# A Superscalar Out-of-Order x86 Soft Processor for FPGA

**Henry Wong**
University of Toronto, Intel
henry@stuffedcow.net

*June 5, 2019*
*Stanford University EE380*

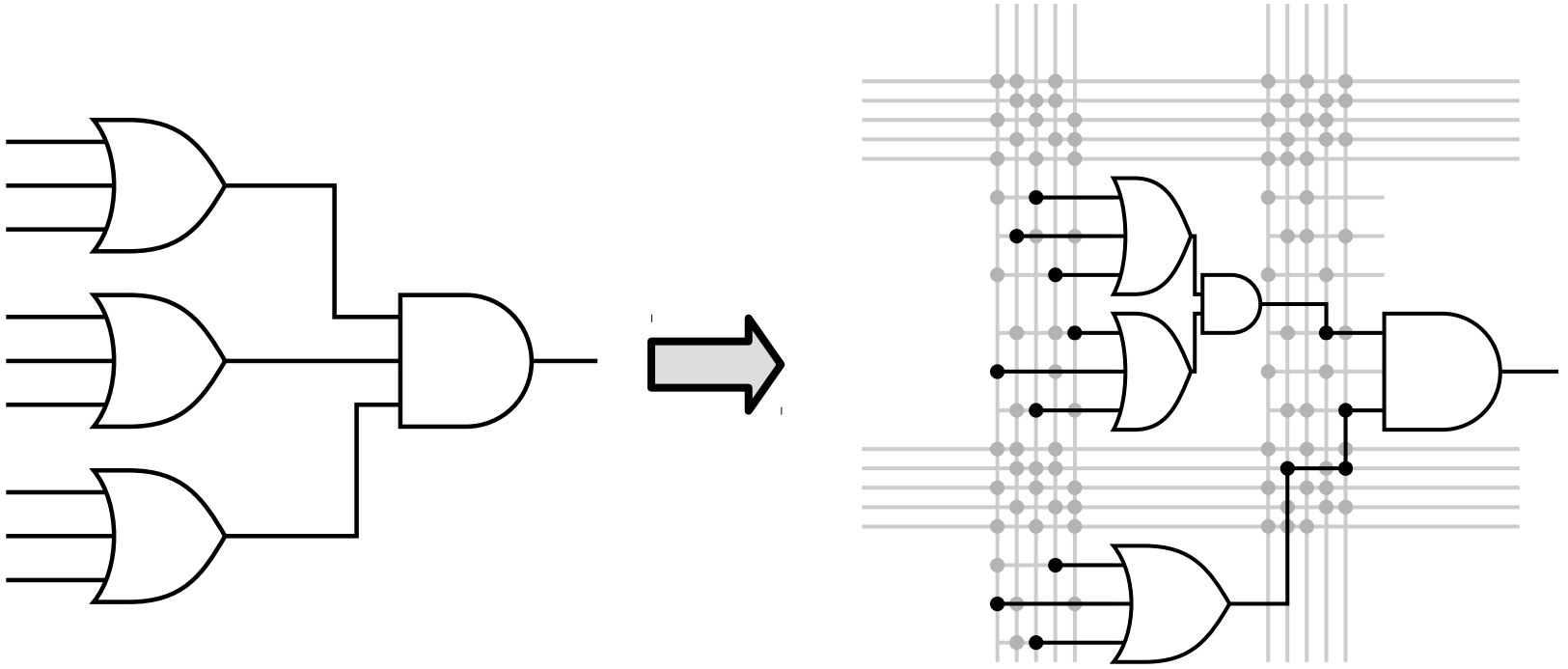
# Hi!

- CPU architect, Intel Hillsboro
- Ph.D., University of Toronto

- Today: x86 OoO processor for FPGA (Ph.D. work)
  - Motivation
  - High-level design and results
  - Microarchitecture details and some circuits

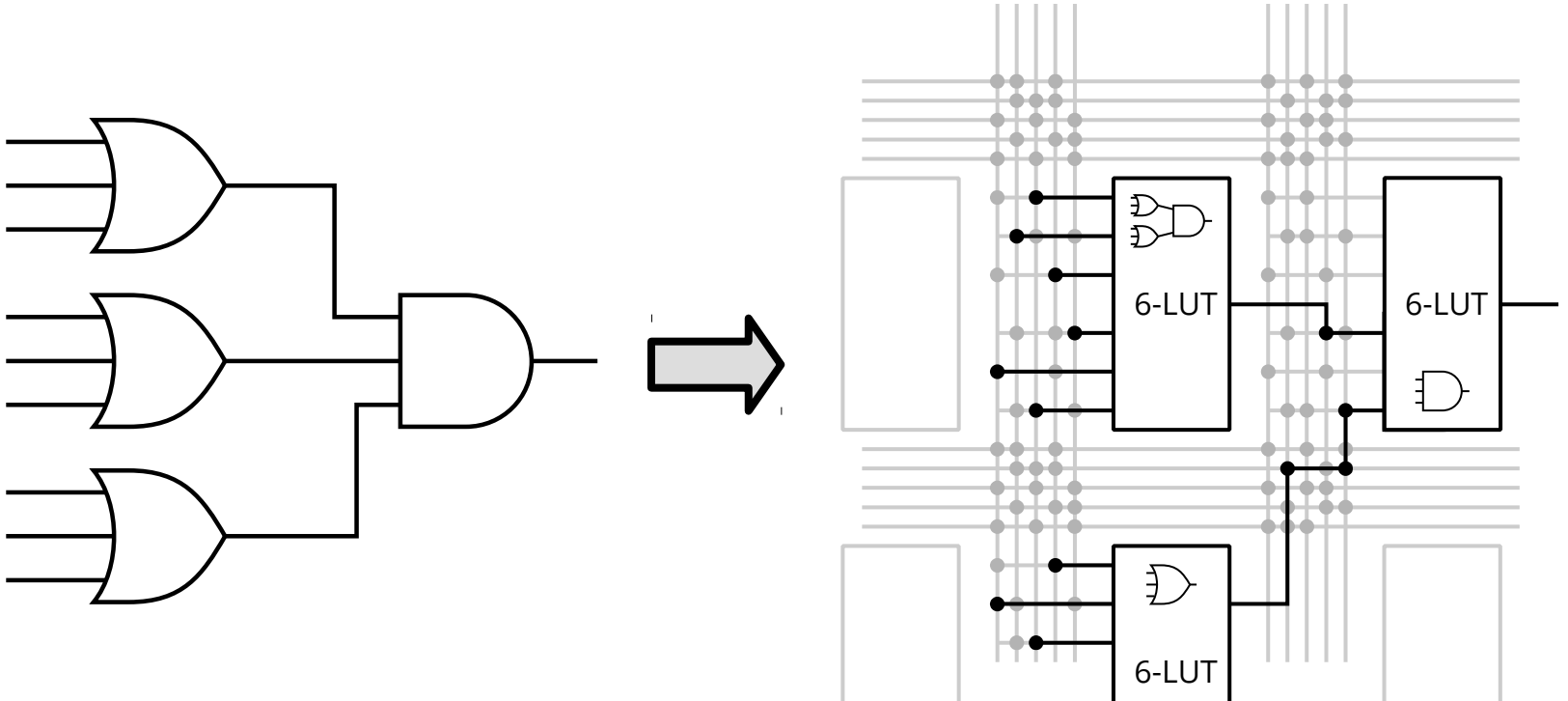
# FPGA: Field-Programmable Gate Array

- Is a digital circuit (logic gates and wires)
- Is field-programmable (at power-on, not in the fab)
- Pre-fab everything you'll ever need
  - 20x area, 20x delay cost
  - Circuit building blocks are somewhat bigger than logic gates



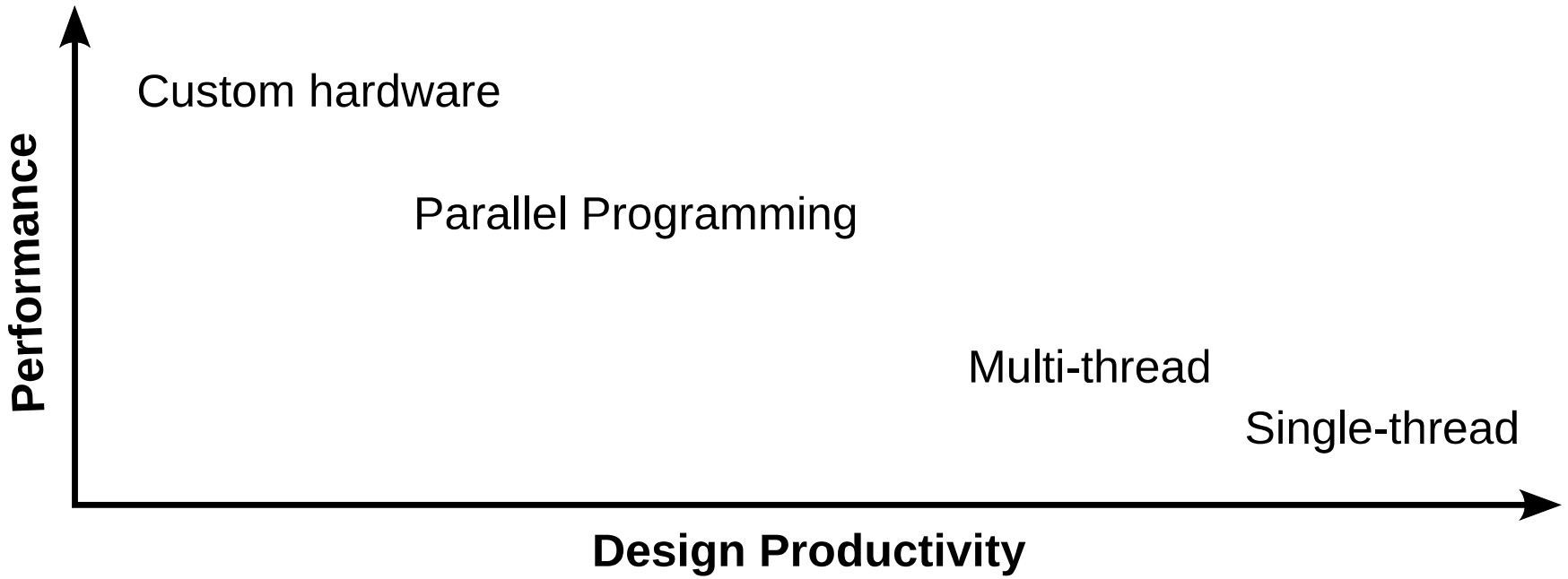
# FPGA: Field-Programmable Gate Array

- Is a digital circuit (logic gates and wires)
- Is field-programmable (at power-on, not in the fab)
- Pre-fab everything you'll ever need
  - 20x area, 20x delay cost
  - Circuit building blocks are somewhat bigger than logic gates



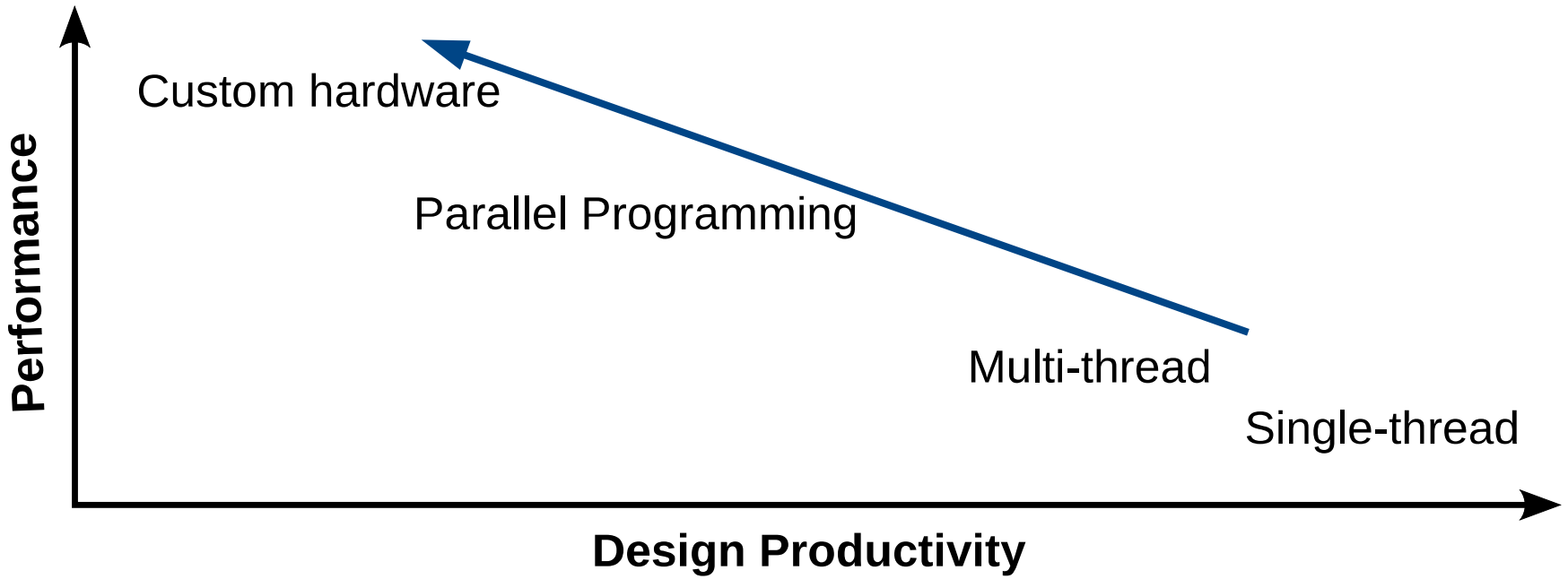
# FPGA Soft Processors

- FPGA systems often have software components
  - Often running on a soft processor
- Need more performance?
  - Parallel code and hardware accelerators need effort
  - Less effort if soft processors got faster



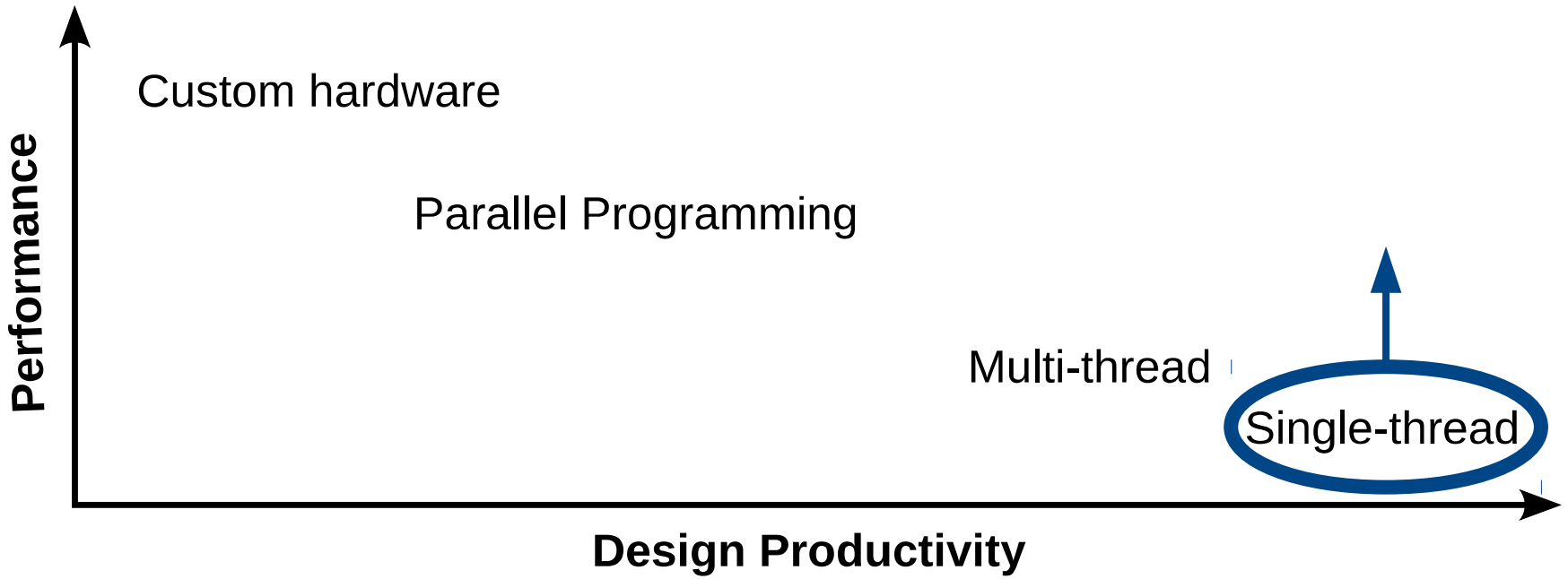
# FPGA Soft Processors

- FPGA systems often have software components
  - Often running on a soft processor
- Need more performance?
  - Parallel code and hardware accelerators need effort
  - Less effort if soft processors got faster



# FPGA Soft Processors

- FPGA systems often have software components
  - Often running on a soft processor
- Need more performance?
  - Parallel code and hardware accelerators need effort
  - Less effort if soft processors got faster



# Current Soft Processors

- CPUs from the FPGA vendors: Small, in-order, 1-way
  - Altera Nios II/f: 1100 ALUT (0.2% of Stratix IV), 240 MHz
  - Xilinx MicroBlaze: 2100 LUT (0.3% of Virtex-7), 246 MHz
- Some other examples:

| Processor | | ALUT | Frequency (Stratix IV) |
|---|---|---|---|
| Leon 3 | In-order 1-way | 5600 | 150 MHz |
| OpenRISC OR1200 | In-order 1-way | 3500 | 130 MHz |
| SPREX | In-order 1-way | 1800 | 150 MHZ (Stratix III) |
| BOOM (RISC-V) | OoO 2-way | ? | 50 MHz (Zync-7000) |
| OPA (RISC-V) | OoO 3-way | 12600 | 215 MHz |

# Current Soft Processors

- CPUs from the FPGA vendors: Small, in-order, 1-way
  - Altera Nios II/f: 1100 ALUT (0.2% of Stratix IV) 240 MHz
  - Xilinx MicroBlaze: 2100 LUT (0.3% of Virtex-7) 246 MHz
- Some other examples:

| Processor | | ALUT | Frequency (Stratix IV) |
|---|---|---|---|
| Leon 3 | In-order 1-way | 5600 | 150 MHz |
| OpenRISC OR1200 | In-order 1-way | 3500 | 130 MHz |
| SPREX | In-order 1-way | 1800 | 150 MHZ (Stratix III) |
| BOOM (RISC-V) | OoO 2-way | ? | 50 MHz (Zynq-7000) |
| OPA (RISC-V) | OoO 3-way | 12600 | 215 MHz |

# Faster Soft Processors

- Faster means extracting parallelism
  - Thread: Multi-core, multi-thread
  - Data: Vectors, SIMD
  - Instruction: Pipelining, out-of-order

- **Challenge: What microarchitecture and circuits?**
  - Increase instructions per clock cycle (IPC)
  - Without decreasing clock cycles per second (MHz)
  - And at an affordable FPGA hardware cost

- First OoO hard processors: ~1.5× IPC, ~1.0× frequency

# Faster Soft Processors

- Faster means extracting parallelism
  - Thread: Multi-core, multi-thread
  - Data: Vectors, SIMD
  - Instruction: Pipelining, out-of-order ← Least user effort

- **Challenge: What microarchitecture and circuits?**
  - Increase instructions per clock cycle (IPC)
  - Without decreasing clock cycles per second (MHz)
  - And at an affordable FPGA hardware cost

- First OoO hard processors: ~1.5× IPC, ~1.0× frequency

# ISA: Why not x86

- A few x86-specific features are really hard
  - Length decoding
  - Self-modifying code
  - Two destination operands (MUL, DIV)
  - …but most hard features are not x86-specific
- Existing ISA: Can’t shift unwanted features to software
- RISC-V: didn’t exist in 2009
  - But is RISC even the right clean-slate design?

