CS152: Computer Systems Architecture
Some Loose Microarchitecture Topics

Sang-Woo Jun

Winter 2021

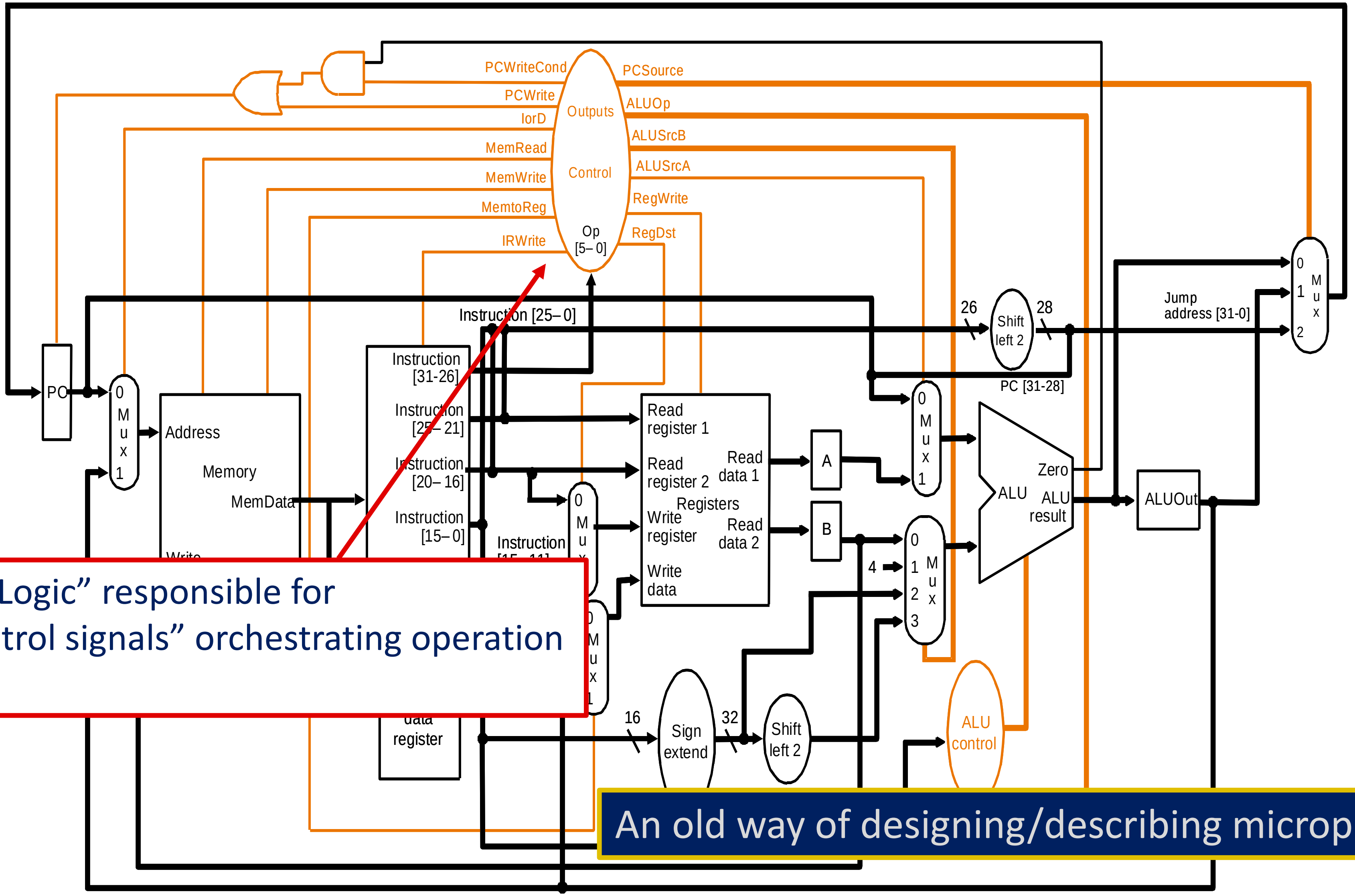
UCI

Large amount of material adapted from MIT 6.004, "Computation Structures",
Morgan Kaufmann "Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition",
and CS 152 Slides by Isaac Scherson

Some loose topics
(Before getting started on memory systems)

- Microprogramming
    - Now seems to mean a combination of two different things!

