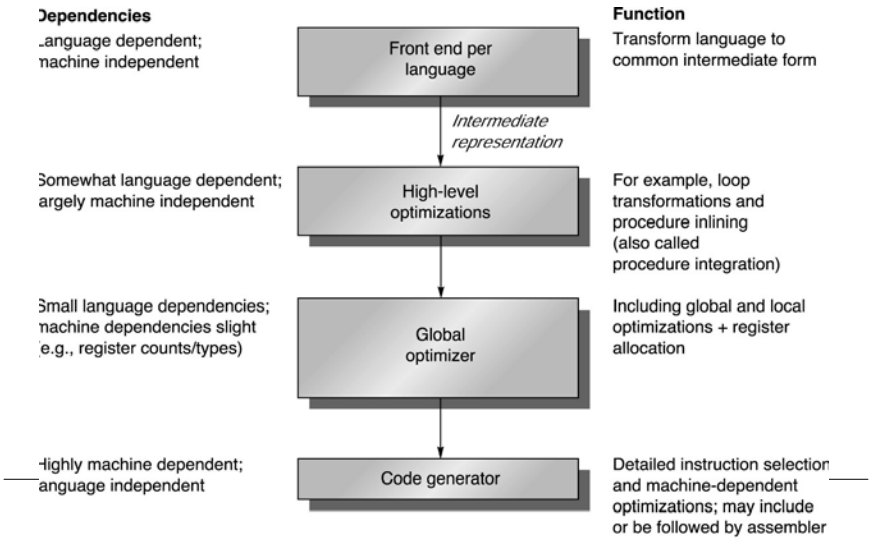
SSE Packed

SSE Scalar

SSE3: Horizontal Add

- HADDPS OpA OpB
  - OpA (128bits, 4 elements): 3_a, 2_a, 1_a, 0_a
  - OpB (128bits, 4 elements): 3_b, 2_b, 1_b, 0_b
  - Result (in OpA): $3_b + 2_b$, $1_b + 0_b$,$3_a + 2_a$, $1_a + 0_a$

Compilers 101



Compiler Optimizations

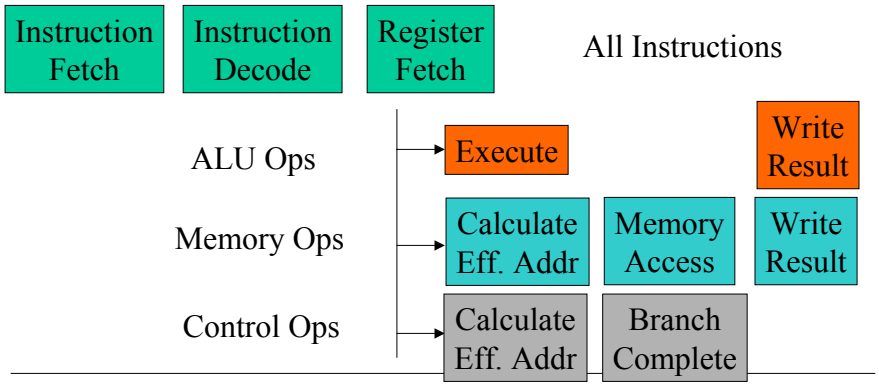
- High-level optimizations
  - Done on source, may be source-to-source conversions
  - Examples – map data for cache efficiency, remove conditions, etc.
- Local Optimizations
  - Optimize code in small straight-line sections
- Global Optimizations
  - Extend local opts across branches and do loop optimizations (loop unrolling)
- Register Allocation
  - Assign temporary values to registers, insert spill code

Compiler support for MM ISAs

- Actually there is very little
- Surprising because vector-computers have good compiler support
- Problems
  - Short, architecture-limited vectors
  - Few registers and simple addressing modes
  - Most programming languages don't support subwords
  - Kernels tend to be handcoded

Implementation Review

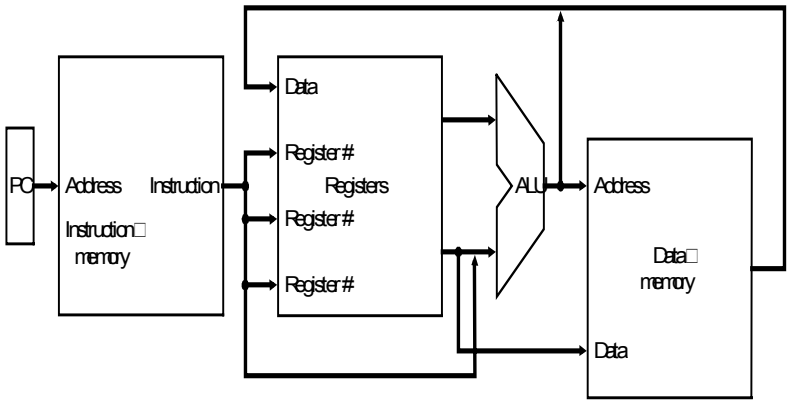
- First, let's think about how different instructions get executed