# ISA: Why x86

- **It's easy to implement!**
- The alternative:
  - Benchmarks
    - "Way too much of my time is wasted on corraling benchmarks"
      — Chris Celio, on porting SPEC to RISC-V, 2015
    - "I'll just use the binaries" — me
  - OS, compiler, development tools
  - Web browser, JavaScript JIT, ...
  - Software in unexpected places: VGA BIOS
    - Alpha uses x86 emulation in firmware

# ISA: Why x86

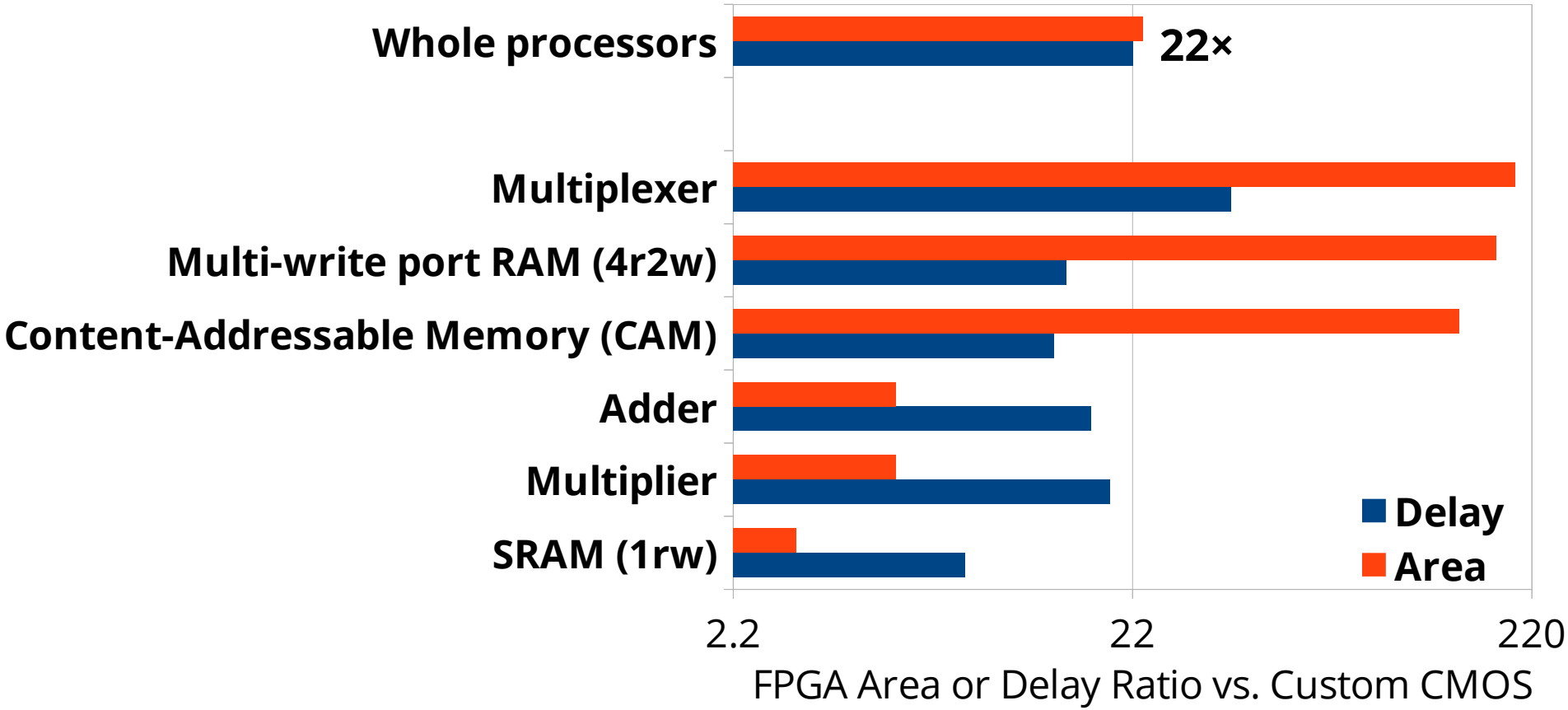
- **It's easy to implement!**
- The alternative:
  - Benchmarks
    - "Way too much of my time is wasted on corraling benchmarks"
      — Chris Celio, on porting SPEC to RISC-V, 2015
    - "I'll just use the binaries" — me
  - OS, compiler, development tools
  - Web browser, JavaScript JIT, …
  - Software in unexpected places: VGA BIOS
    - Alpha uses x86 emulation in firmware

**… I still can't do any of this**

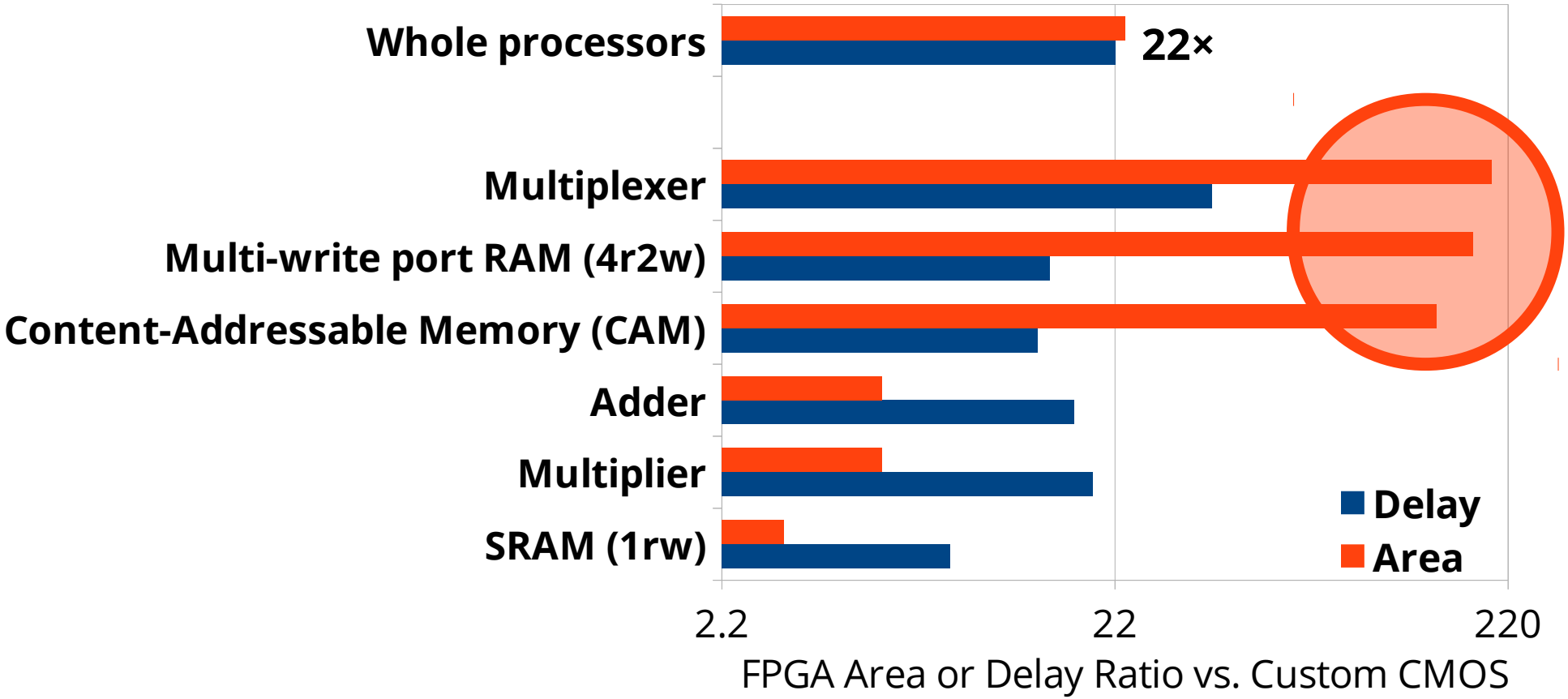
# Soft Processor Design Goals

- **Performance:** Two-issue, out-of-order
  - Reasonable per-clock performance (~2× vs. Nios II/f)
  - 300 MHz (25% higher than Nios II/f on Stratix IV)
- **No software rewrite:** 32-bit x86 instruction set (P6)
  - Binary compatible: OS, dev tools, user programs
  - Both system and user mode
    - Tested with 16 OSes and >50 user benchmarks

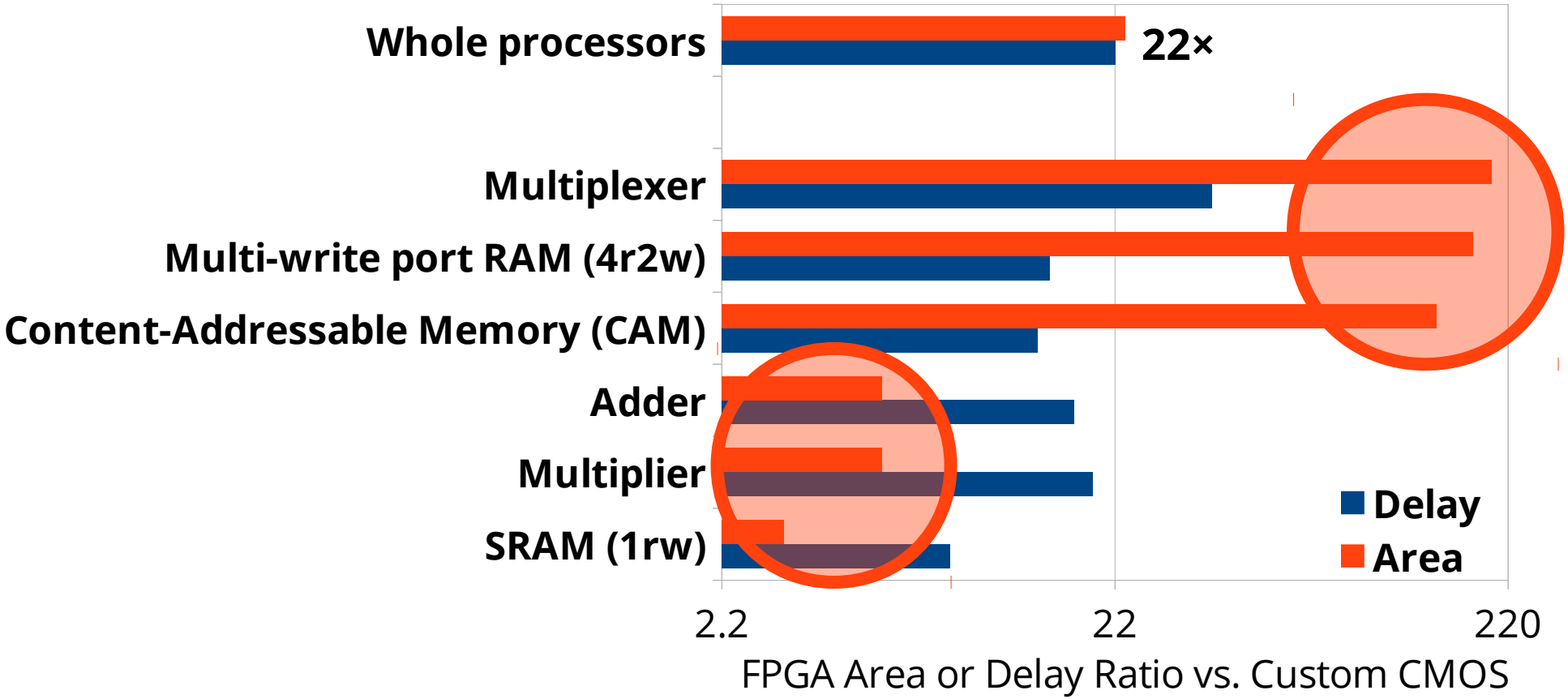
# How are FPGAs different from CMOS?



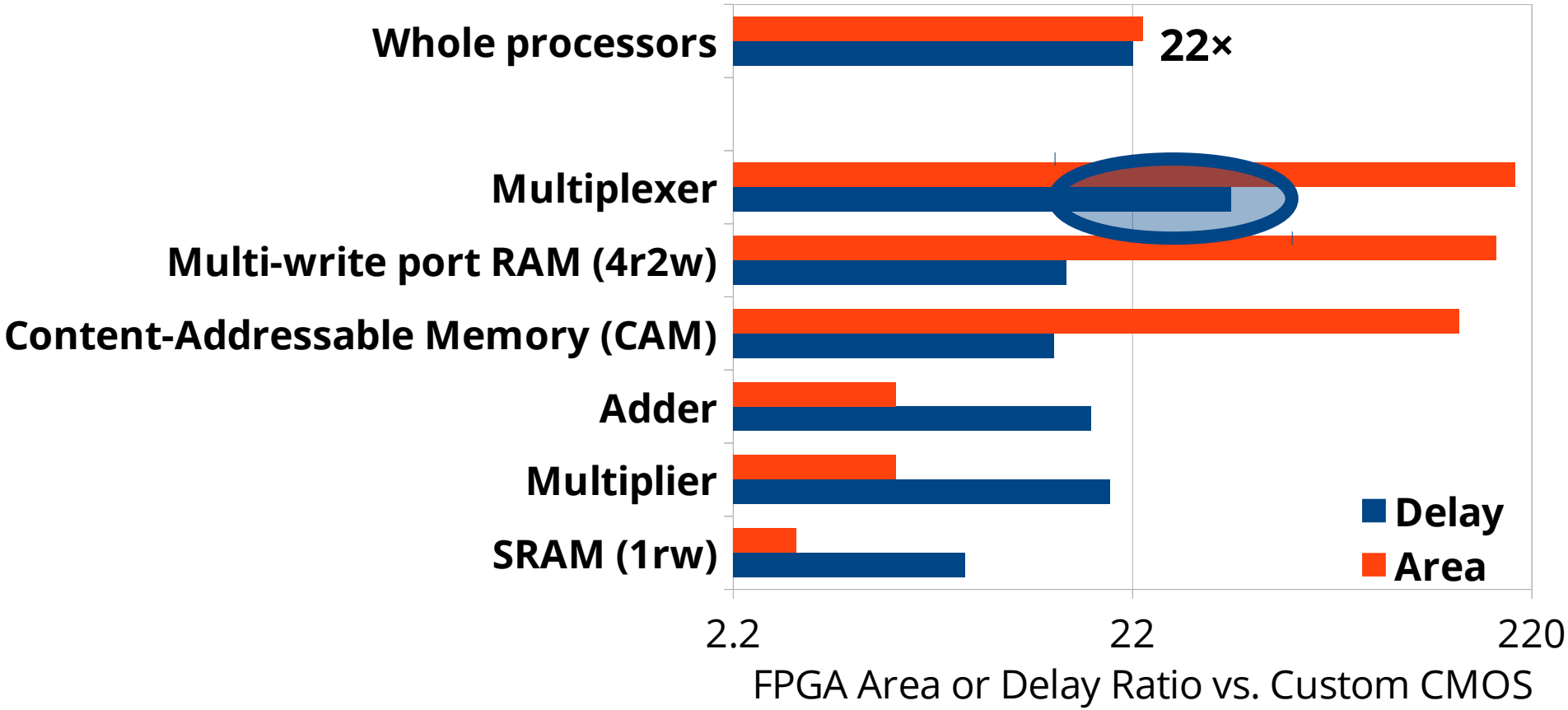
# How are FPGAs different from CMOS?



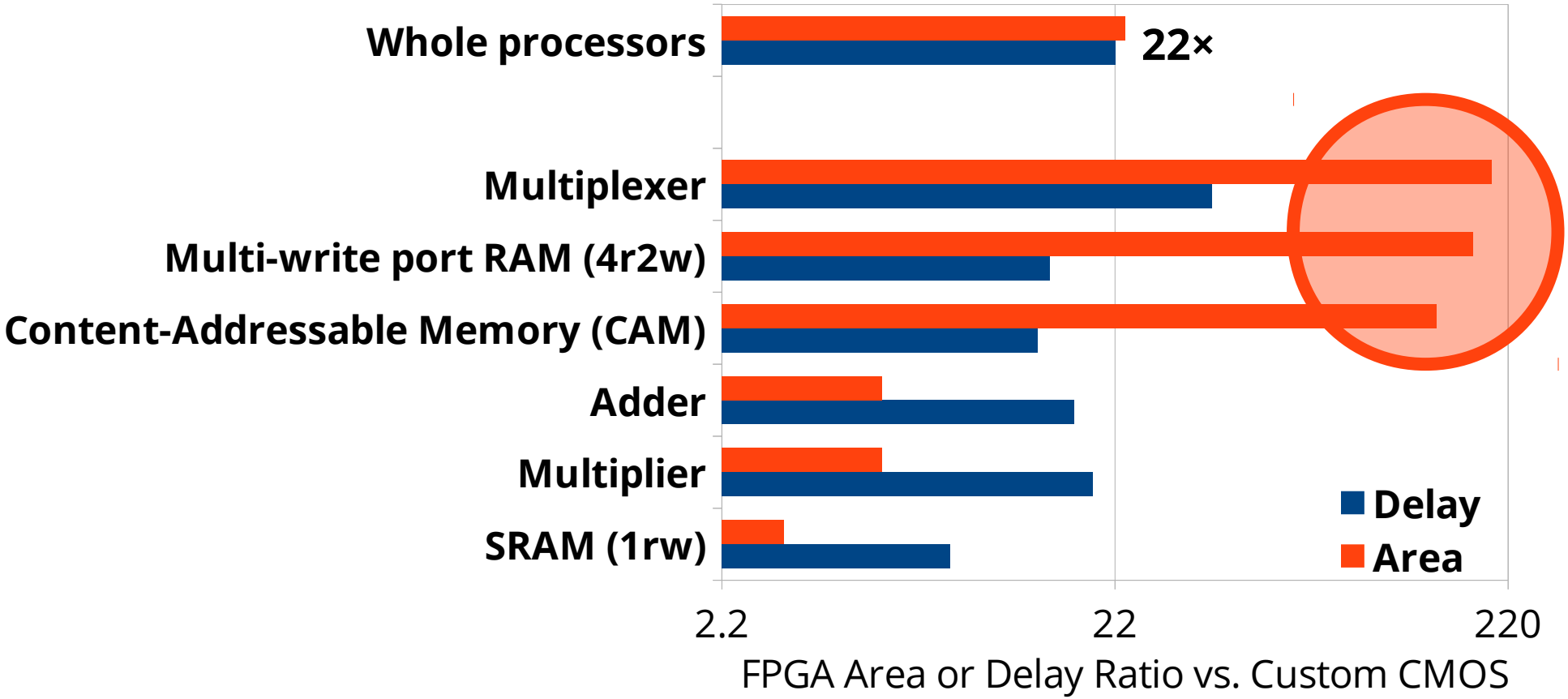
# How are FPGAs different from CMOS?



# How are FPGAs different from CMOS?



# How are FPGAs different from CMOS?

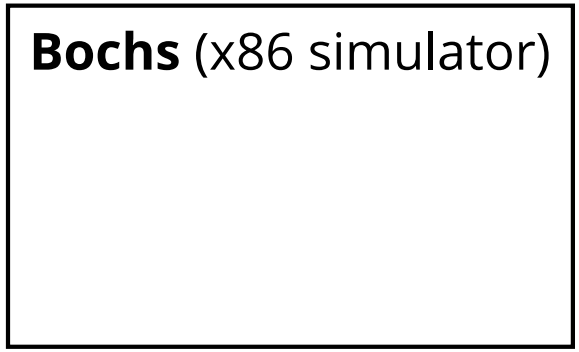



- Conclusion: ~Conventional processor microarchitecture, but try to avoid expensive circuits
  - Physical Register File: Reduce SRAM ports, CAM size
  - Low-associativity caches: Reduce multiplexers, CAMs