Microprogramming of old

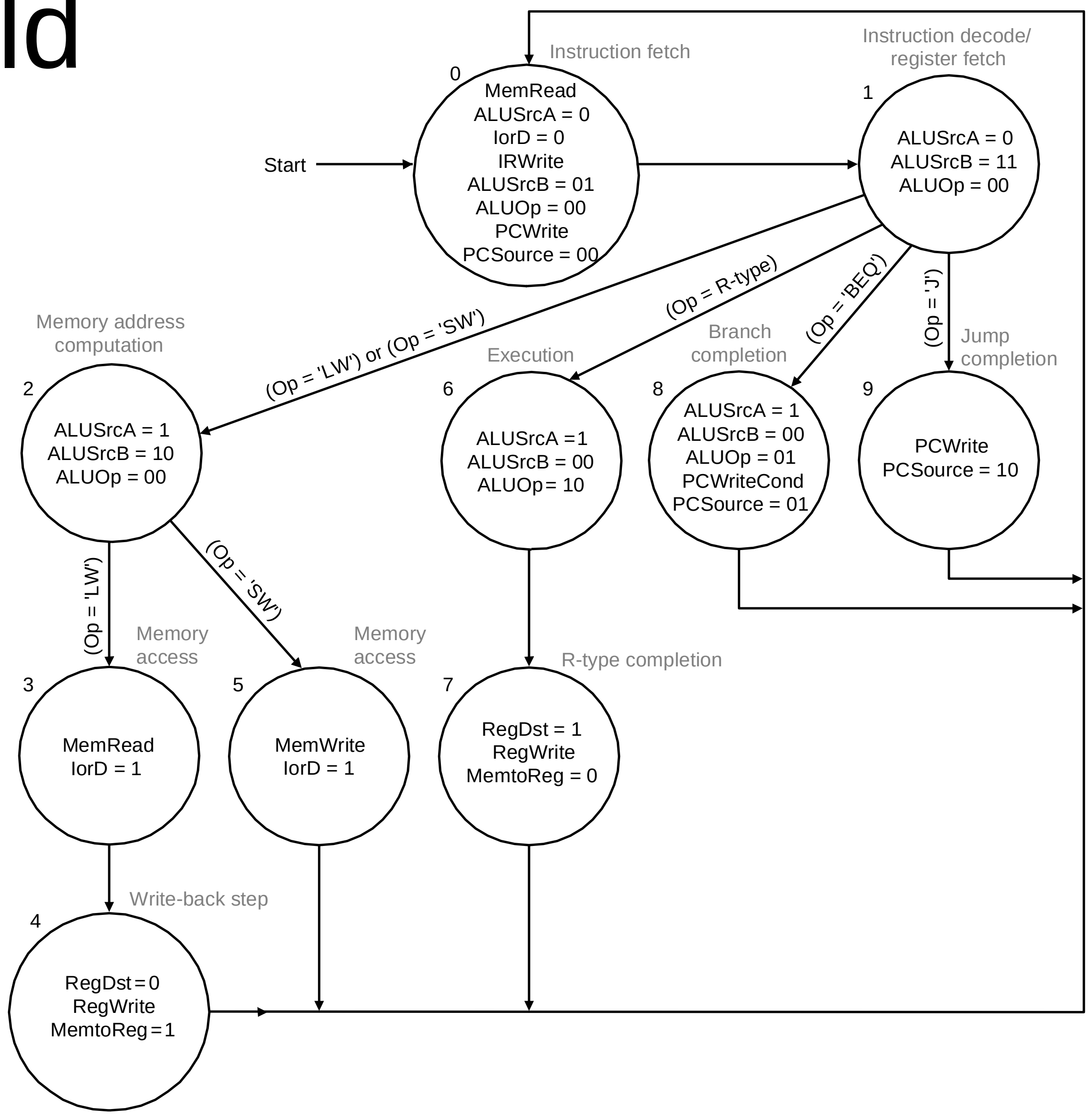


Single “Control Logic” responsible for generating “control signals” orchestrating operation at each cycle