Instruction Fetch

- Send the Program Counter (PC) to memory
- Fetch the current instruction from memory
  - IR <= Mem[PC]
- Update the PC to the next sequential
  - PC <= PC + 4 (4-bytes per instruction)
- Optimizations
  - Instruction Caches, Instruction Prefetch
- Performance Affected by
  - Code density, Instruction size variability (CISC/RISC)

Abstract Implementation

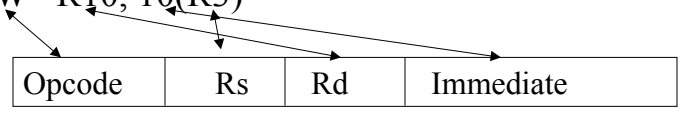


Instruction Decode/Reg Fetch

- Decide what type of instruction we have
  - ALU, Branch, Memory
  - Decode Opcode
- Get operands from Reg File
  - A <= Regs[$IR_{25..21}$]; B <= Regs[$IR_{20..16}$];
  - Imm <= SignExtend($IR_{15..0}$)
- Performance Affected by
  - Regularity in instruction format, instruction length

Calculate Effective Address: Memory Ops

- Calculate Memory address for data
- ALU_{output} <= A + Imm
- LW R10, 10(R3)



Calculate Effective Address: Branch/Jump Ops

- Calculate target for branch/jump operation
- BEQZ, BNEZ, J
  - ALU_{output} <= NPC + Imm; cond <= A op 0
  - "op" is a check against 0, equal, not-equal, etc.
  - J is an unconditional
- ALU_{output} <= A

Execution: ALU Ops

- Perform the computation
- Register-Register
  - ALU_{output} <= A op B
- Register-Immediate
  - ALU_{output} <= A op Imm
- No ops need to do effective address calc *and* perform an operation on data
- Why?

Memory Access

- Take effective address, perform Load or Store
- Load
  - LMD <= Mem[ALU_{output}]
- Store
  - Mem[ALU_{output}] <= B

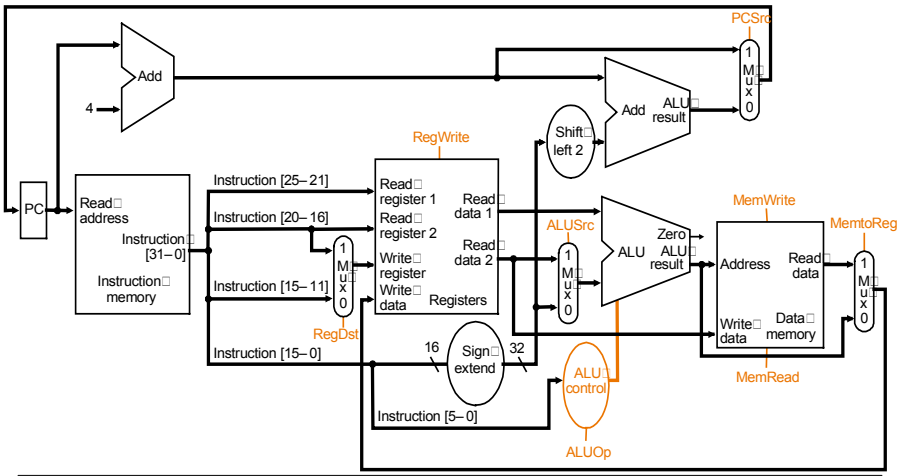
Mem Phase on Branches

- Set PC to the calculated effective address
- BEQZ, BNEZ
  - If (cond) PC <= ALU_{output} else PC <= NPC

Write-Back

- Send results back to register file
- Register-register ALU instructions
  - $Regs[IR_{15..11}] <= ALU_{output}$
- Register-Immediate ALU instruction
  - $Regs[IR_{20..16}] <= ALU_{output}$
- Load Instruction
  - $Regs[IR_{20..16}] <= LMD$
- Why does this have to be a separate step?

Final Implementation



For next time

- Implementation Review
- Pipelining
  - Start to read Appendix A or review previous textbook