# Soft Processor Design Methodology

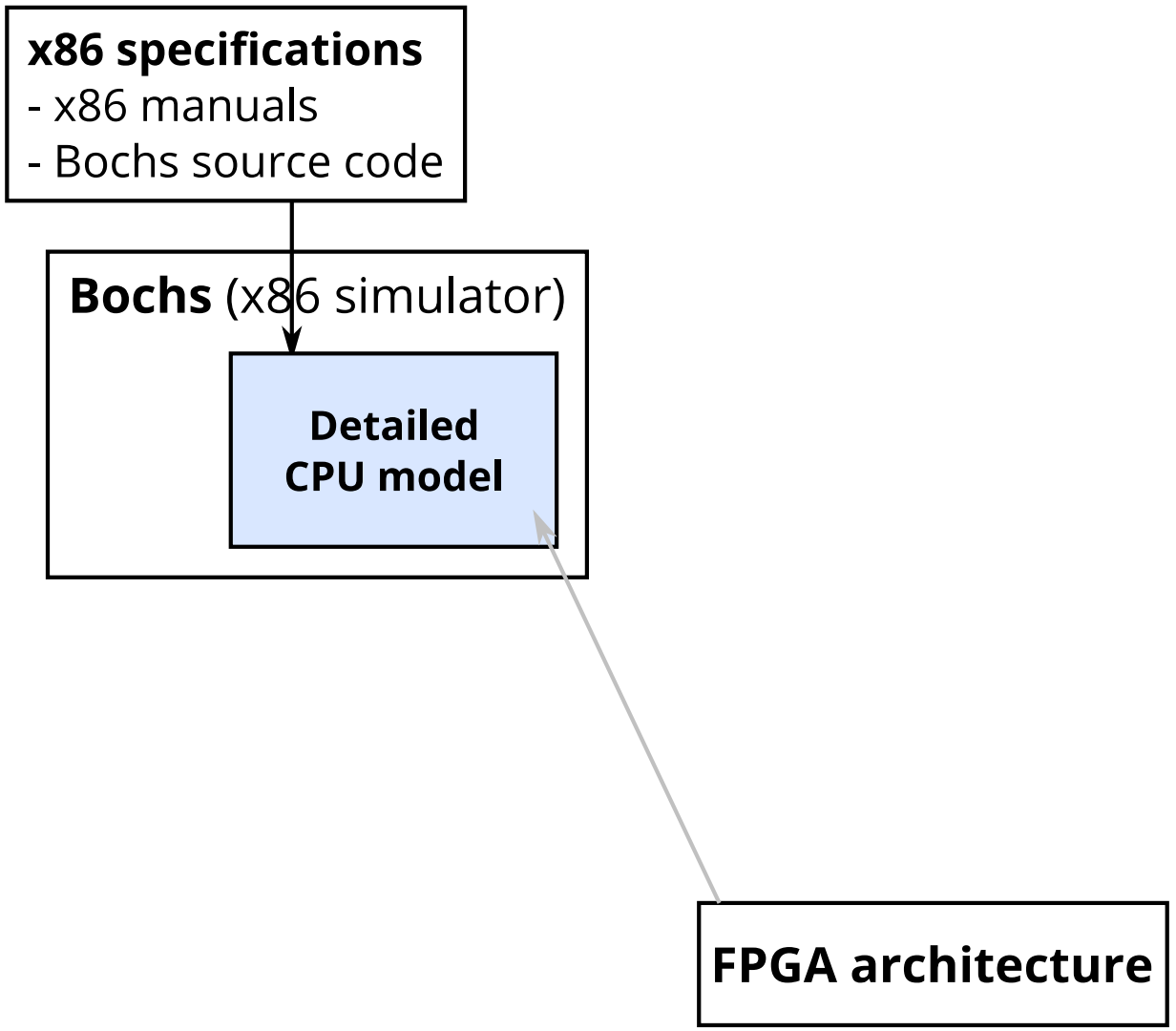
# Soft Processor Design Methodology

1. Bochs behavioural simulator



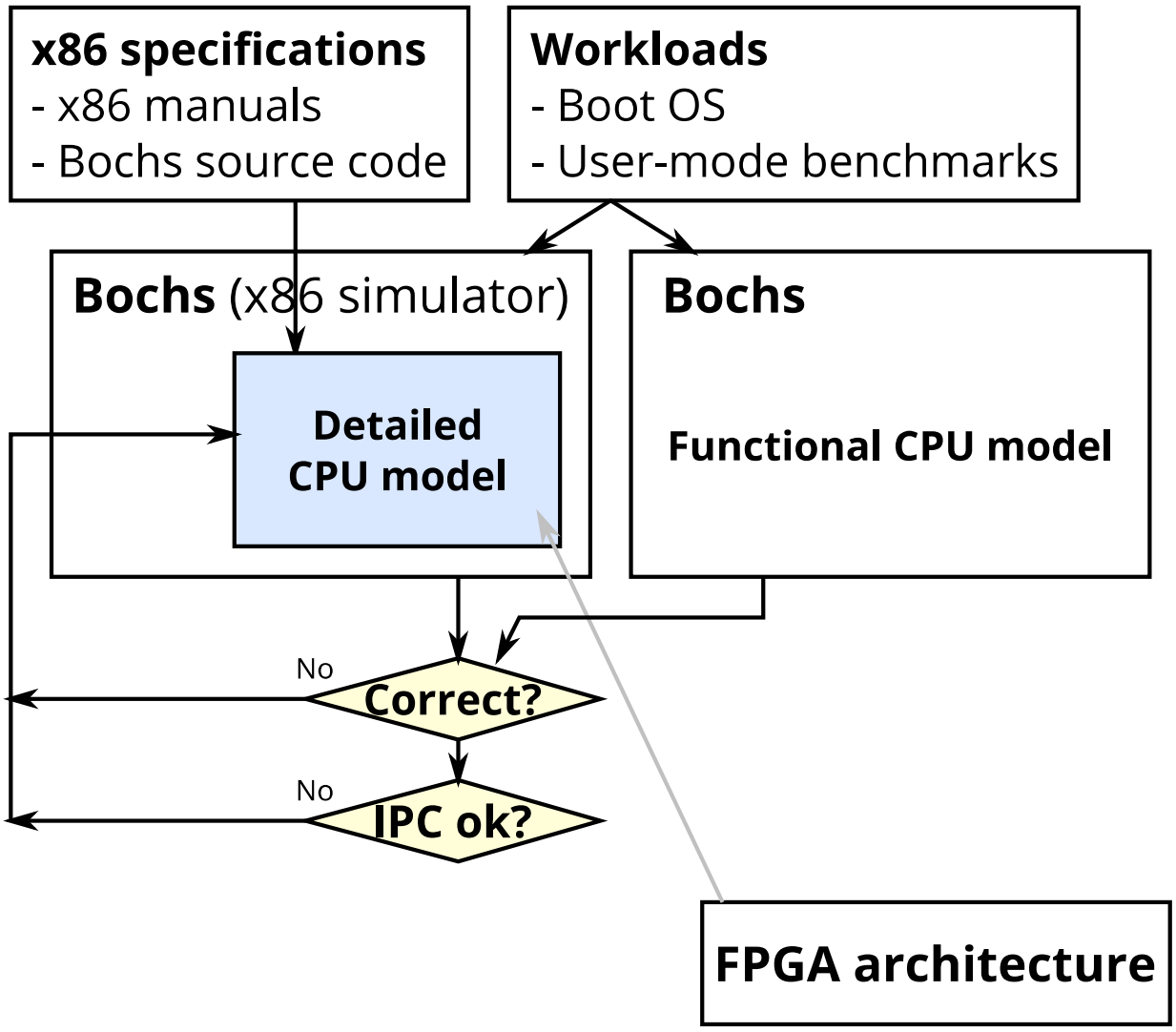
# Soft Processor Design Methodology

1. Bochs behavioural simulator
2. New detailed CPU pipeline model



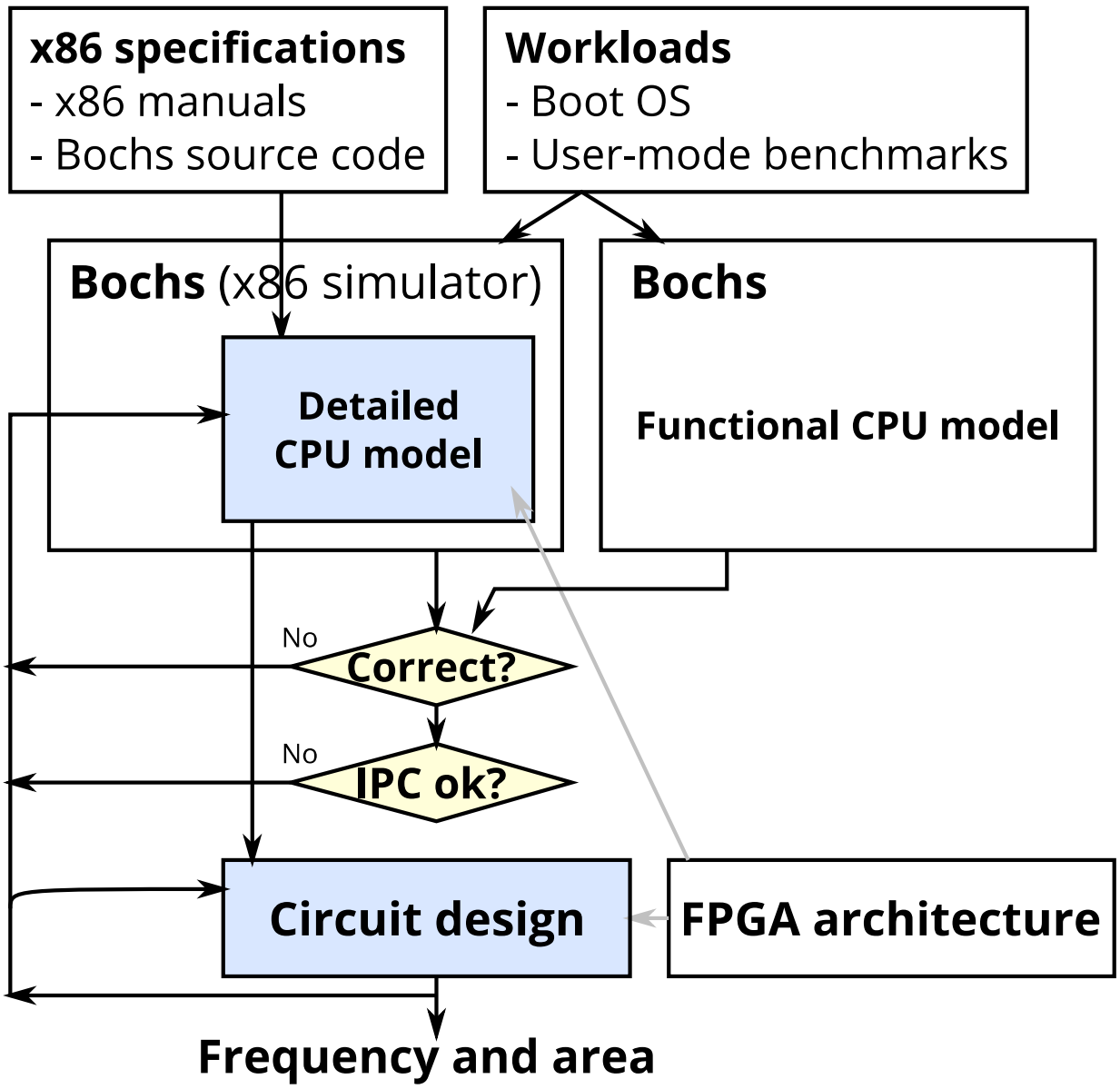
# Soft Processor Design Methodology

1. Bochs behavioural simulator
2. New detailed CPU pipeline model
3. Verify, optimize microarchitecture



# Soft Processor Design Methodology

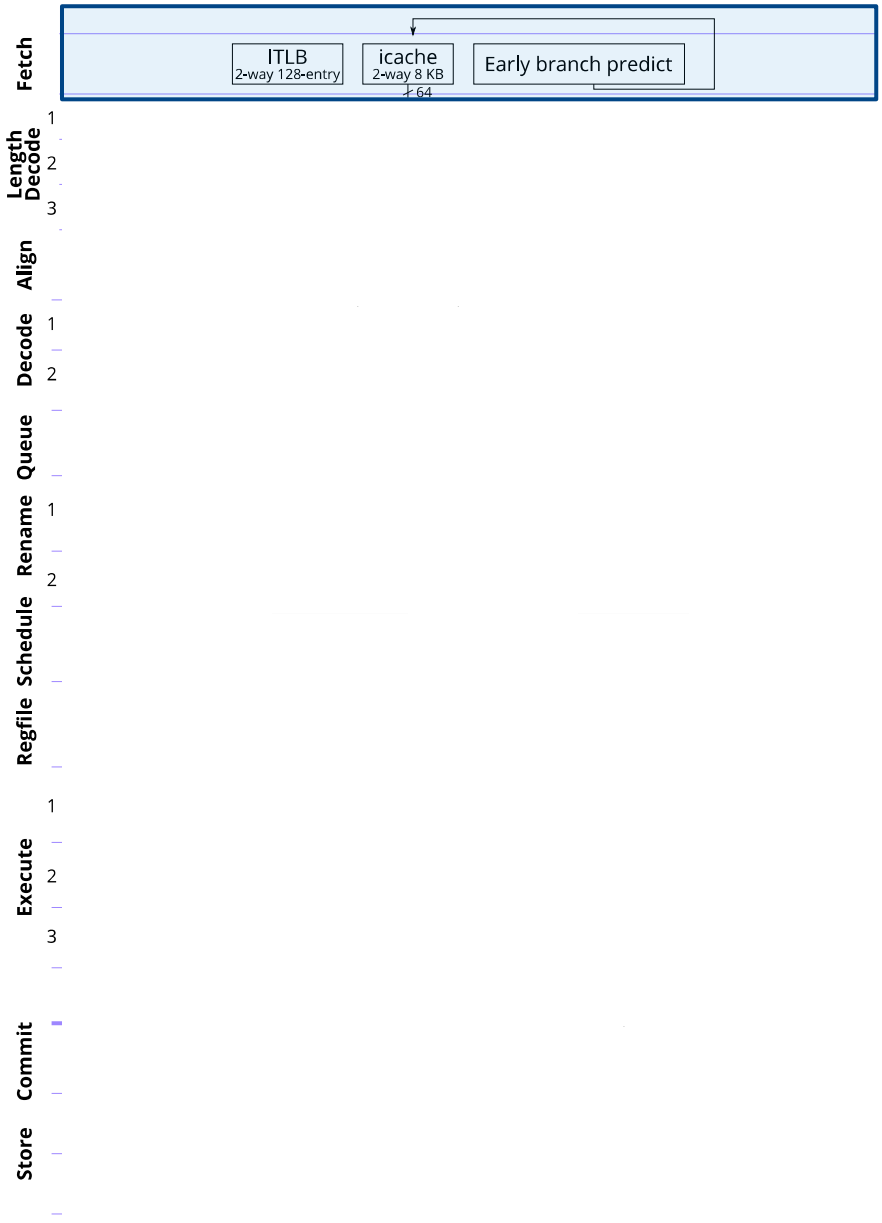
1. Bochs behavioural simulator
2. New detailed CPU pipeline model
3. Verify, optimize microarchitecture
4. Circuit design, optimization



# Our Processor's Microarchitecture

- Fetch

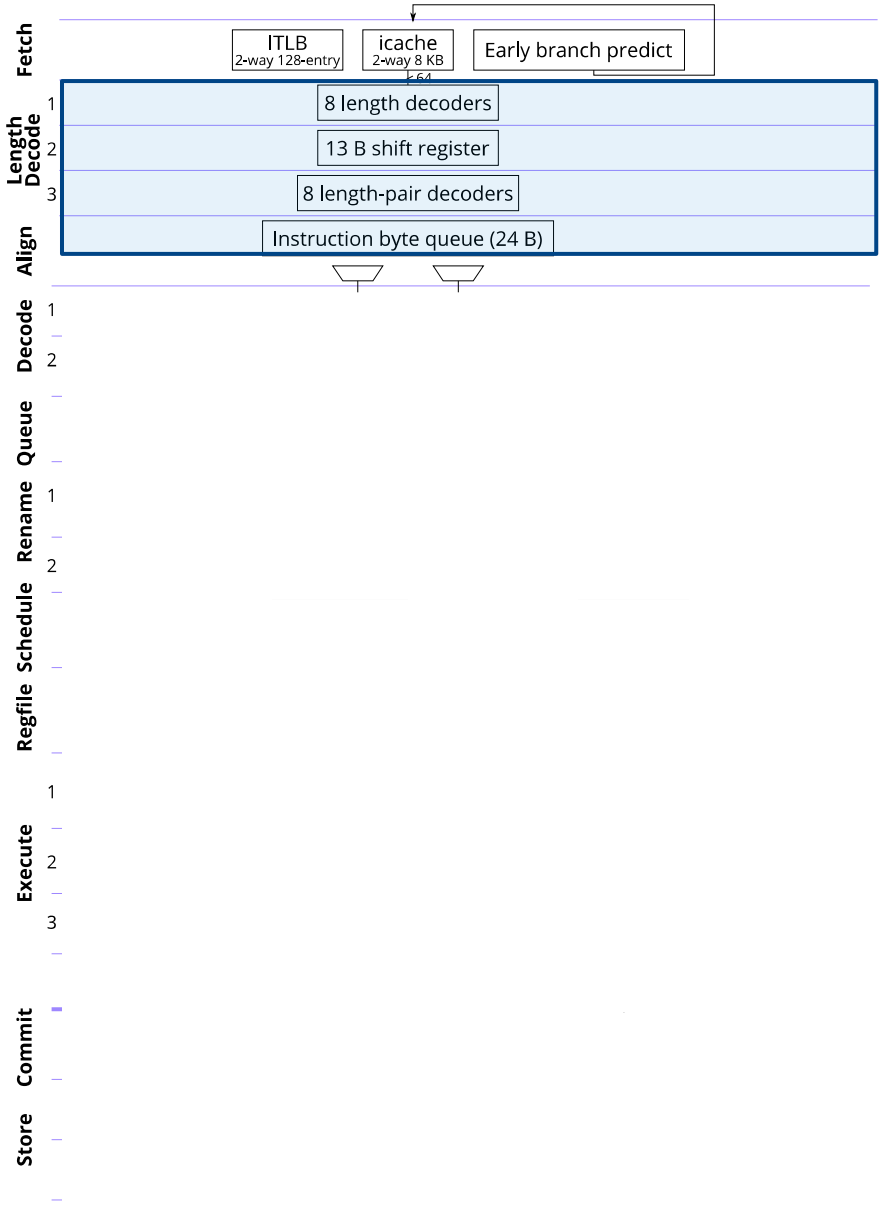
8b 06 03 46 04 89 46 08



# Our Processor's Microarchitecture

- Fetch
- Length decode

8b 06 03 46 04 89 46 08

# Our Processor's Microarchitecture

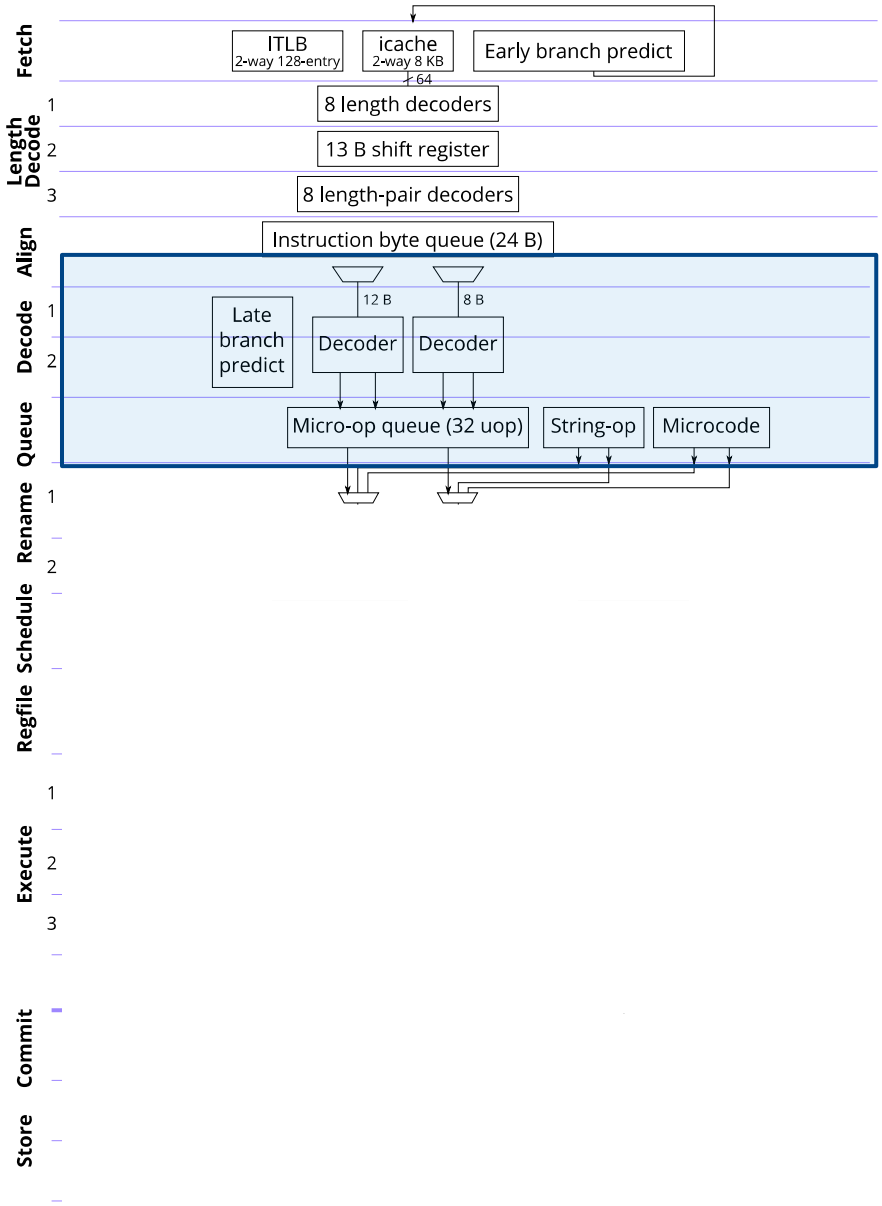
- Fetch
- Length decode
- Decode

8b 06 03 46 04 89 46 08


8b 06
mov eax, [esi]
load eax, [esi]

03 46 04
add eax, [esi+4]
load tmp0, [esi+4]
add eax, eax, tmp0

89 46 08
mov [esi+8], eax
store [esi+8], eax



# Our Processor's Microarchitecture

- Fetch
- Length decode
- Decode
- Rename

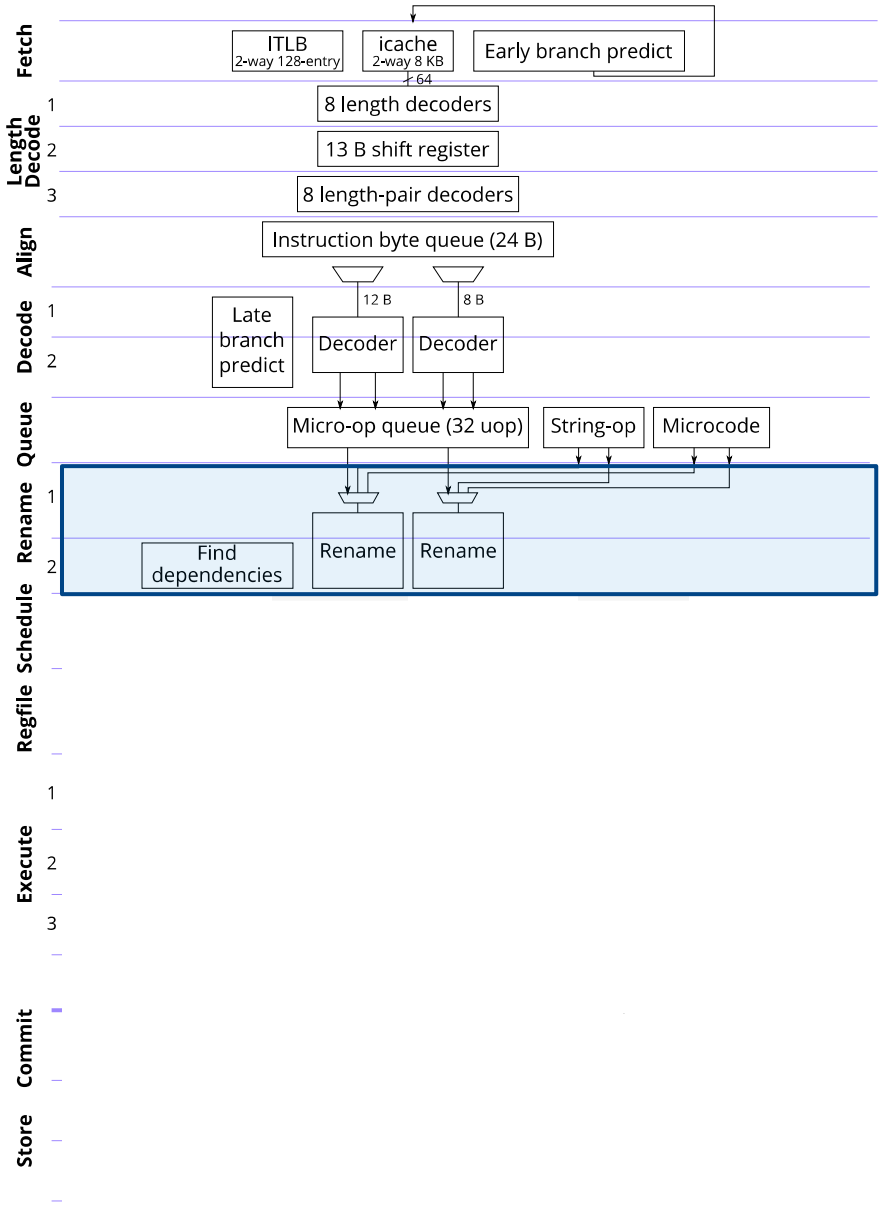
8b 06 03 46 04 89 46 08

8b 06 | 03 46 04 | 89 46 08

8b 06
mov eax, [esi]
load r1, [r4]

03 46 04
add eax, [esi+4]
load r20, [r4+4]
add r21, r1, r20

89 46 08
mov [esi+8], eax
store [r4+8], r21



# Our Processor's Microarchitecture

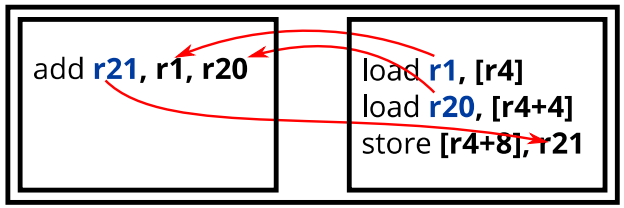
- Fetch
- Length decode
- Decode
- Rename
- Schedule

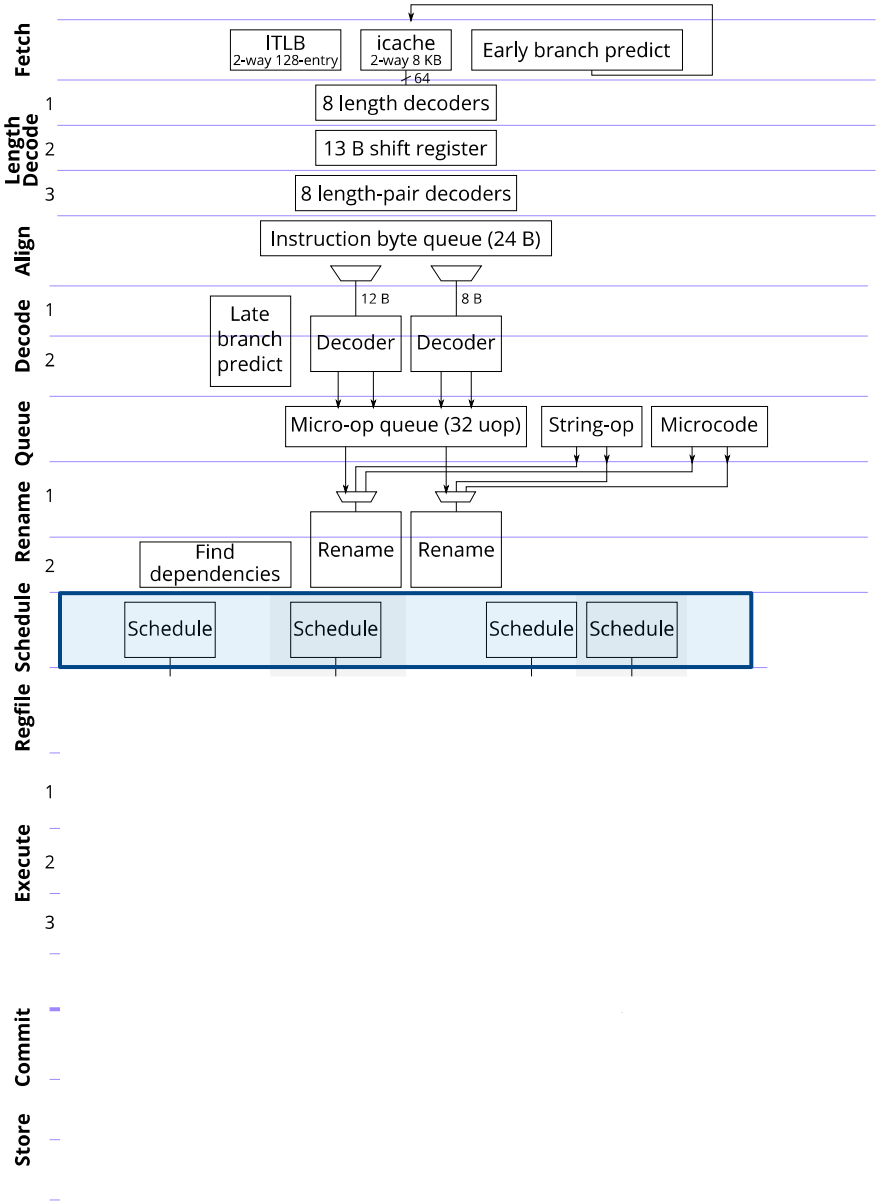
8b 06 03 46 04 89 46 08


8b 06
mov eax, [esi]
load r1, [r4]