An old way of designing/describing microprocessor operation

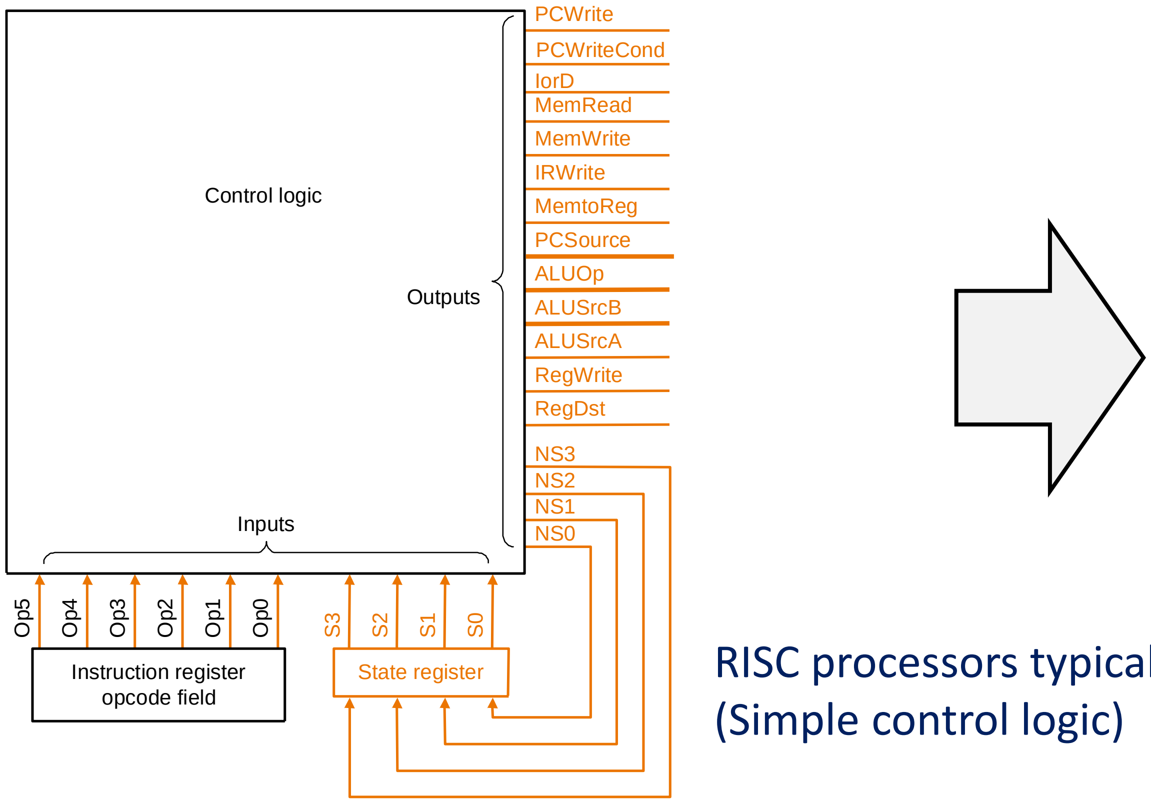
Microprogramming of old

- Control logic operation described as a Finite State Machine (FSM)
  - Next state depends on current state, and input to the control logic
  - Control signal output depends on current state of the FSM

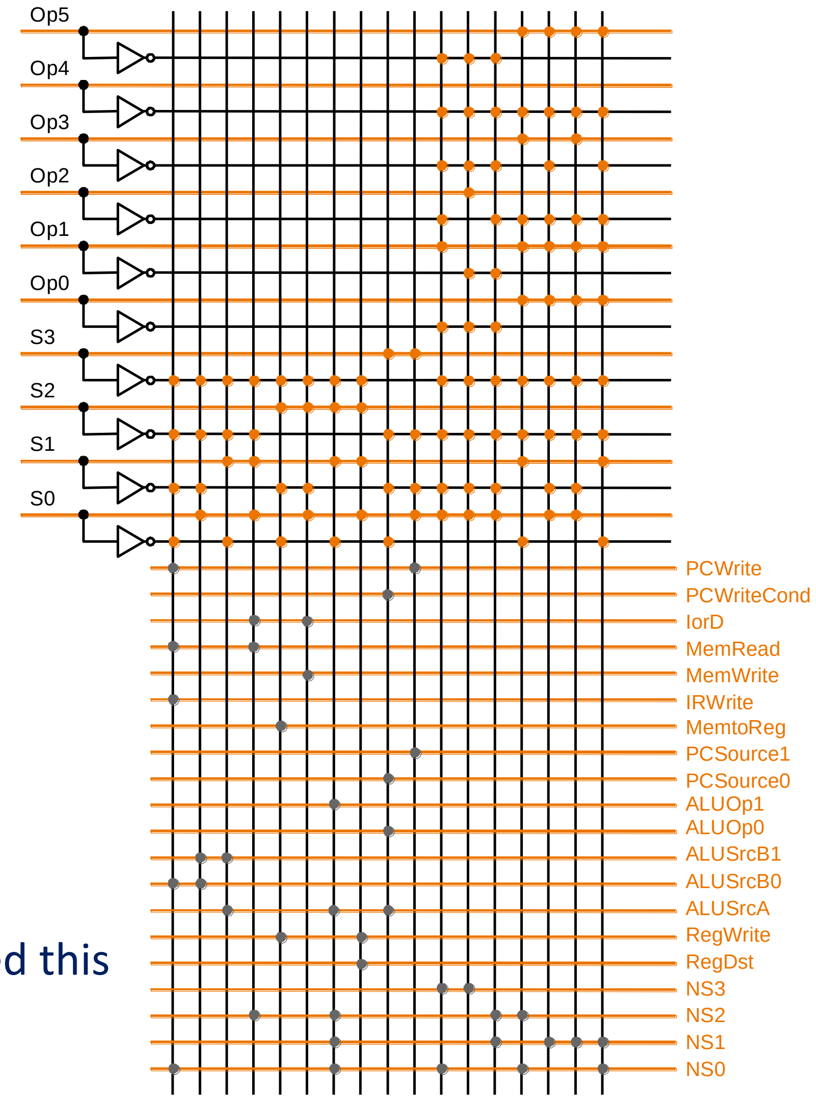


Microprogramming of old

- Control logic FSM implemented via ROM or PLA
  - "Microprogramming"



Programmable Logic Array (PLA)



RISC processors typically don't need this
(Simple control logic)

Aside: Microcode and bug patches

- Modern CPUs have programmable portion of the microcode storage
  - No longer entirely ROM
  - Programmable portion takes precedence over original microcode
  - Makes live bug patches possible!
  - Implement same x86 instruction using a different (“bug free”) sequence of microcode operations
- For example, CPU patches for the infamous Spectre exploit involved microcode patches
  - When “BIOS updates” are required, this is often what’s happening

Microprogramming of new: CISC and x86

- x86 ISA is CISC ("Complex")

Hex	Mnemonics
C3	ret
48 b8 88 77 66 55 44 33 22 11	movabs rax,0x1122334455667788
64 ff 03	DWORD PTR fs:[ebx]
64 67 66 f0 ff 07	lock inc WORD PTR fs:[bx]
2e c4 e2 71 96 84 be 34 23 12 01	vfmaddsub132ps xmm0, xmm1, xmmword ptr cs: [esi + edi * 4 + 0x11223344]

Philipp Koppe et.al., "Reverse Engineering x86 Processor Microcode," USENIX security 2017

Microprogramming of new: CISC and x86

- Modern microarchitectural advances are difficult to get right on CISC architectures
  - Superscalar, Out-of-Order, Transactional memory, etc
  - Too many conditions and states to keep track of!
- Instead, modern CISC processors internally implement a RISC core with modern bells and whistles
  - e.g., AMD’s patented RISC86 ISA
  - “Front-end” x86 ISA translated by CPU hardware on-the-fly to RISC instructions

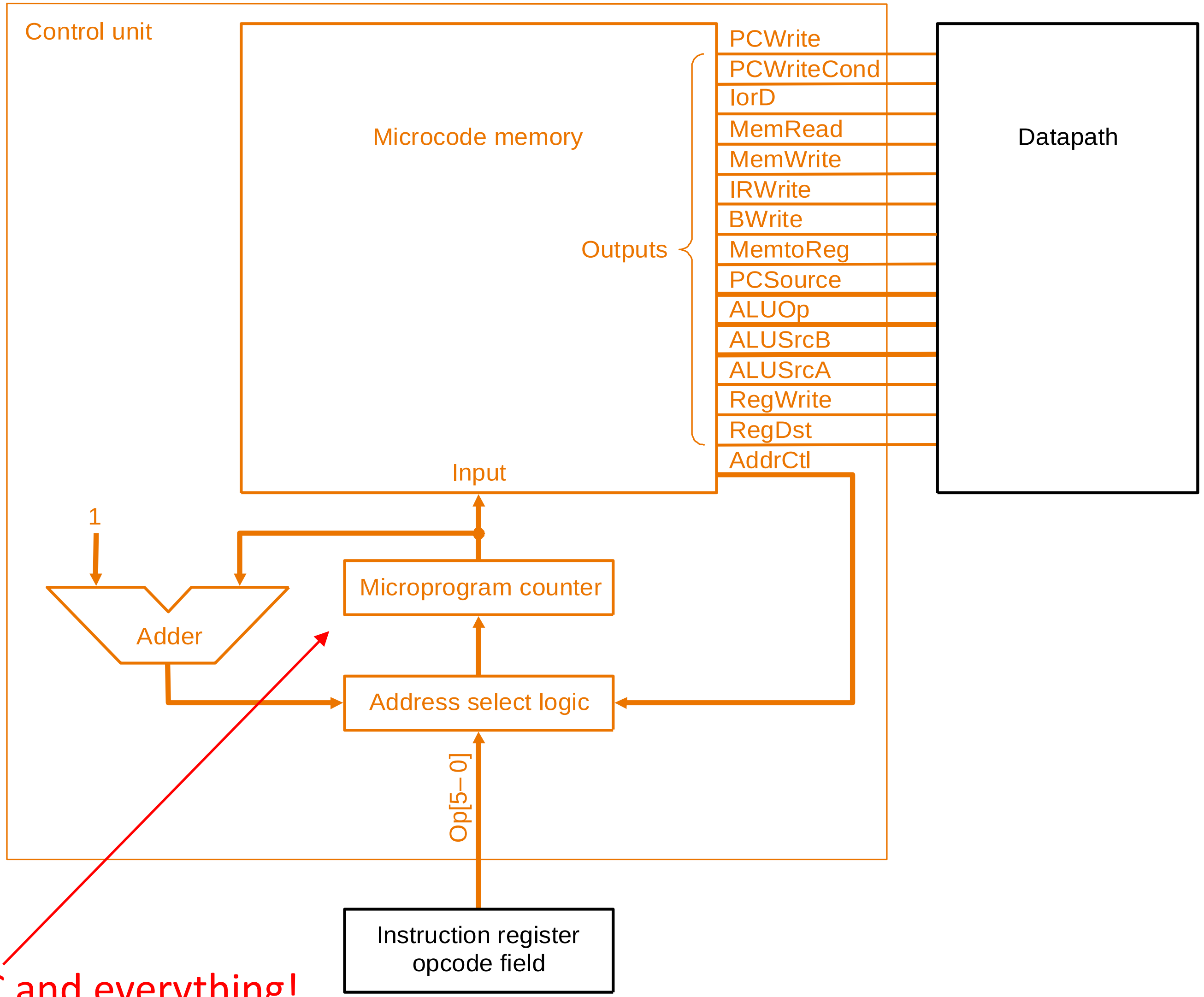
pop [ebx] →

load temp , [esp]
store [ebx] , temp
add esp , 4

Philipp Koppe et.al., “Reverse Engineering x86 Processor Microcode,” USENIX security 2017