03 46 04
add eax, [esi+4]
load r20, [r4+4]
add r21, r1, r20

89 46 08
mov [esi+8], eax
store [r4+8], r21



# Our Processor's Microarchitecture

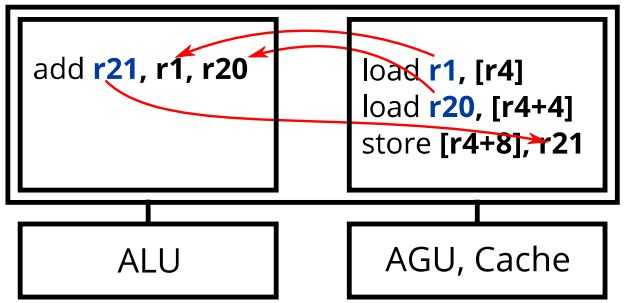
- Fetch
- Length decode
- Decode
- Rename
- Schedule
- Execute

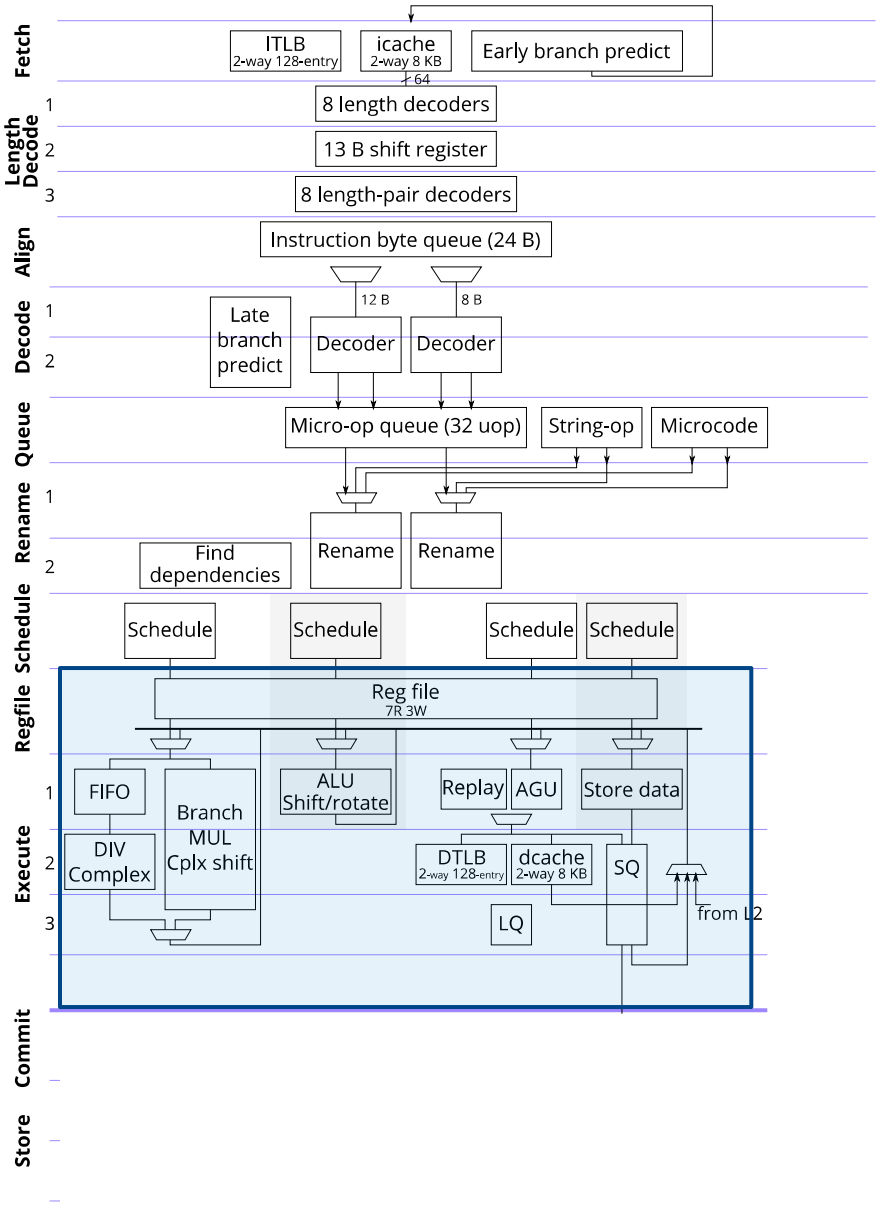
8b 06 03 46 04 89 46 08


8b 06
mov eax, [esi]
load r1, [r4]

03 46 04
add eax, [esi+4]
load r20, [r4+4]
add r21, r1, r20

89 46 08
mov [esi+8], eax
store [r4+8], r21



# Our Processor's Microarchitecture

- Fetch
- Length decode
- Decode
- Rename
- Schedule
- Execute
- Commit

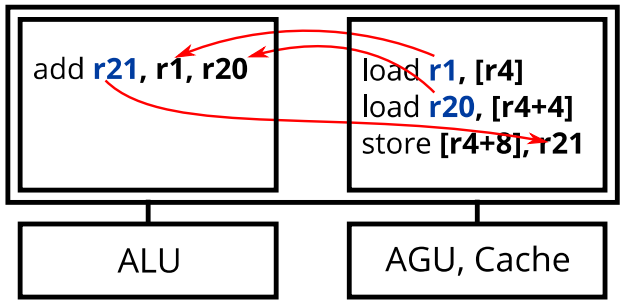
8b 06 03 46 04 89 46 08

8b 06 | 03 46 04 | 89 46 08

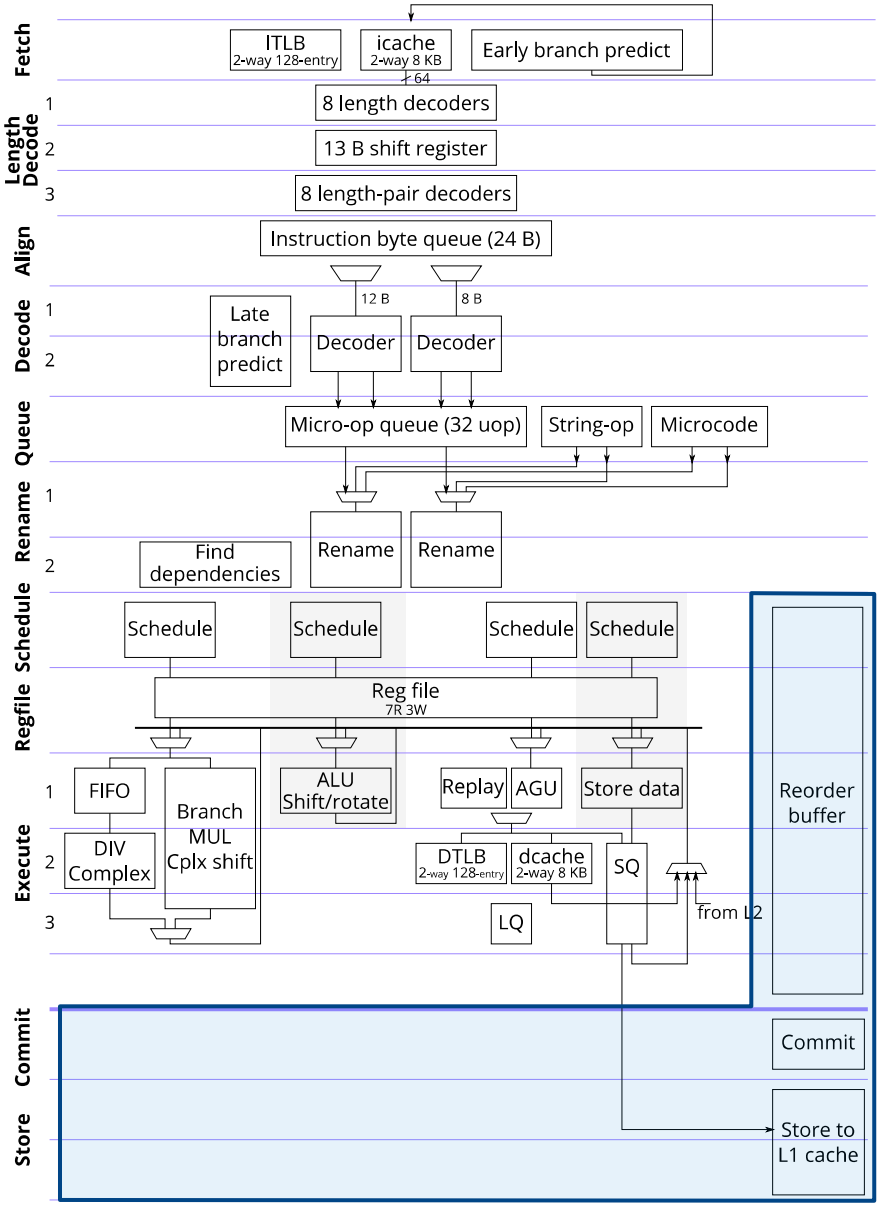
8b 06
mov eax, [esi]
load r1, [r4]

03 46 04
add eax, [esi+4]
load r20, [r4+4]
add r21, r1, r20

89 46 08
mov [esi+8], eax
store [r4+8], r21




✓ load r1, [r4]
✓ load r20, [r4+4]
add r21, r1, r20
store [r4+8], r21



# Our Processor's Microarchitecture

1.
- Fetch
- Length decode
- Decode
- Rename

8b 06 03 46 04 89 46 08

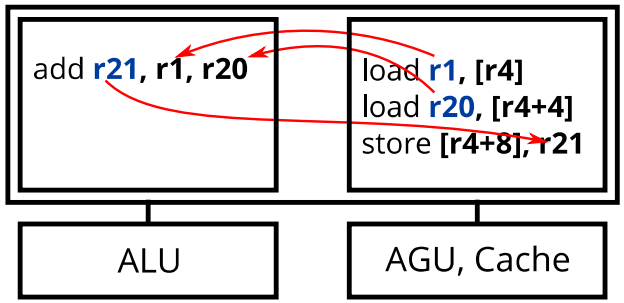
8b 06 | 03 46 04 | 89 46 08

8b 06 — mov eax, [esi] — load r1, [r4]

03 46 04 — add eax, [esi+4] — load r20, [r4+4]; add r21, r1, r20

89 46 08 — mov [esi+8], eax — store [r4+8], r21

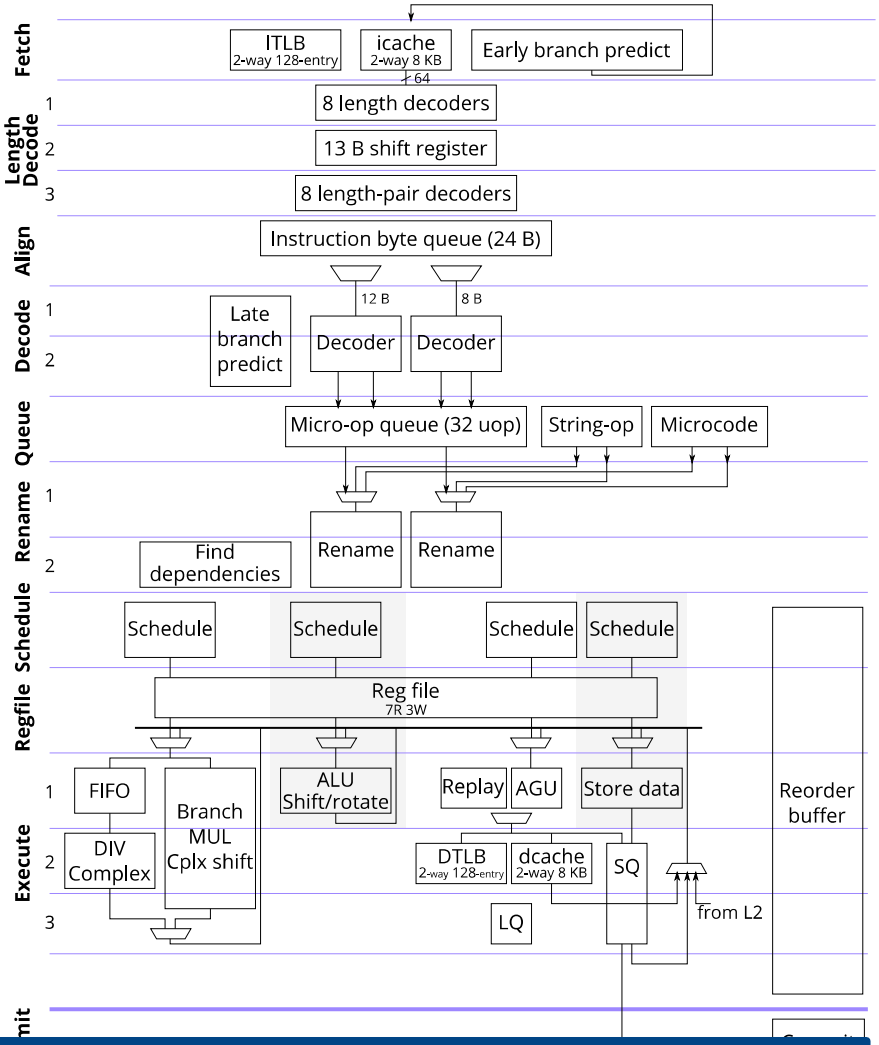
2. • Schedule




3. • Execute

✓ load r1, [r4]
✓ load r20, [r4+4]
add r21, r1, r20
store [r4+8], r21

- Commit




**Main challenges:**

1. Circuit complexity vs. support common case fast, worst-case correct
2. Circuit complexity vs. capacity (IPC)
3. Circuit complexity vs. pipeline latency (IPC)

# Our Processor's Microarchitecture

1.
- Fetch
- Length decode
- Decode
- Rename

2.
- Schedule

3.
- Execute

- Commit

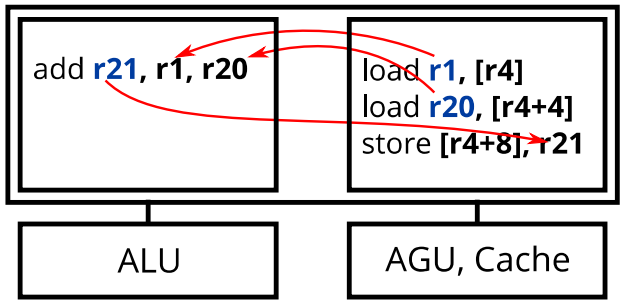
8b 06 03 46 04 89 46 08

8b 06 | 03 46 04 | 89 46 08

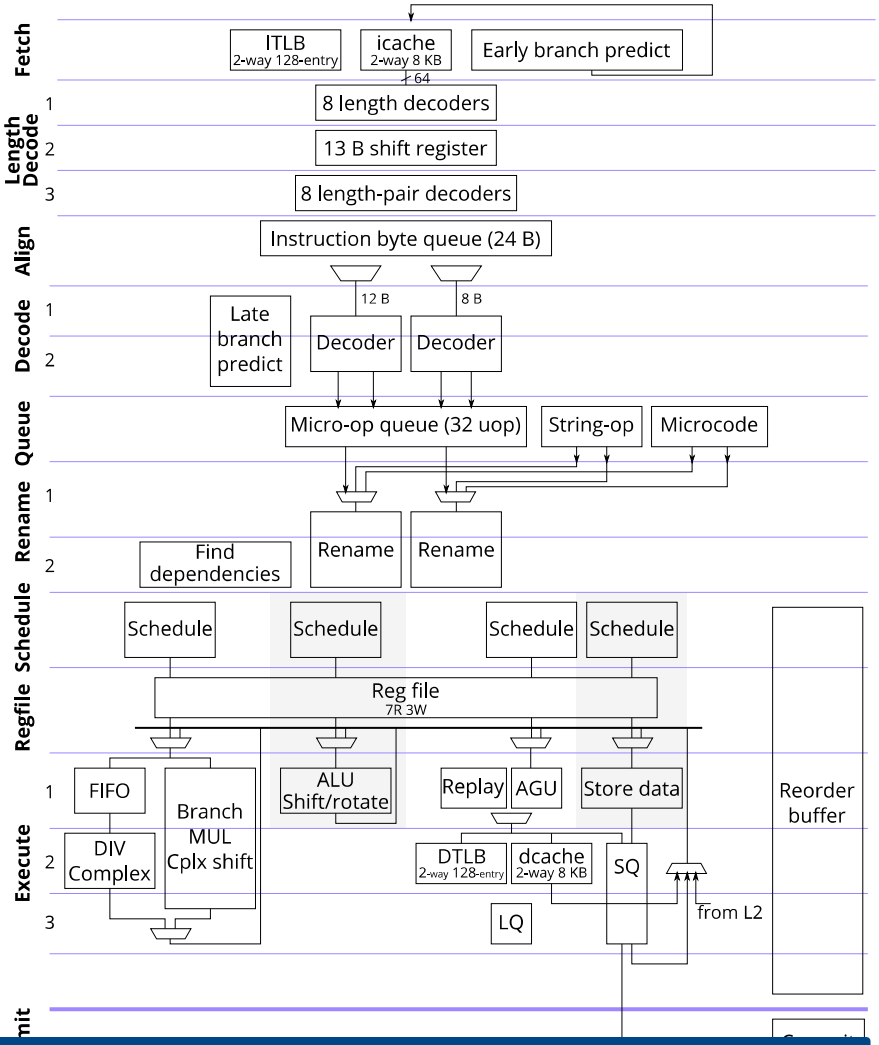
8b 06 — mov eax, [esi] — load r1, [r4]

03 46 04 — add eax, [esi+4] — load r20, [r4+4]; add r21, r1, r20

89 46 08 — mov [esi+8], eax — store [r4+8], r21




✓ load r1, [r4]
✓ load r20, [r4+4]
add r21, r1, r20
store [r4+8], r21




**Main challenges:**

1 Circuit complexity vs. support common case fast, worst-case correct

2 Circuit complexity vs. capacity (IPC)

3 Circuit complexity vs. pipeline latency (IPC)

# Our Processor's Microarchitecture

1.
- Fetch
- Length decode
- Decode
- Rename

2.
- Schedule

3.
- Execute

- Commit

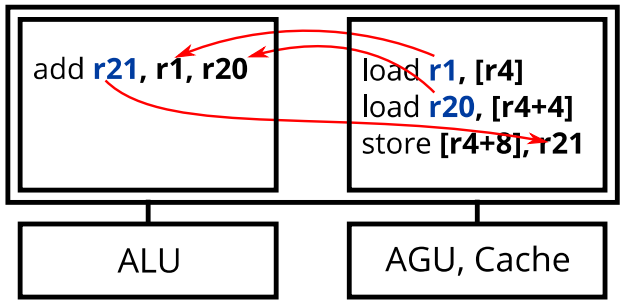
8b 06 03 46 04 89 46 08

8b 06 | 03 46 04 | 89 46 08

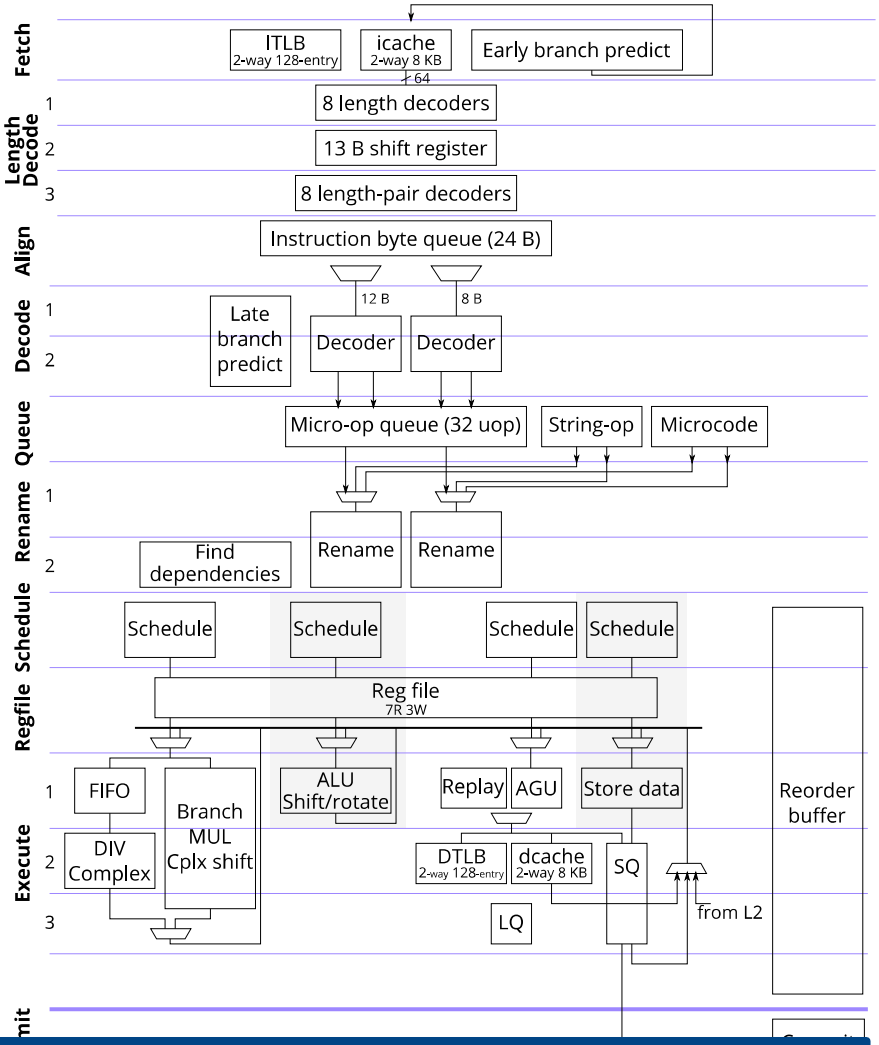
8b 06 — mov eax, [esi] — load r1, [r4]

03 46 04 — add eax, [esi+4] — load r20, [r4+4]; add r21, r1, r20

89 46 08 — mov [esi+8], eax — store [r4+8], r21




✓ load r1, [r4]
✓ load r20, [r4+4]
add r21, r1, r20
store [r4+8], r21




**Main challenges:**

1. Circuit complexity vs. support common case fast, worst-case correct
2. Circuit complexity vs. capacity (IPC)
3. Circuit complexity vs. pipeline latency (IPC)

# Processor Area and Frequency

| Component | Estimated Area (ALM) | Frequency (MHz) |
|---|---|---|
| Decode * | 6 000 | 247 |
| Renaming | 1 900 | 317 |
| Scheduler * | 4 000 | 275 |
| Register file | 2 400 | 260 |
| Execution | 2 300 | 240 |
| Memory system | (Logic) 9 000<br>(Caches) 5 000 | 200 |
| Commit + ROB * | 2 000 | |
| Microcode * | 500 | |
| **Total** | **28 700** | **200** |

* Area estimate for partial or unimplemented circuit

- Compare to Nios II/f with MMU and 32K L1I + 32K L1D
  - 4400 ALM (6.5×☹), 245 MHz (0.82×☹)

# Processor Area and Frequency

| Component | Estimated Area (ALM) | Frequency (MHz) |
|---|---|---|
| Decode * | 6 000 | 247 |
| Renaming | 1 900 | 317 |
| Scheduler * | 4 000 | 275 |
| Register file | 2 400 | 260 |
| Execution | 2 300 | 240 |
| Memory system | (Logic) 9 000<br>(Caches) 5 000 | 200 |
| Commit + ROB * | 2 000 | |
| Microcode * | 500 | |
| **Total** | **28 700** | **200** |

* Area estimate for partial or unimplemented circuit

7% of Stratix IV

- Compare to Nios II/f with MMU and 32K L1I + 32K L1D
  - 4400 ALM (6.5×☹), 245 MHz (0.82×☹)

# Processor Area and Frequency

| Component | Estimated Area (ALM) | Frequency (MHz) |
|---|---|---|
| Decode * | 6 000 | 247 |
| Renaming | 1 900 | 317 |
| Scheduler * | 4 000 | 275 |
| Register file | 2 400 | 260 |
| Execution | 2 300 | 240 |
| Memory system | (Logic) 9 000<br>(Caches) 5 000 | 200 |
| Commit + ROB * | 2 000 | |
| Microcode * | 500 | |
| **Total** | **28 700** | **200** |

* Area estimate for partial or unimplemented circuit

Optimize more?