Microprogramming complex instructions

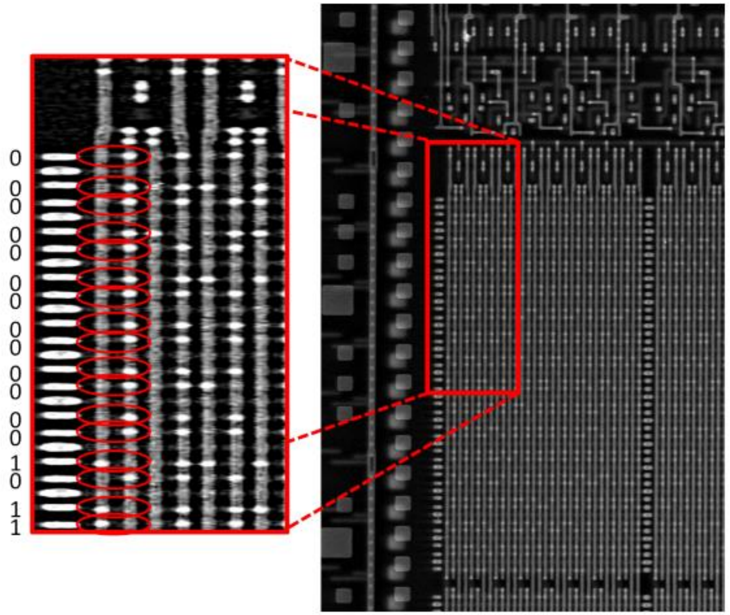
- There is typically a fixed sequence of control signals/RISC instructions to generate for one CISC instruction
  - Decoder is programmed with a "program" for generating them
- This is not exclusive to CISC-RISC translation. Idea is old!



Microcode decoder is like a small CPU, with PC and everything!

Microprogramming of new

- ❑ Microprogramming can be used to generate a sequence of control signals per input instruction
  - o Implemented via a chain of FSM states in the control logic
  - o No longer designed manually though! Lots of tool research into efficient microcode compilation
  - o Usually multiple “decoders” operating in parallel
- ❑ We know traditional techniques are still used



Philipp Koppe et.al., “Reverse Engineering x86 Processor Microcode,” USENIX security 2017

Some loose topics
(Before getting started on memory systems)

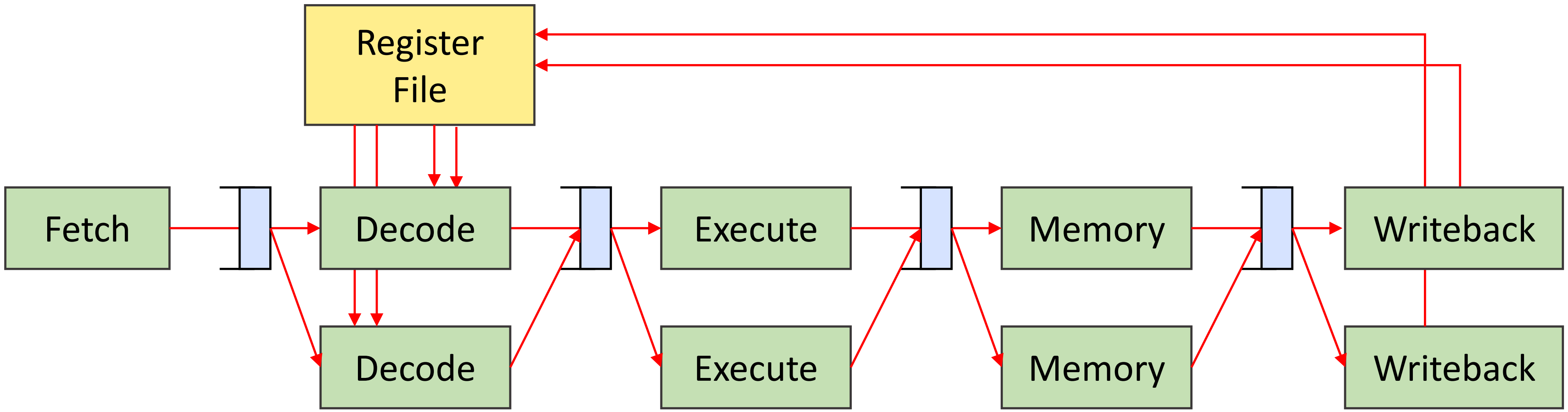
- Superscalar
  - Just a taste!

Superscalar Processing

- An ideally pipelined processor can handle up to one instructions per cycle
  - Instructions Per Cycle (IPC) = 1, Cycles Per Instruction (CPI) = 1
- Superscalar wants to process multiple instruction per cycle
  - IPC > 1, CPI <1
  - An N-way superscalar processor handles N instructions per cycle
  - Requires multiple pipeline hardware instances/resources
  - Hardware performs dependency checking on-the-fly between concurrently-fetched instructions

Pipeline for superscalar processing

- Multiple copies of the datapath supports multiple instructions/cycle
- Register file needs many more ports
- Actually requires a complex scheduler in the decode stage!



Superscalar has concurrent hazards

- What if two concurrently issued instructions have dependencies?
  - No choice but to stall the dependent instruction…
  - … in an in-order pipeline! ← Topic for another day
- Data hazards
  - e.g., “addi s1, s0, 1” and “addi s2, s1, 1” issued at the same time?
    - Forwarding won’t work here! Both instructions in decode stage at the same time
    - Scheduler must stagger “addi s2, s1, 1”, sacrificing performance
- Control hazards
  - e.g., How to handle a beq, followed by another instruction?
    - Branch prediction, as usual

Results in very complex control logic! (Chip resources/cost!)

In-order superscalar example

Ideal IPC = 2 (2-Way superscalar)

lw t0, 40($s0)

add t1, $s1, $s2

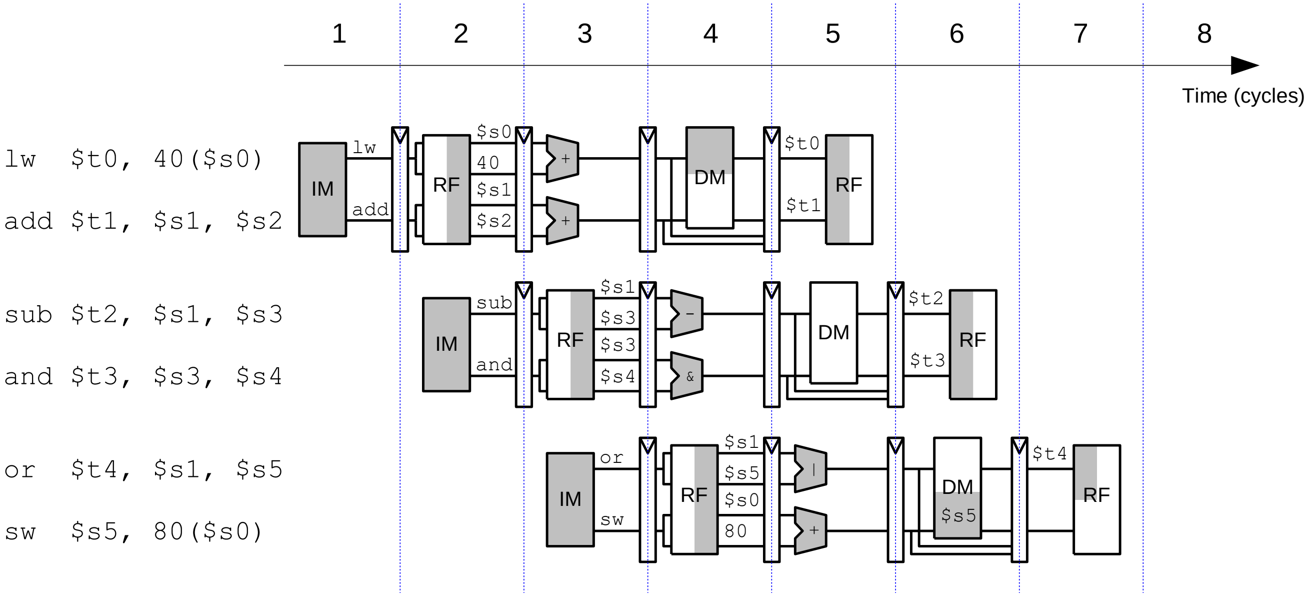
sub t2, s1, s3

and t3, s3, s4

or t4, s1, s5

sw s5, 80(s0)

No dependencies between any instructions



Actual IPC = 2 (6 instructions issued in 3 cycles)

Source: Onur Mutlu, “Design of Digital Circuits,” Lecture 16, 2019

In-order superscalar with dependencies

Ideal IPC = 2 (2-Way superscalar)

lw t0, 40(s0)