7% of Stratix IV

- Compare to Nios II/f with MMU and 32K L1I + 32K L1D
  - 4400 ALM (6.5×☹), 245 MHz (0.82×☹)

# Processor Area and Frequency

| Component | Estimated Area (ALM) | Frequency (MHz) |
|---|---|---|
| Decode * | 6 000 | 247 |
| Renaming | 1 900 | 317 |
| Scheduler * | 4 000 | 275 |
| Register file | 2 400 | 260 |
| Execution | 2 300 | 240 |
| Memory system | (Logic) 9 000<br>(Caches) 5 000 | 200 |
| Commit + ROB * | 2 000 | |
| Microcode * | 500 | |
| **Total** | **28 700** | **200** |

* Area estimate for partial or unimplemented circuit

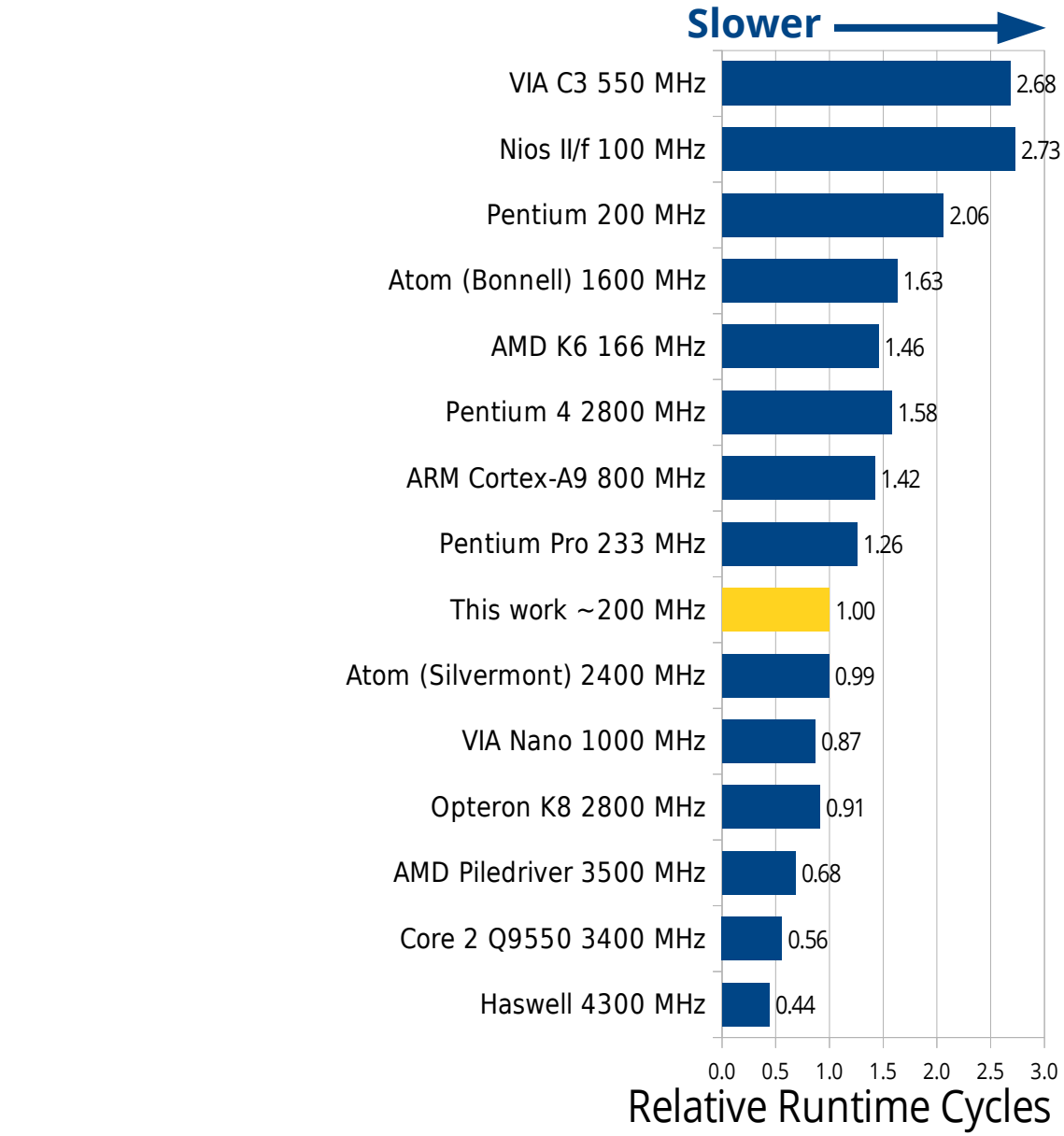
"OoO stuff"

Optimize more?

7% of Stratix IV

- Compare to Nios II/f with MMU and 32K L1I + 32K L1D
  - 4400 ALM (6.5×☹), 245 MHz (0.82×☹)

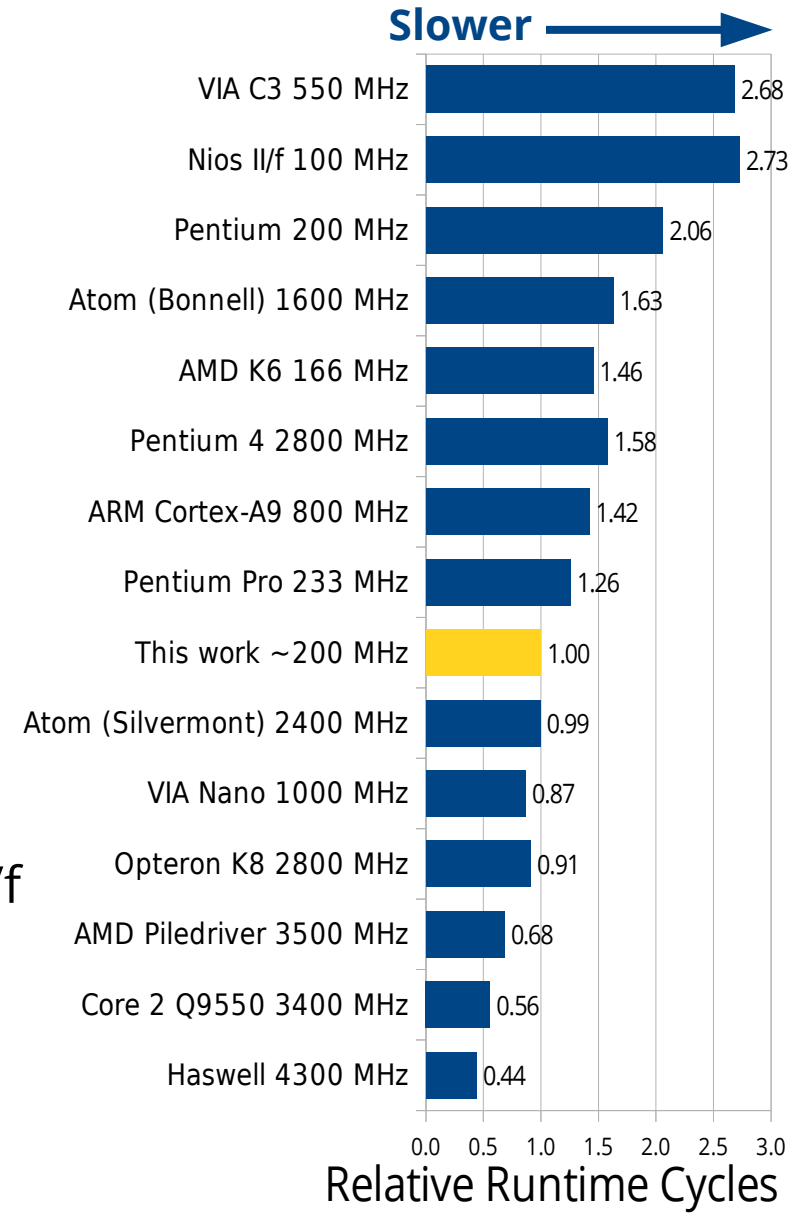
# Per-clock performance (SPECint2000)



# Per-clock performance (SPECint2000)

- Nios II/f: 2.73×
  - Wall-clock: 2.23×
- Pentium Pro (1995): 1.26×
  - Also 8 KB/256 KB cache
  - 3-way OoO
- Atom Silvermont (2013): 0.99×
  - Also 2-way OoO
  - 32 KB/2 MB cache

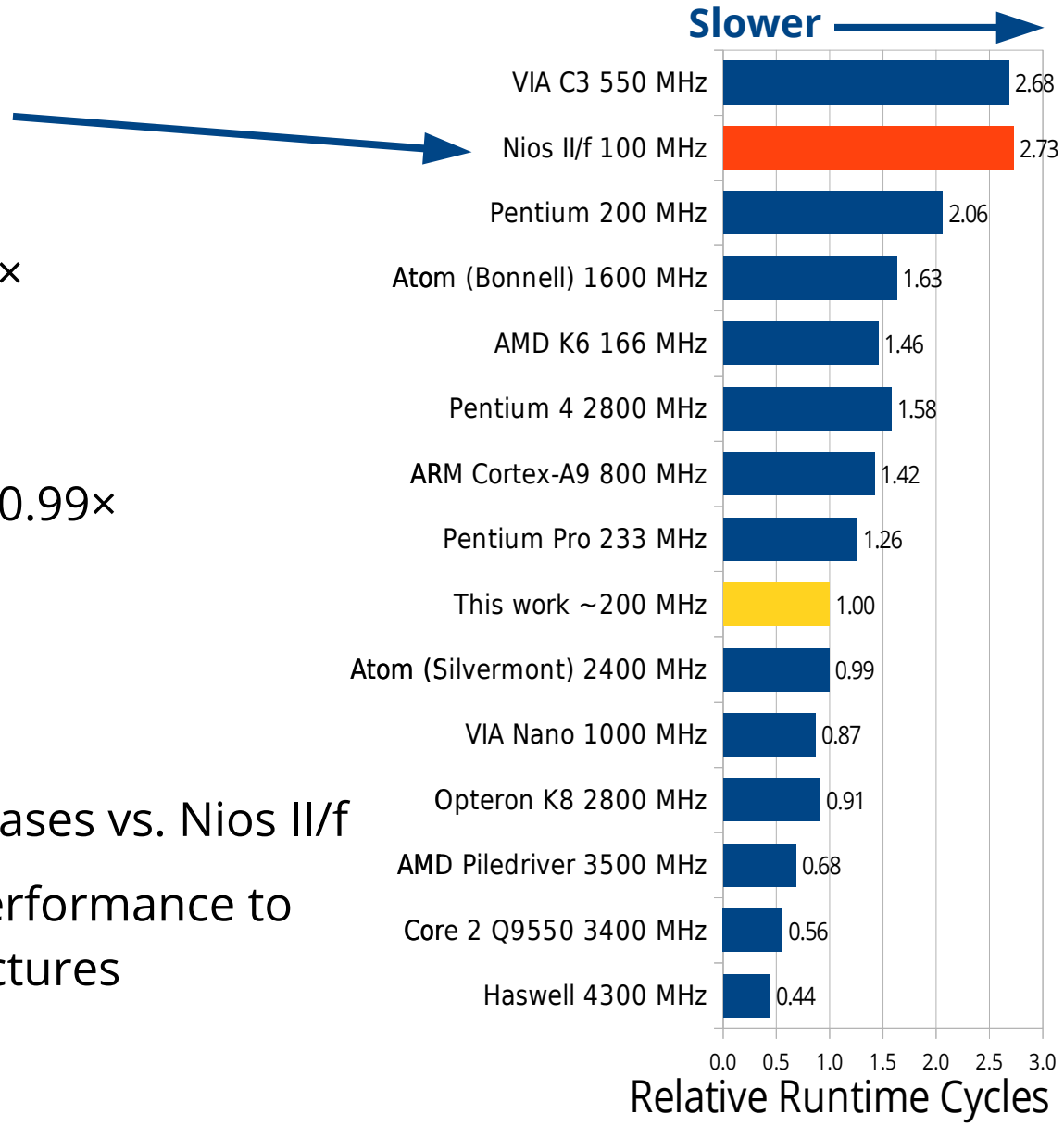
- Large performance increases vs. Nios II/f
- Comparable per-clock performance to similar x86 microarchitectures



# Per-clock performance (SPECint2000)

- Nios II/f: 2.73×
  - Wall-clock: 2.23× 🙂
- Pentium Pro (1995): 1.26×
  - Also 8 KB/256 KB cache
  - 3-way OoO
- Atom Silvermont (2013): 0.99×
  - Also 2-way OoO
  - 32 KB/2 MB cache

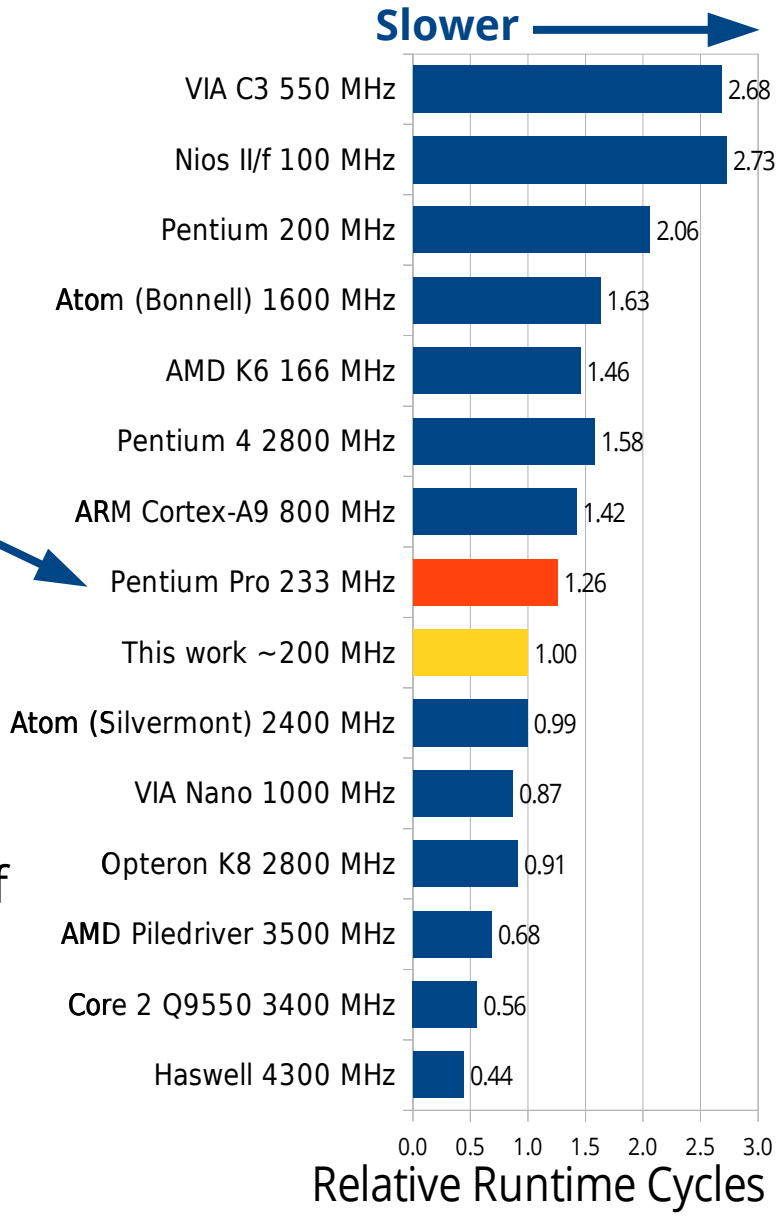
- Large performance increases vs. Nios II/f
- Comparable per-clock performance to similar x86 microarchitectures



# Per-clock performance (SPECint2000)

- Nios II/f: 2.73×
  - Wall-clock: 2.23× 🙂
- Pentium Pro (1995): 1.26×
  - Also 8 KB/256 KB cache
  - 3-way OoO
- Atom Silvermont (2013): 0.99×
  - Also 2-way OoO
  - 32 KB/2 MB cache

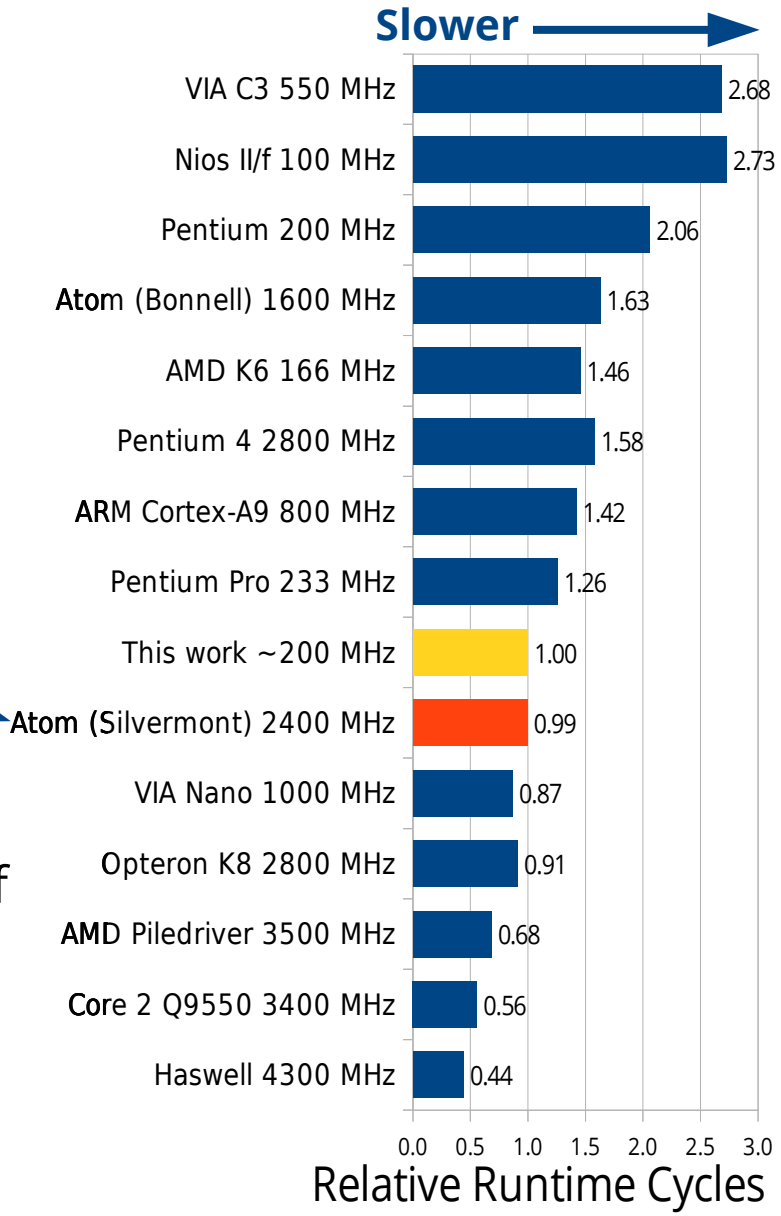
- Large performance increases vs. Nios II/f
- Comparable per-clock performance to similar x86 microarchitectures



# Per-clock performance (SPECint2000)

- Nios II/f: 2.73×
  - Wall-clock: 2.23× 🙂
- Pentium Pro (1995): 1.26×
  - Also 8 KB/256 KB cache
  - 3-way OoO
- Atom Silvermont (2013): 0.99×
  - Also 2-way OoO
  - 32 KB/2 MB cache

- Large performance increases vs. Nios II/f
- Comparable per-clock performance to similar x86 microarchitectures



# Summary 1

- Designed microarchitecture and circuits for a superscalar out-of-order x86 soft processor
  - Both user and system modes
- Area: **6.5×** Nios II/f, but affordable
  - 7% Stratix IV or 1.3% Stratix 10
- Performance: **2.2×** Nios II/f on SPECint2000
  - Per-clock: ~2.7×
  - Frequency: ~0.8×