add t1, t0,$s1

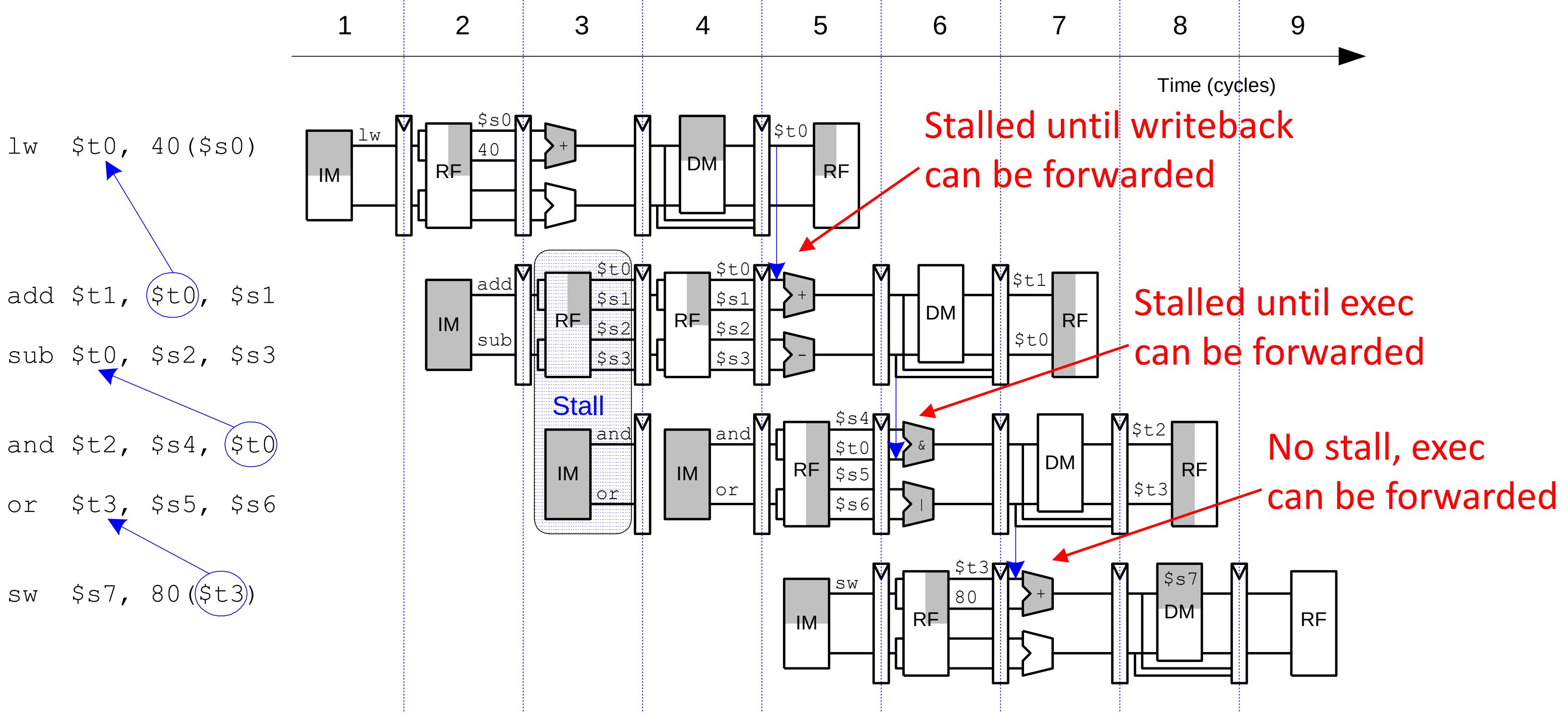
sub t0, s2, s3

and t2, s4, t0

or t3, s5, s6

sw s7, 80(t3)

Dependencies across many instructions!

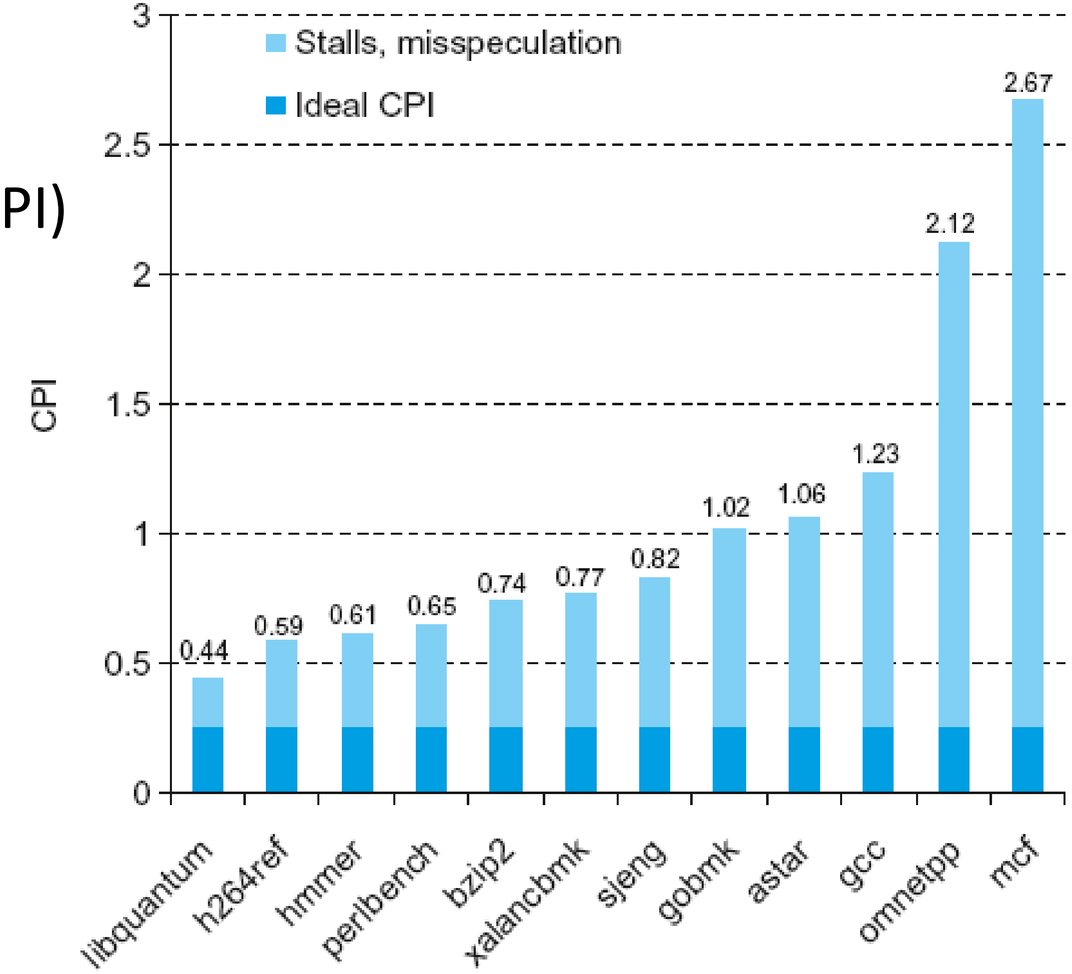


Actual IPC = 1.2 (6 instructions issued in 5 cycles)

Source: Onur Mutlu, "Design of Digital Circuits," Lecture 16, 2019

In the real-world: Core i7 performance

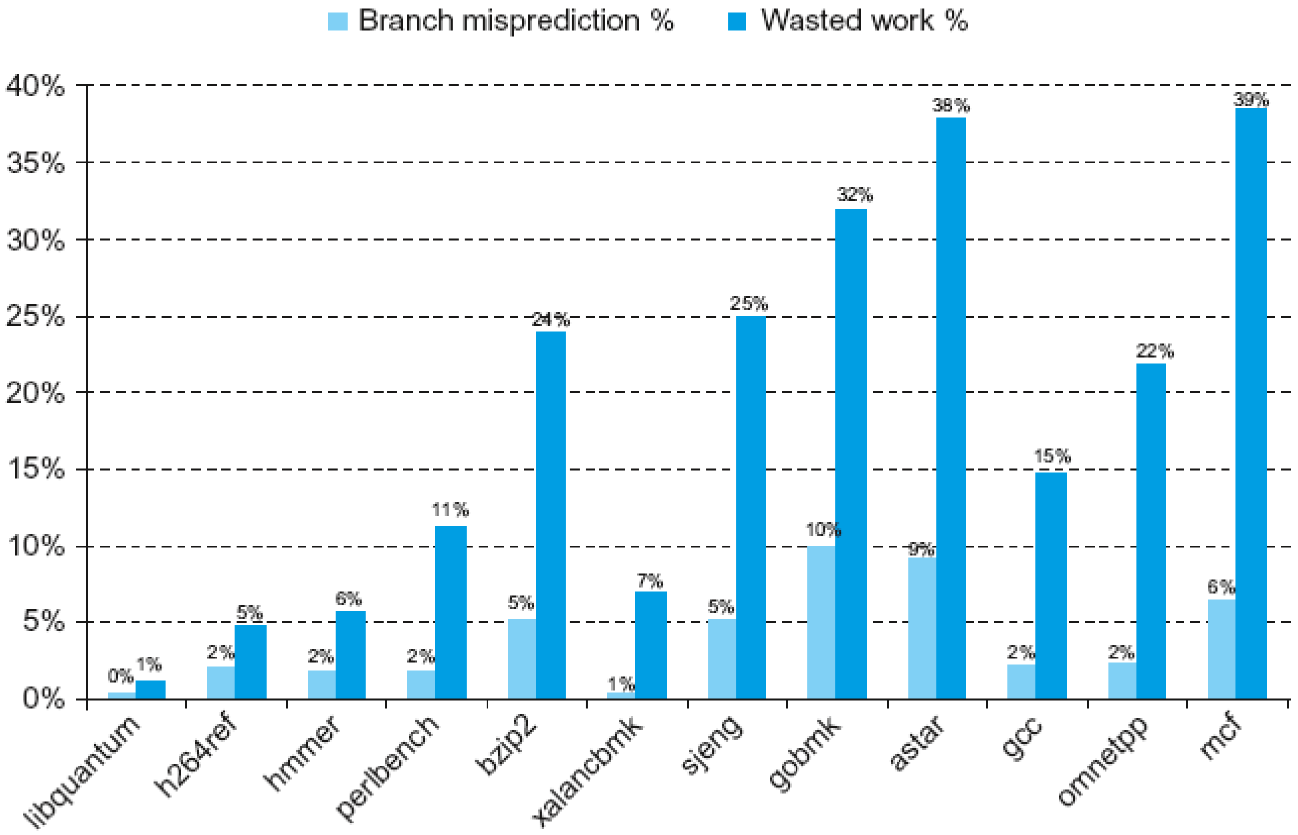
- Core i7 has a 4-way *Out-of-Order* Superscalar pipeline
  - Ideally, 0.25 Cycles Per Instruction (CPI)
  - Dependencies and misprediction typically results in much lower performance



Is it worth it? Or do we want just more, simpler cores? Depends on your target area (servers? phones?) and profiling results…

In the real-world: Core i7 performance

- Branch predictors work pretty well!
  - But deep/wide pipelines result in high mispredict overhead



Some loose topics
(Before getting started on memory systems)

- Hardware Performance Counters
  - Small number of special-purpose registers (few dozens in modern x86)
  - User-configured to count hardware activities
  - E.g., number of issued instructions, cache misses, branch mis-predicts, etc
  - Important for performance profiling! (And some security attacks)
- Easiest is to use utility “perf”

```
[redacted]@[redacted]:~$ sudo perf stat sleep 1
[sudo] password for [redacted]:

 Performance counter stats for 'sleep 1':

          1.293090      task-clock (msec)         #    0.001 CPUs utilized
                 1      context-switches          #    0.773 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                60      page-faults               #    0.046 M/sec
         1,024,993      cycles                    #    0.793 GHz
           841,073      instructions              #    0.82  insn per cycle
           163,636      branches                  #  126.546 M/sec
             7,572      branch-misses             #    4.63% of all branches

       1.002117785 seconds time elapsed
```

Some loose topics
(Before getting started on memory systems)

- Macro-op fusion
  - Multiple instructions can be “fused” into a larger one
    - Two four-byte instruction treated as one 8-byte one
    - This is independent from ISA design!
  - Why?
    - Smaller number of instructions to process
    - While still maintaining RISC ISA (Also used in CISC / x86 with smaller instructions)
    - Typical criticism of RISC is a larger number of generated instructions for same program

```
// rd = array[offset]
add rd, rs1, rs2
ld rd, 0(rd)
```

Can be fused into one instruction
Without more functionality in the execute stage

Source: Wikichip