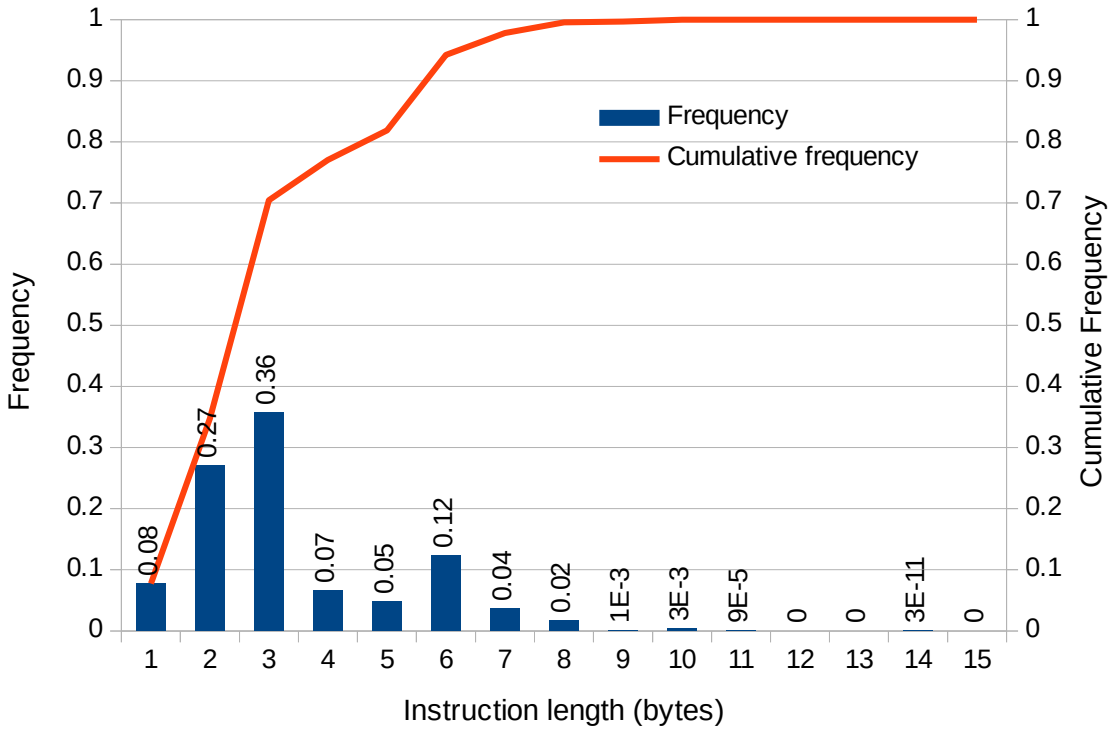
- Out-of-order increases soft processor performance without rewriting software
- x86 is feasible on FPGA

# Part 2: Pipeline Details

- Sketch of interesting circuits at each stage
- Timing budget: ~5 LUT levels (< 3.5 ns)
- Many circuits designed bottom-up
  - LUT granularity

# Front end: Fetch-decode

- Fetch bandwidth
  - 3.4 B per instruction
  - Fetch 8 B/cycle

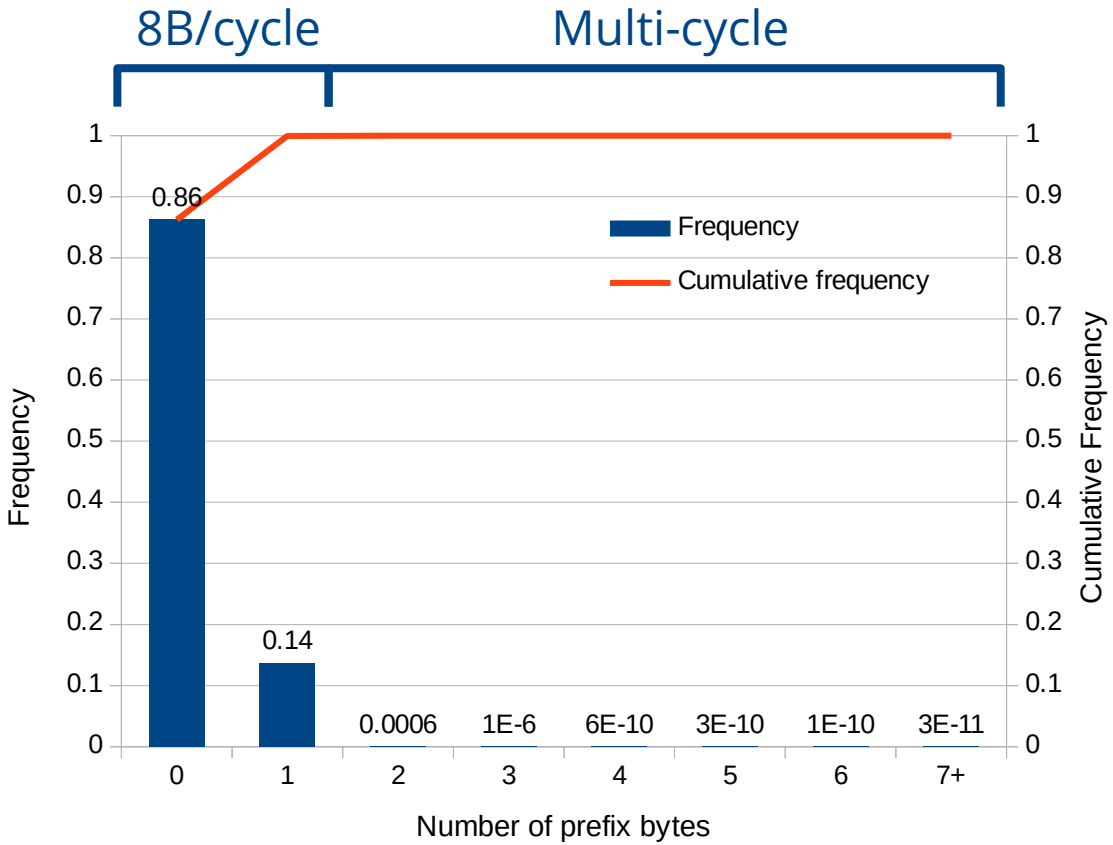



**x86**: Worst case is complex, but common case isn't too bad

ICache → **Bytes** → **Instructions** → **Micro-ops** → Renamer

# Front end: Fetch-decode

- Fetch bandwidth
  - 3.4 B per instruction
  - Fetch 8 B/cycle
- Length decode
  - Prefix bytes uncommon
  - Fast decode up to 1 prefix

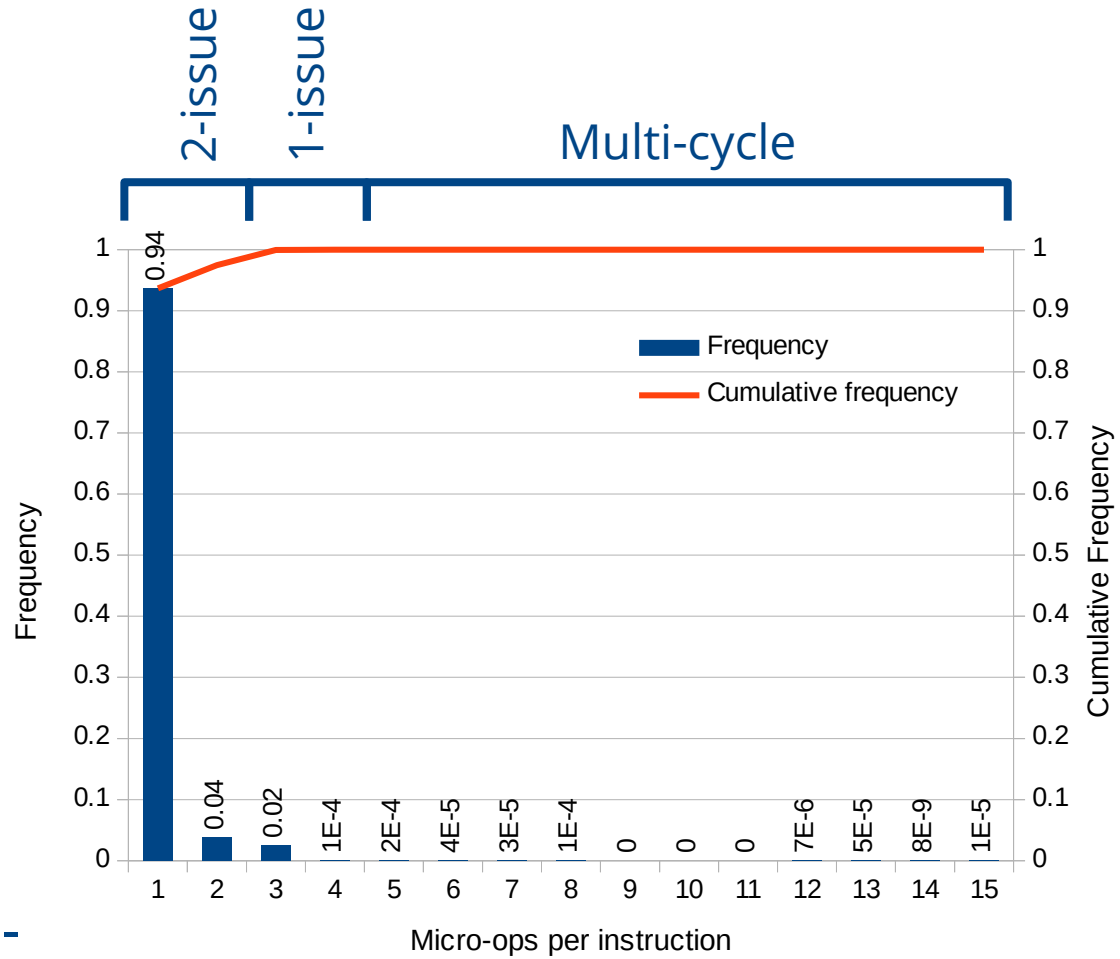



**x86**: Worst case is complex, but common case isn't too bad

ICache → **Bytes** → **Instructions** → **Micro-ops** → Renamer

# Front end: Fetch-decode

- Fetch bandwidth
  - 3.4 B per instruction
  - Fetch 8 B/cycle
- Length decode
  - Prefix bytes uncommon
  - Fast decode up to 1 prefix
- Decode into micro-ops
  - 1 is common case
  - Dual-issue up to 2, single-issue up to 4

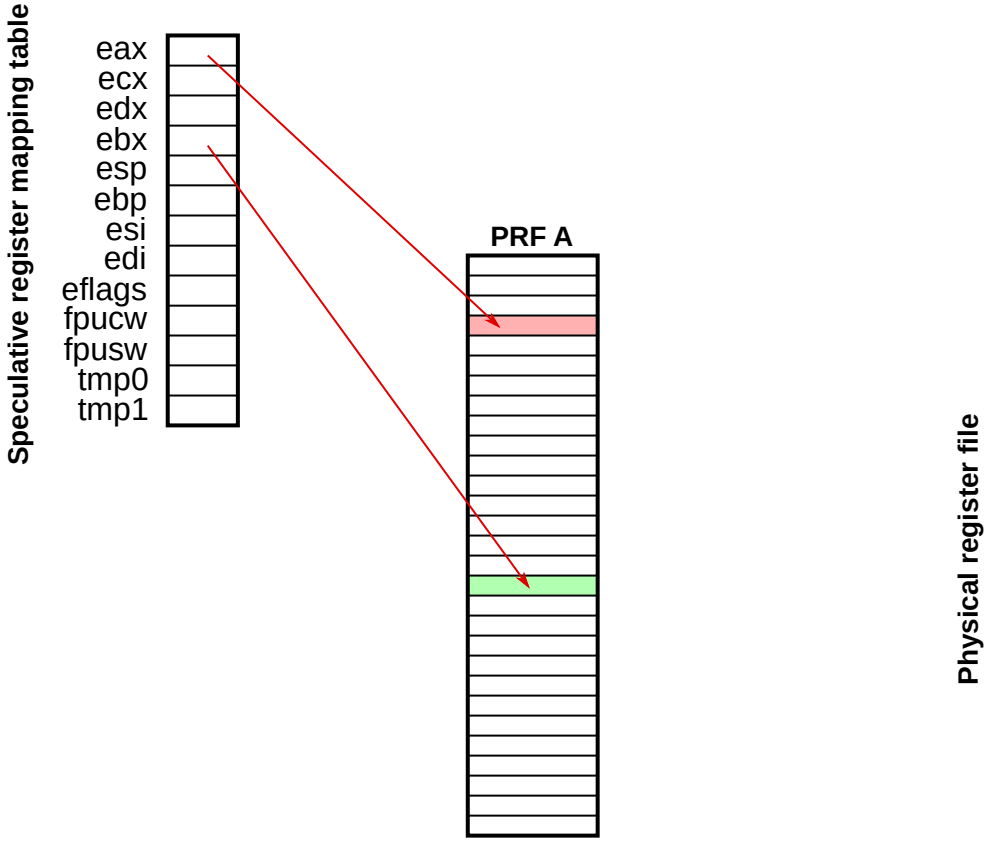



**x86**: Worst case is complex, but common case isn't too bad

ICache → **Bytes** → **Instructions** → **Micro-ops** → Renamer

# Register Renamer

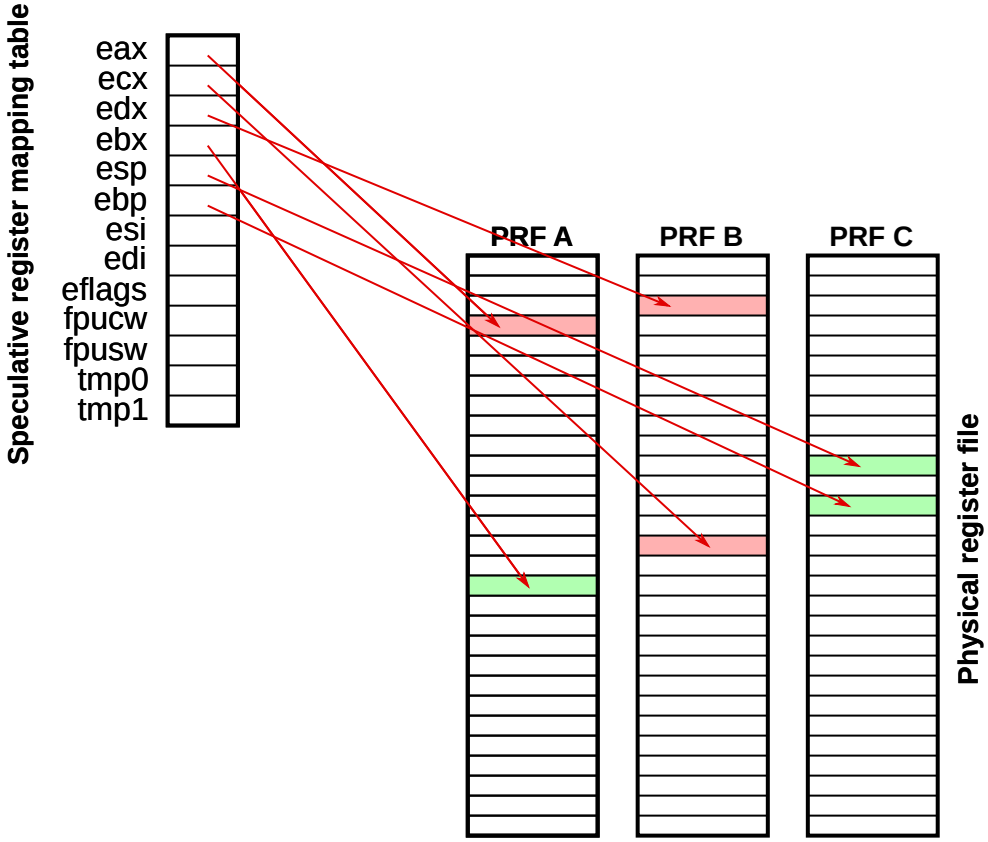
- **Read**: Map logical register numbers to physical
  - ~14 sources/clk
- **Write**: Update mapping table
  - ~4 destinations/clk




Map **logical reg** → **physical reg**, 2 uops/clock

# Register Renamer

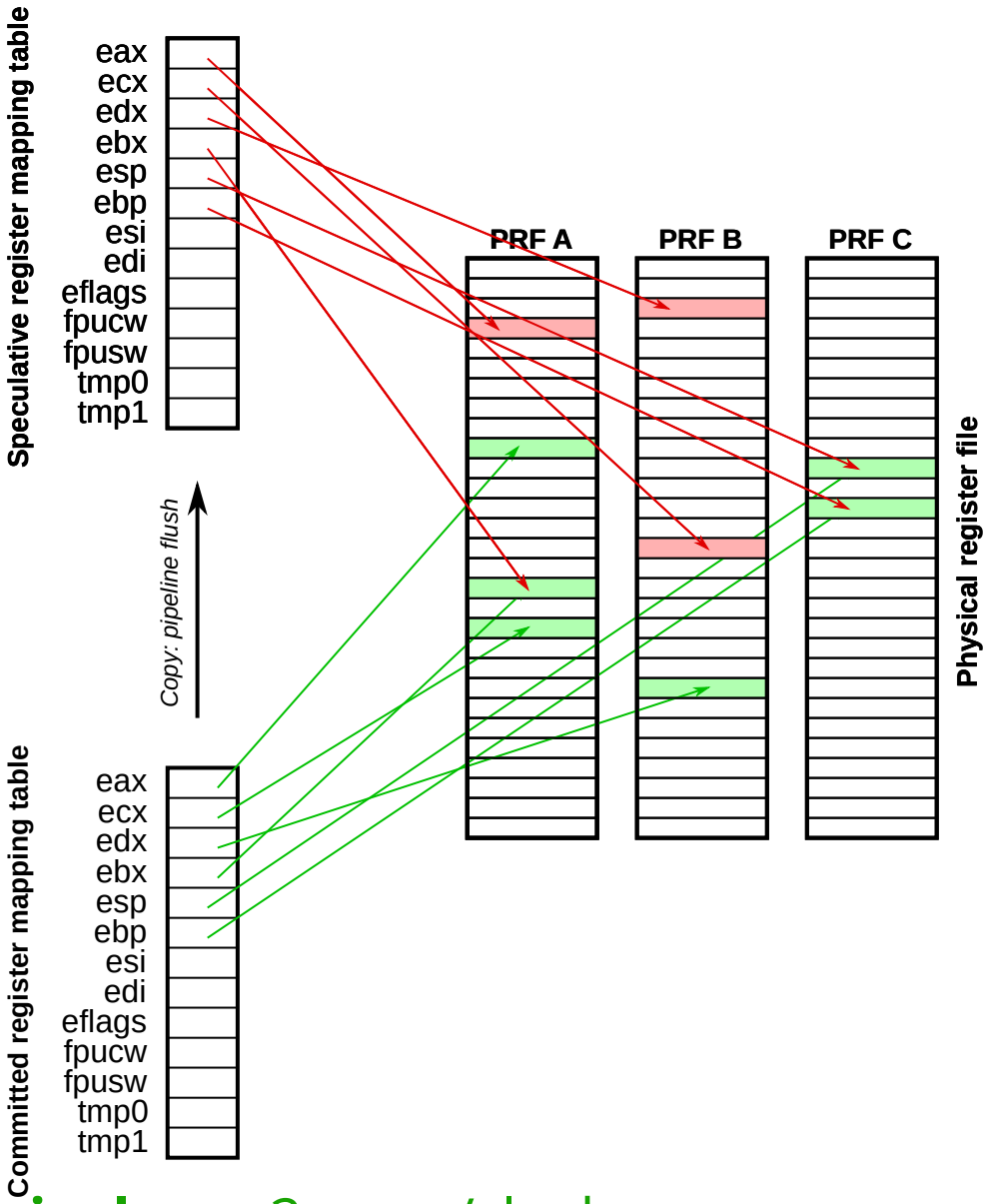
- **Read**: Map logical register numbers to physical
  - ~14 sources/clk
- **Write**: Update mapping table
  - ~4 destinations/clk
- Allows 1w-port reg. files
  - Each ALU “owns” an RF




Map **logical reg** → **physical reg**, 2 uops/clock

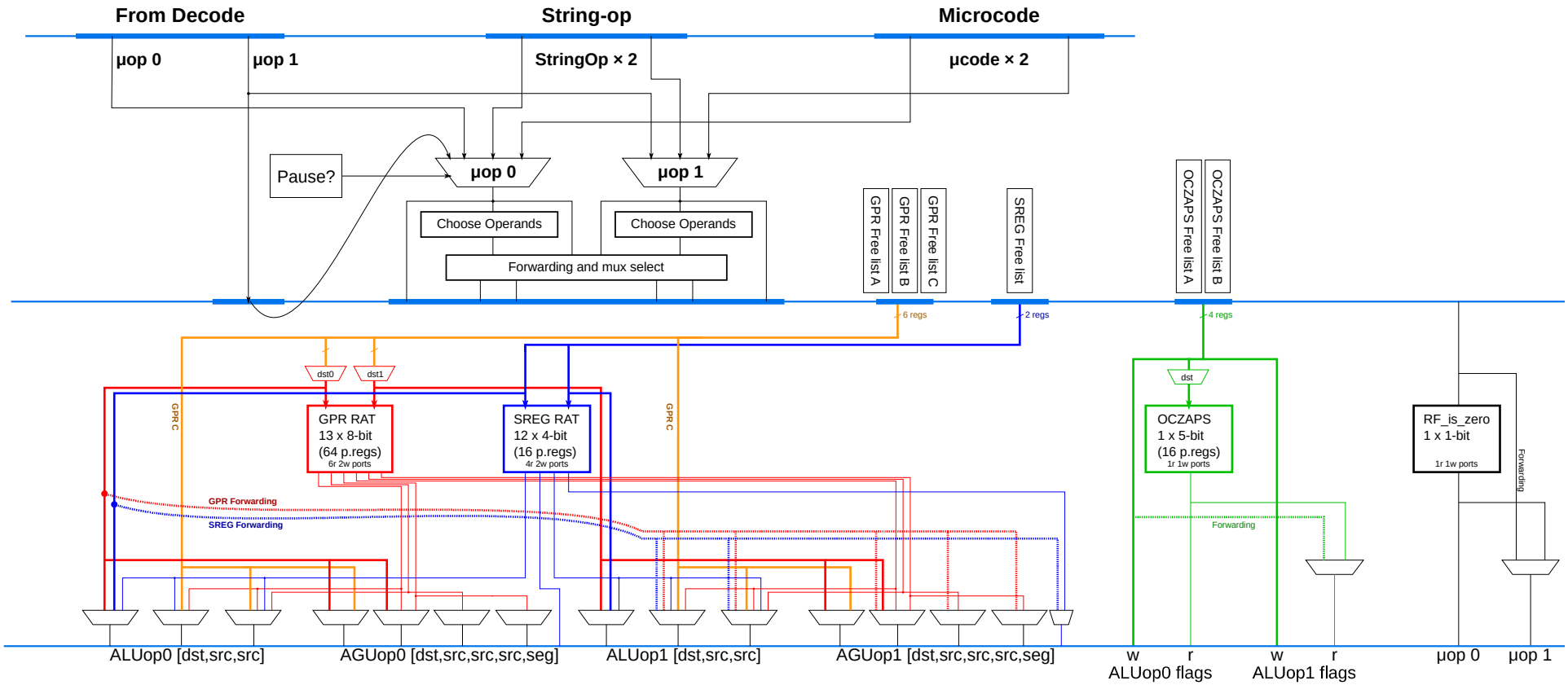
# Register Renamer

- **Read**: Map logical register numbers to physical
  - ~14 sources/clk
- **Write**: Update mapping table
  - ~4 destinations/clk
- Allows 1w-port reg. files
  - Each ALU "owns" an RF
- Used for recovery from misspeculation



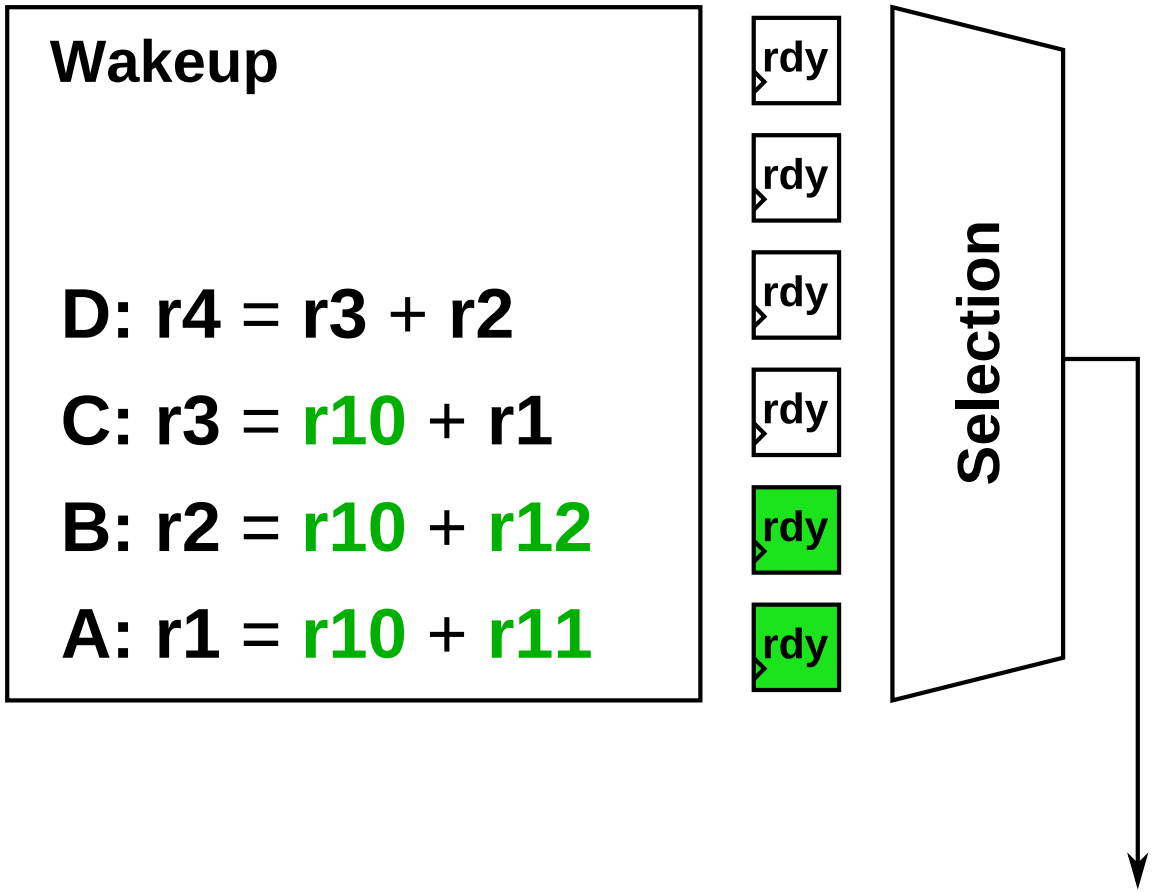

Map **logical reg** → **physical reg**, 2 uops/clock

# Renamer Circuit



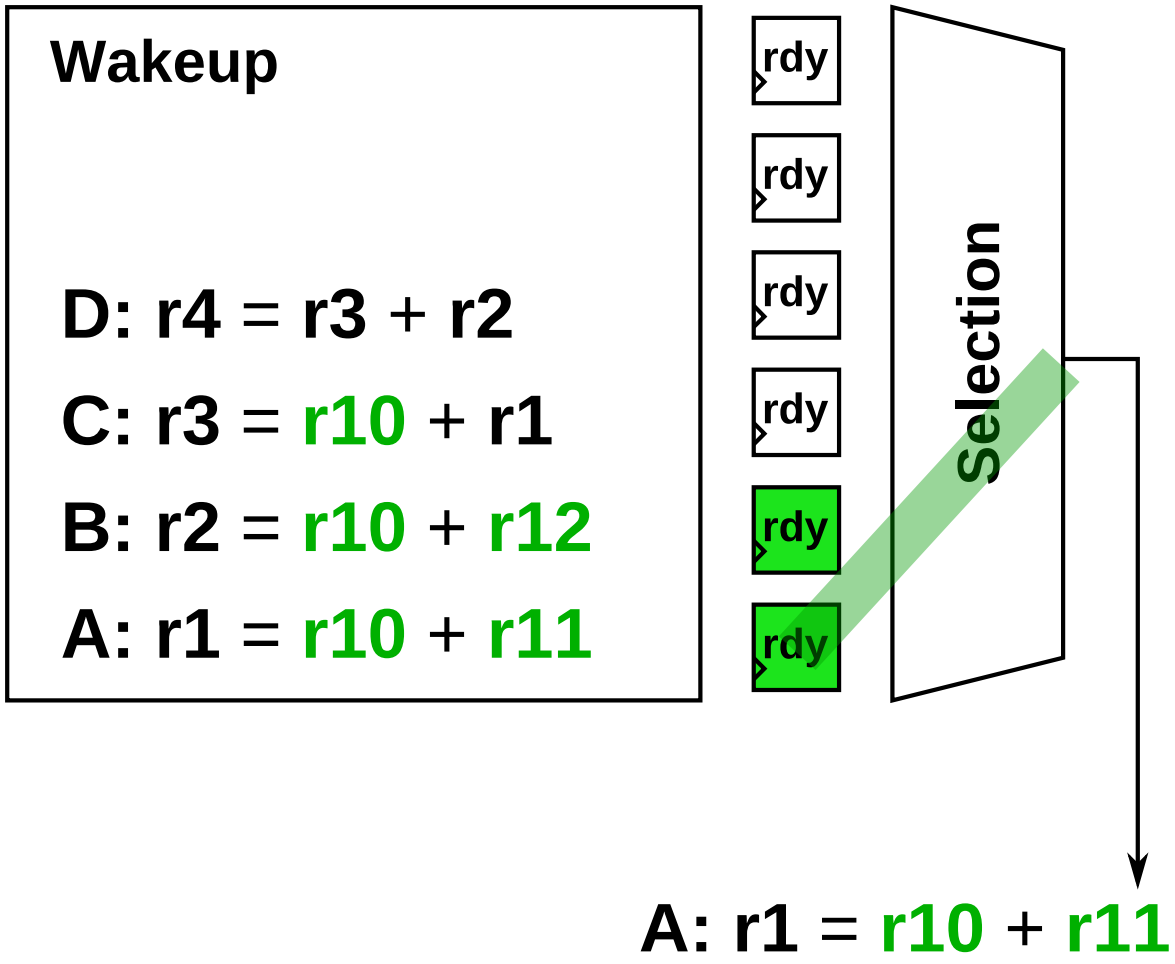

- Stage 1: Pick two uops; find where each operand comes from
- Stage 2: A bunch of read muxes; write destination regs to RAT
- 317 MHz, 1900 ALMs
- **x86**: Few registers: small FF-based RAT. But ≥3 register types.

# Scheduling: Track dependencies



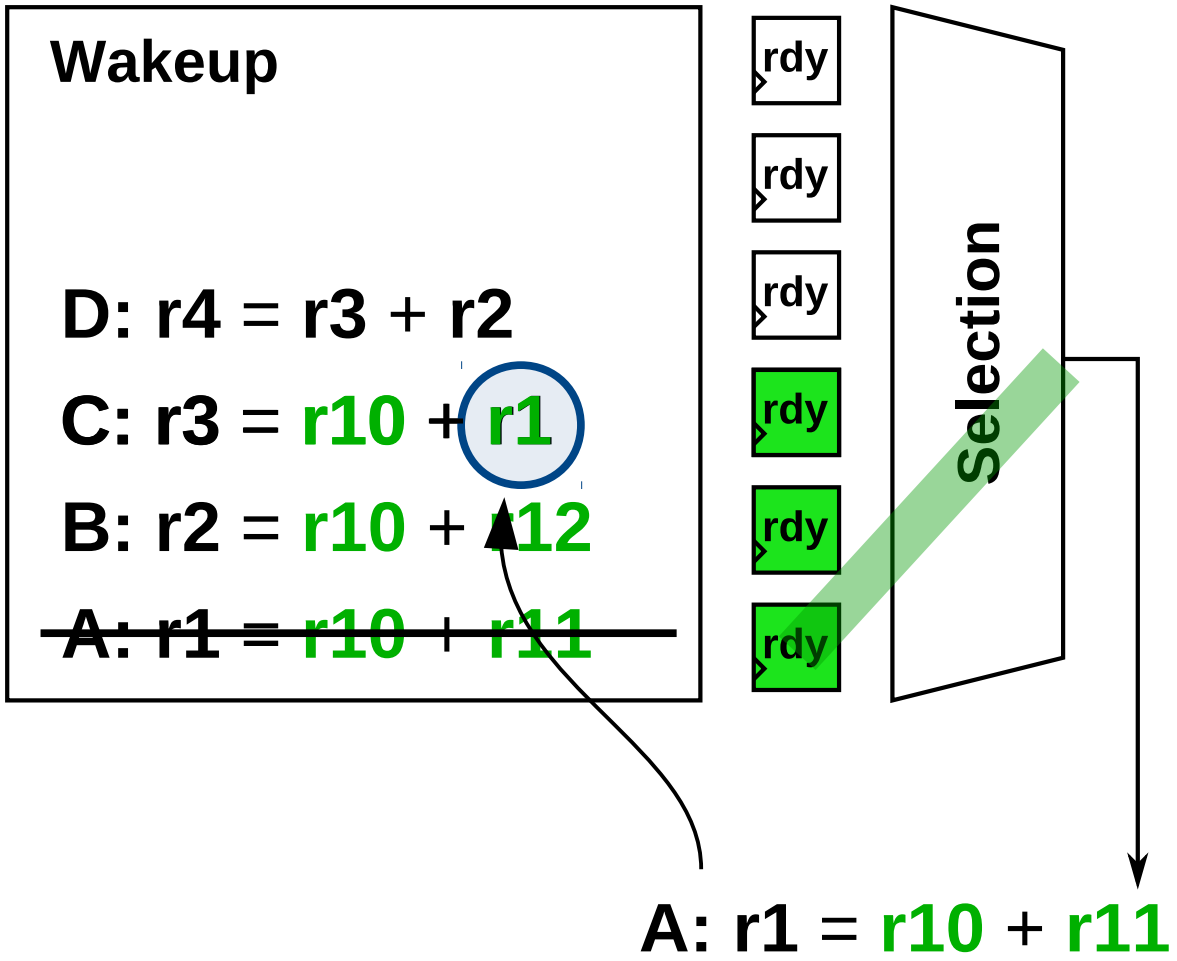

- **Pick** a ready operation, **execute**, and **wake up** dependents

# Scheduling: Track dependencies



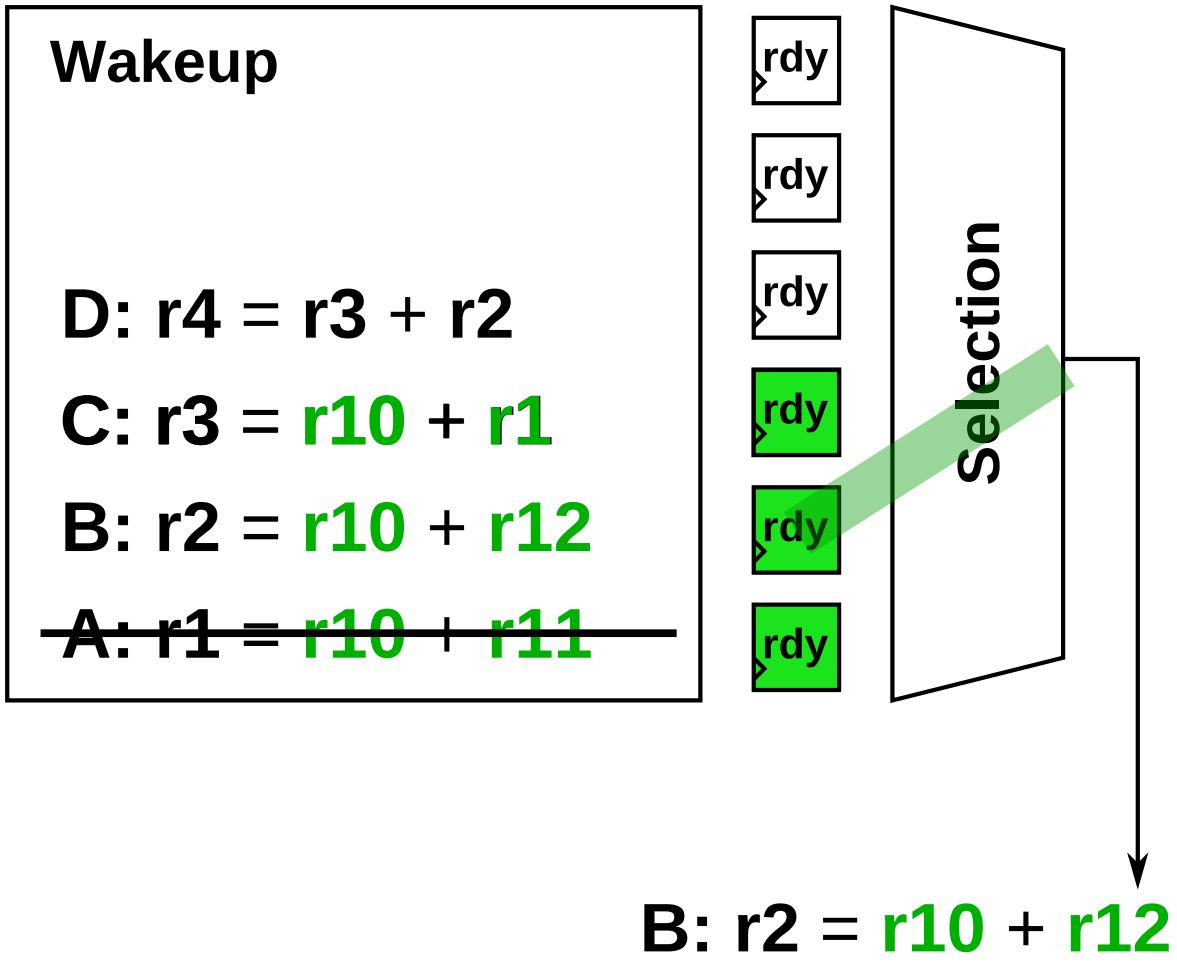

- **Pick** a ready operation, **execute**, and **wake up** dependents

# Scheduling: Track dependencies



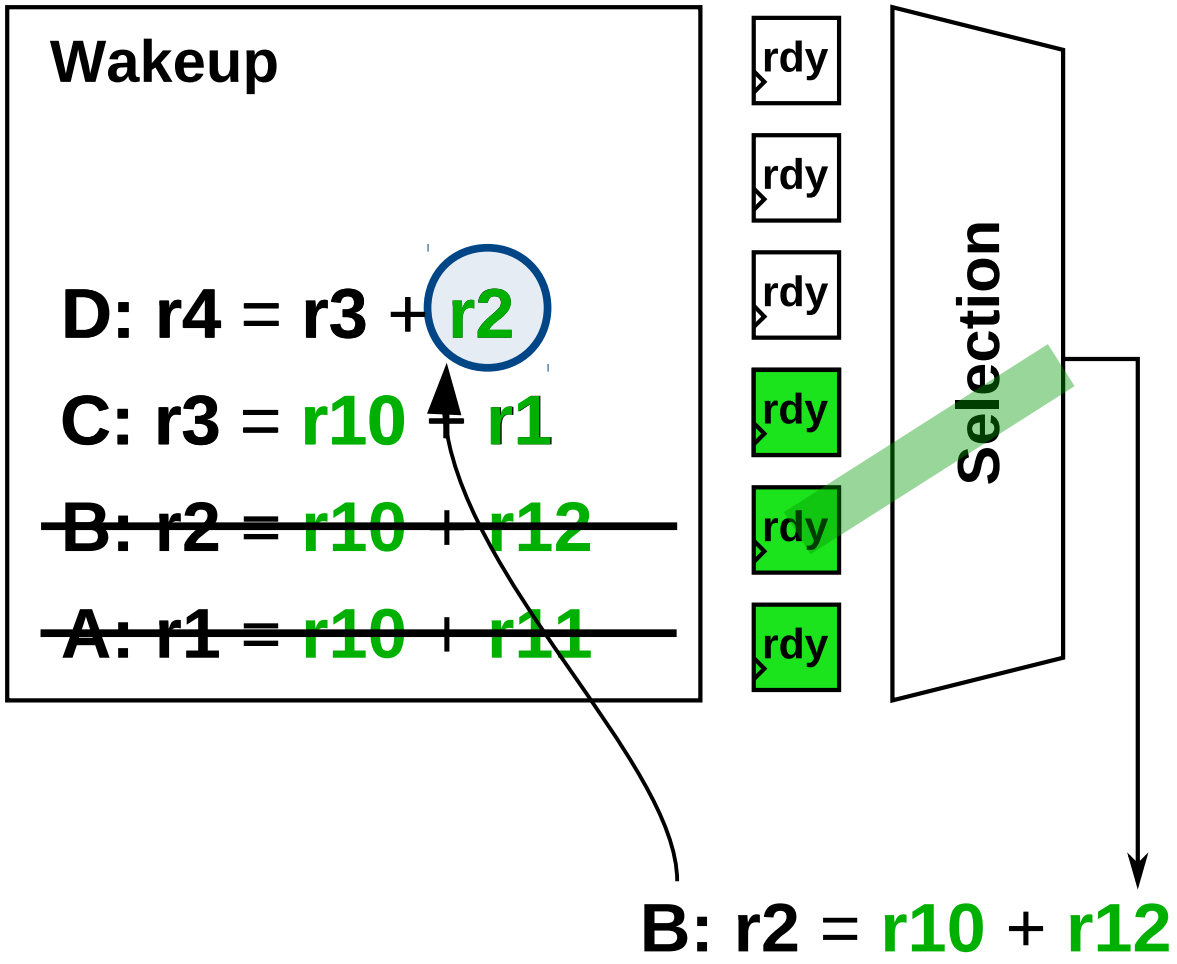

- **Pick** a ready operation, **execute**, and **wake up** dependents

# Scheduling: Track dependencies



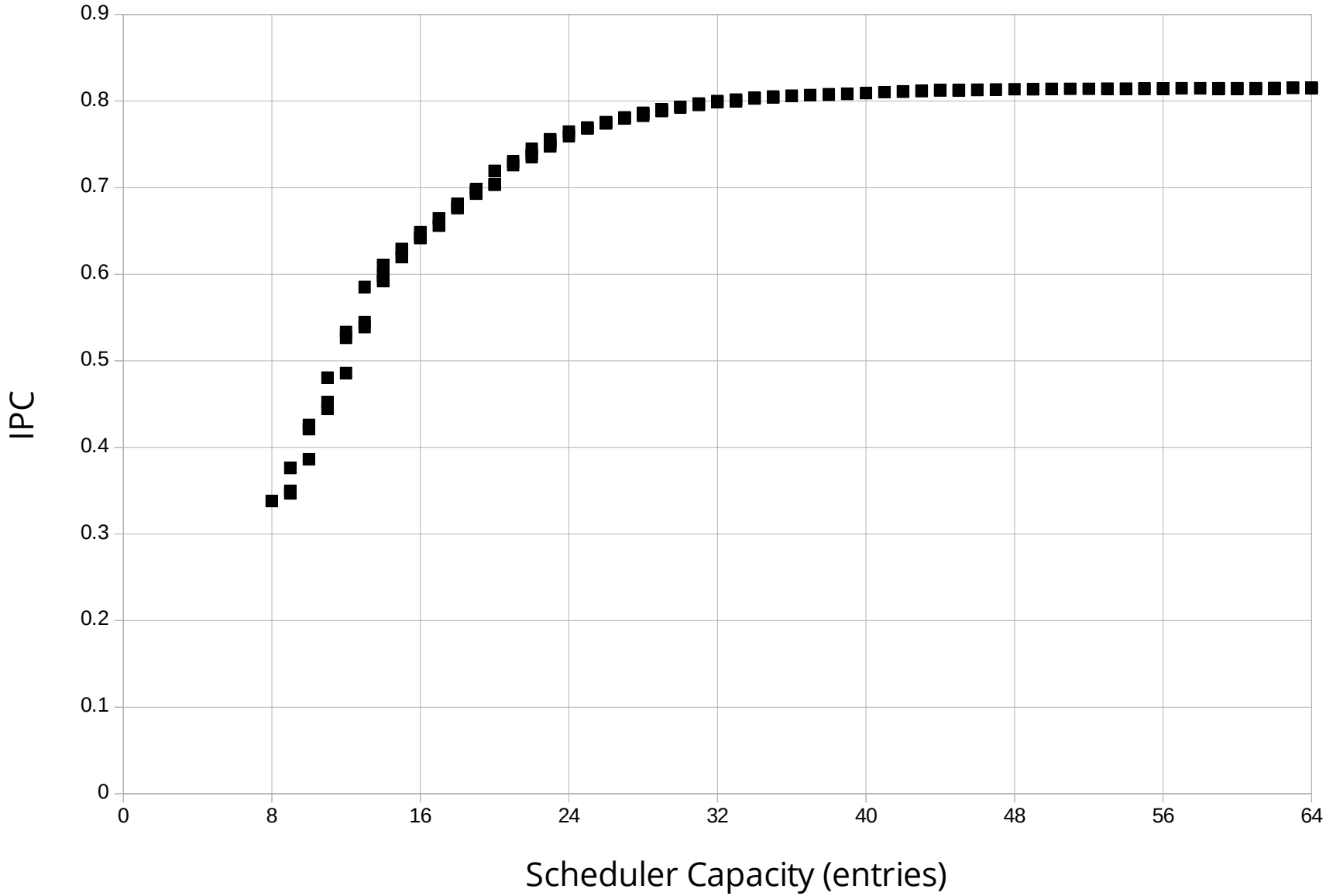

- **Pick** a ready operation, **execute**, and **wake up** dependents

# Scheduling: Track dependencies



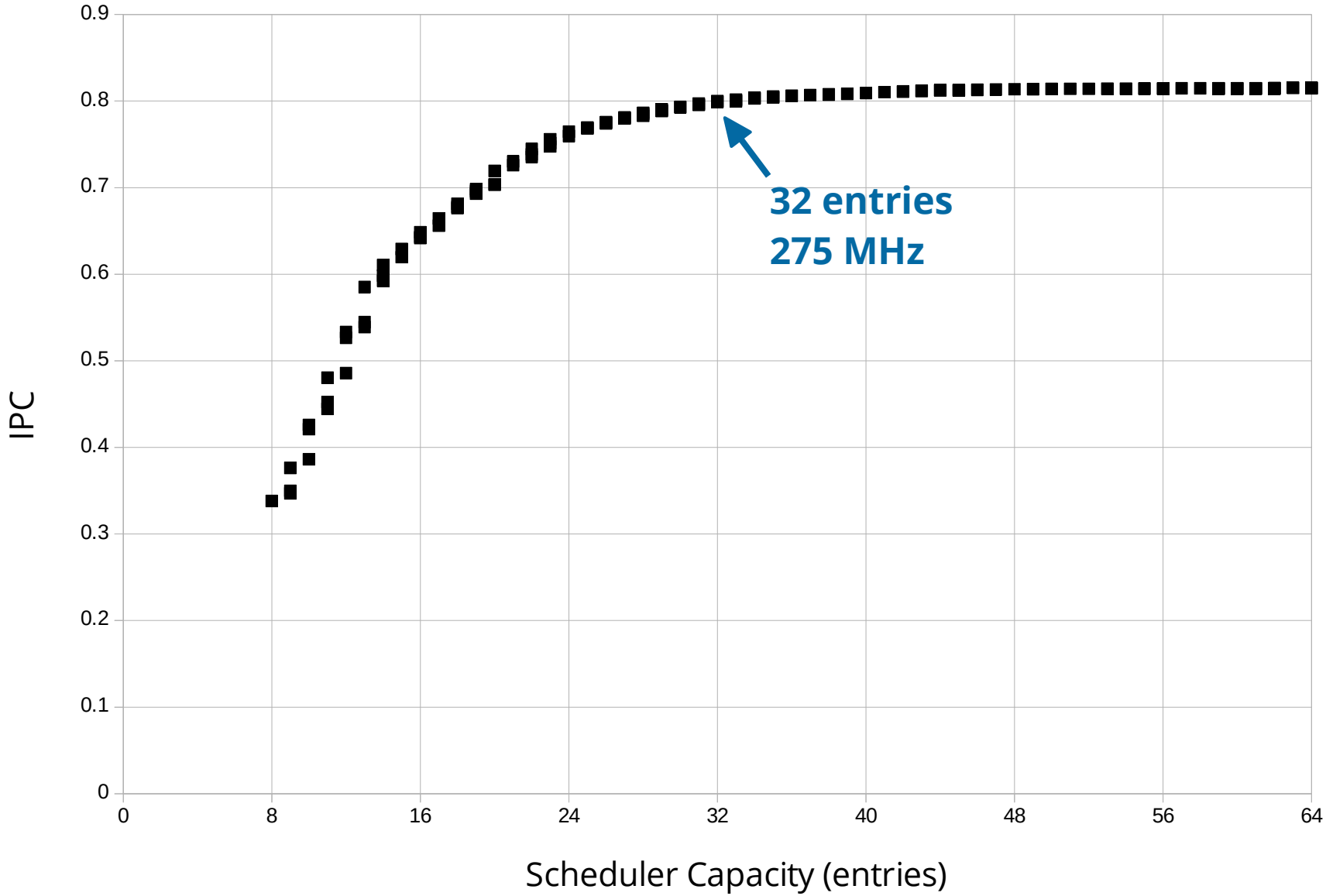

- **Pick** a ready operation, **execute**, and **wake up** dependents

# Scheduler Size




- Can trade capacity (area and frequency) for IPC

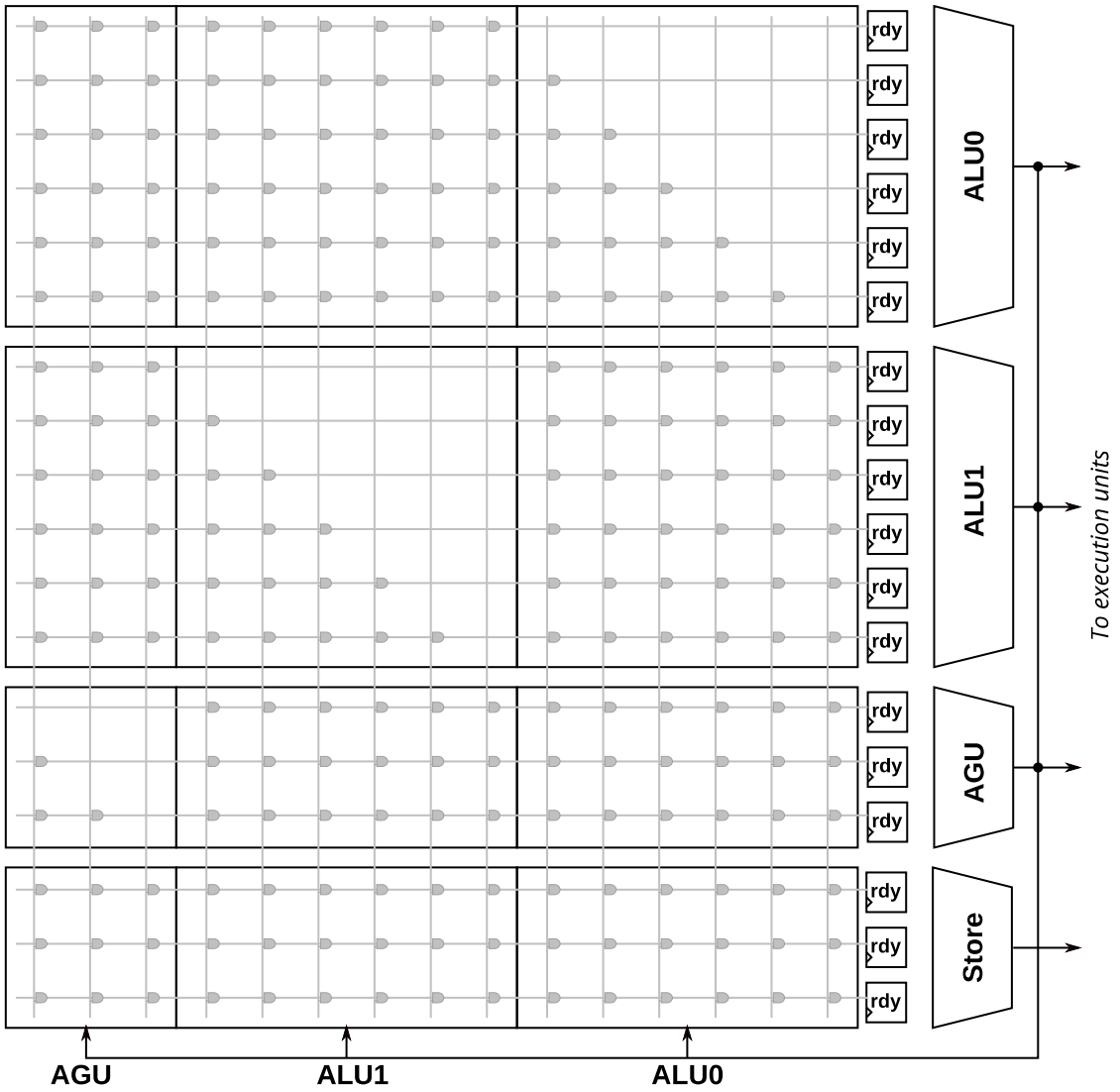
# Scheduler Size




- Can trade capacity (area and frequency) for IPC

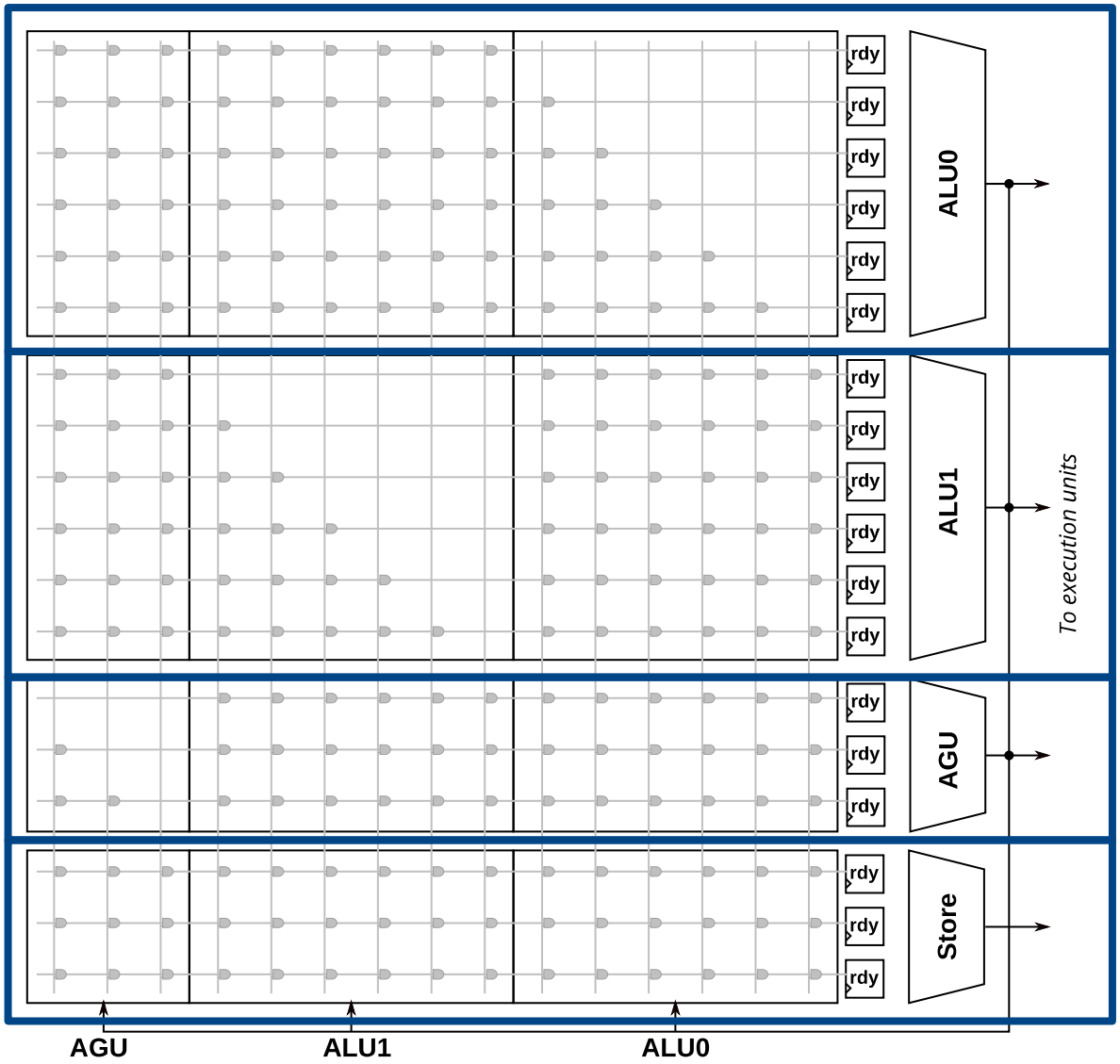
# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97



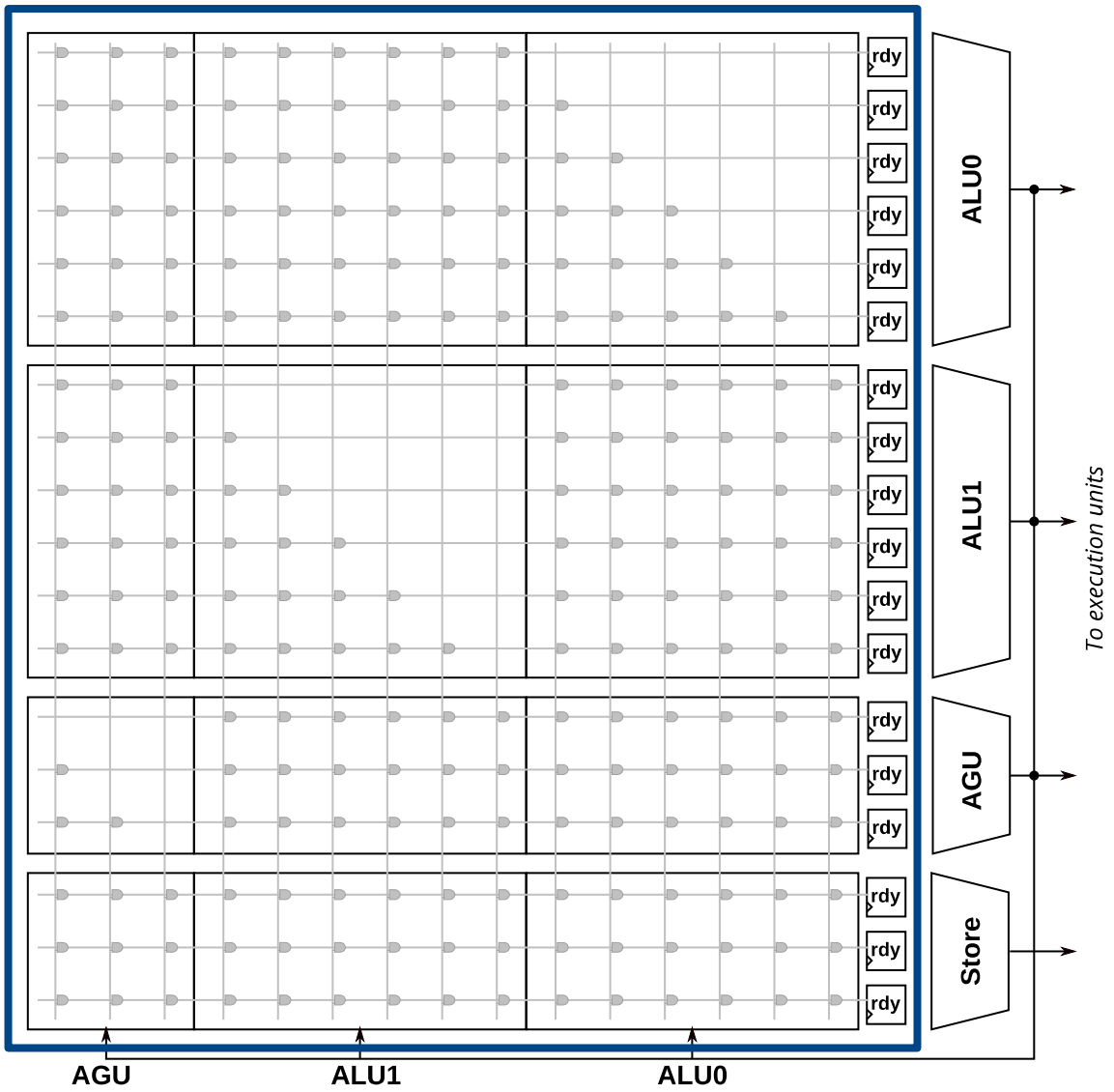
# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97



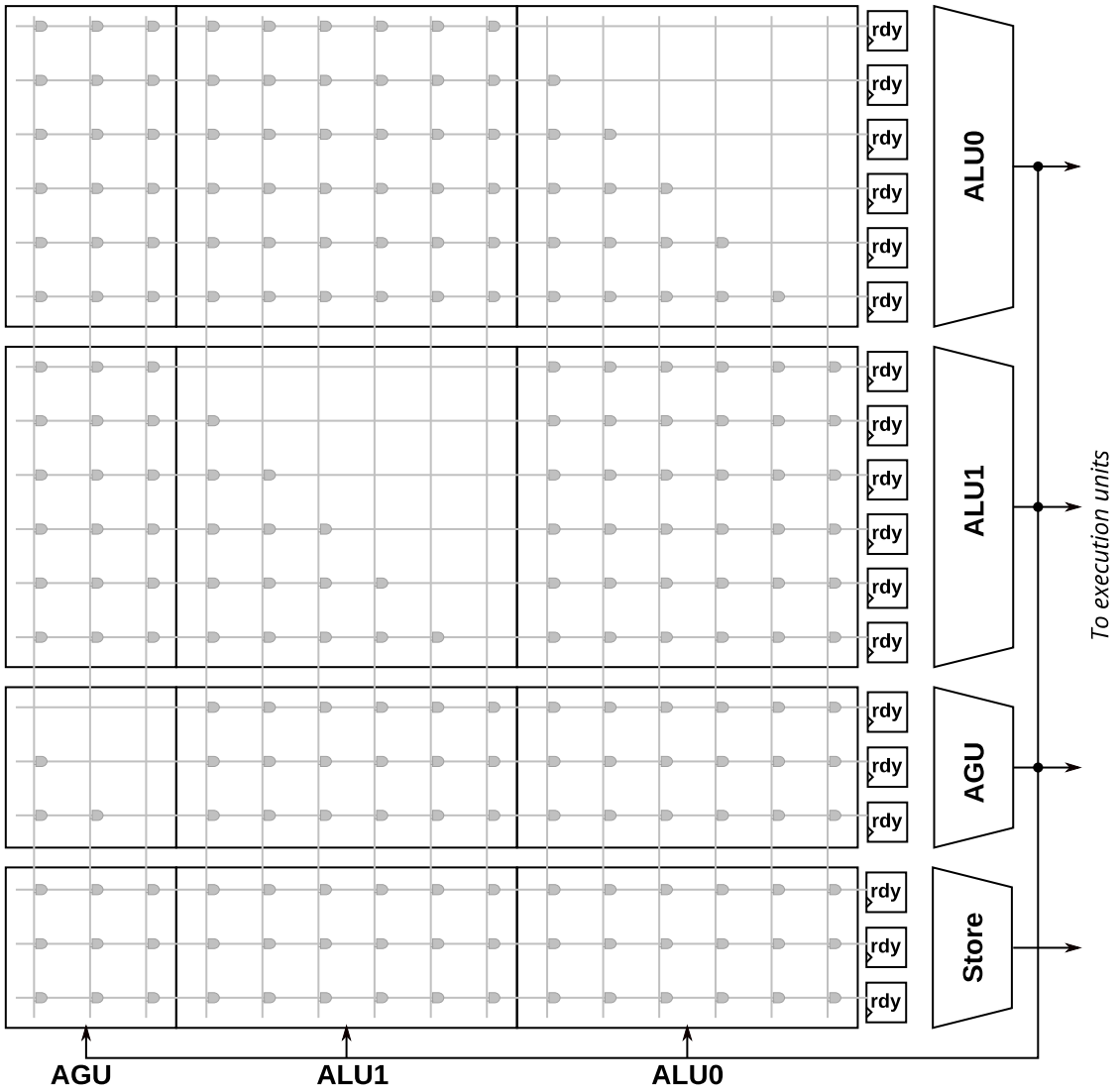
# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97



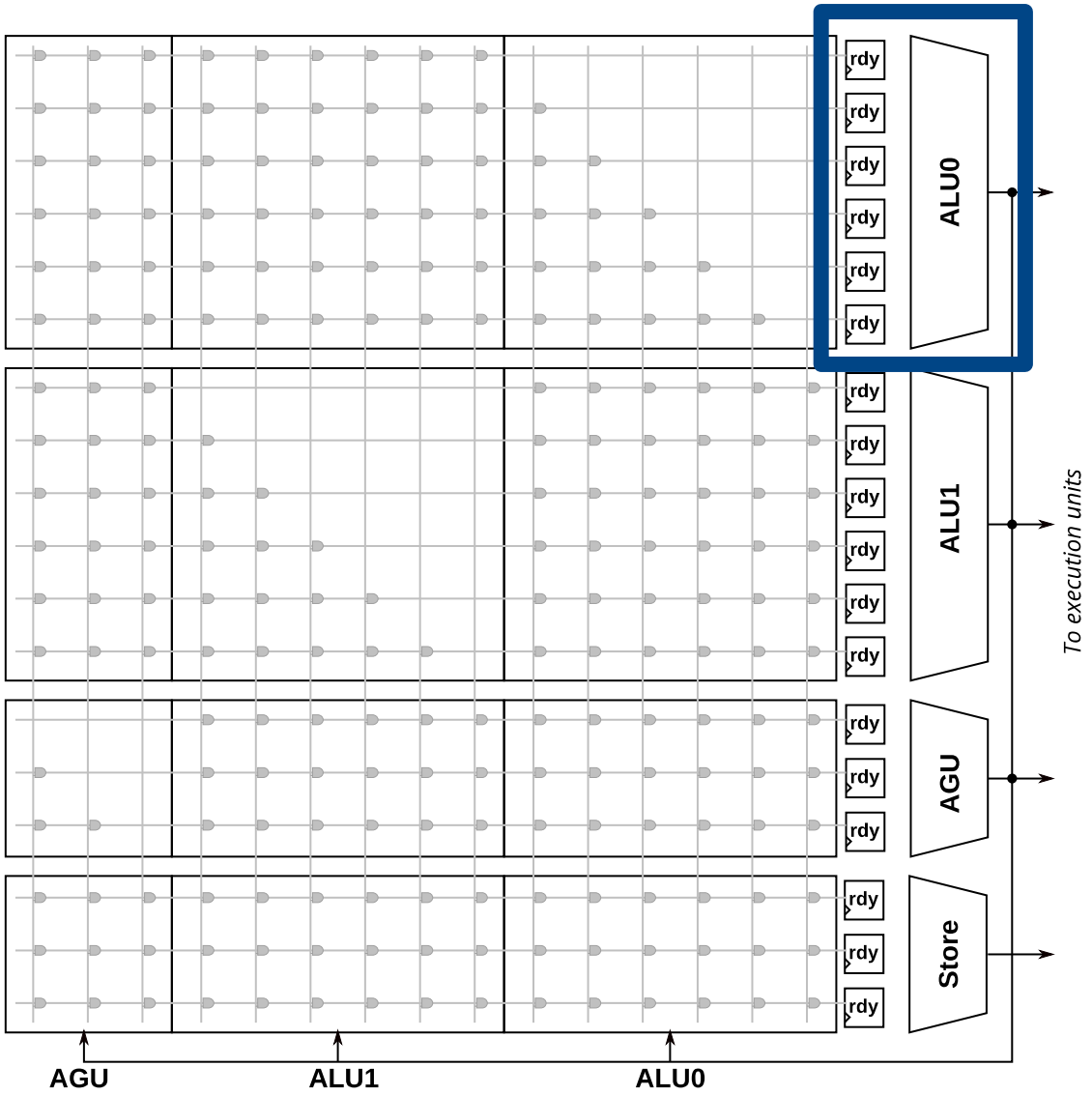
# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97



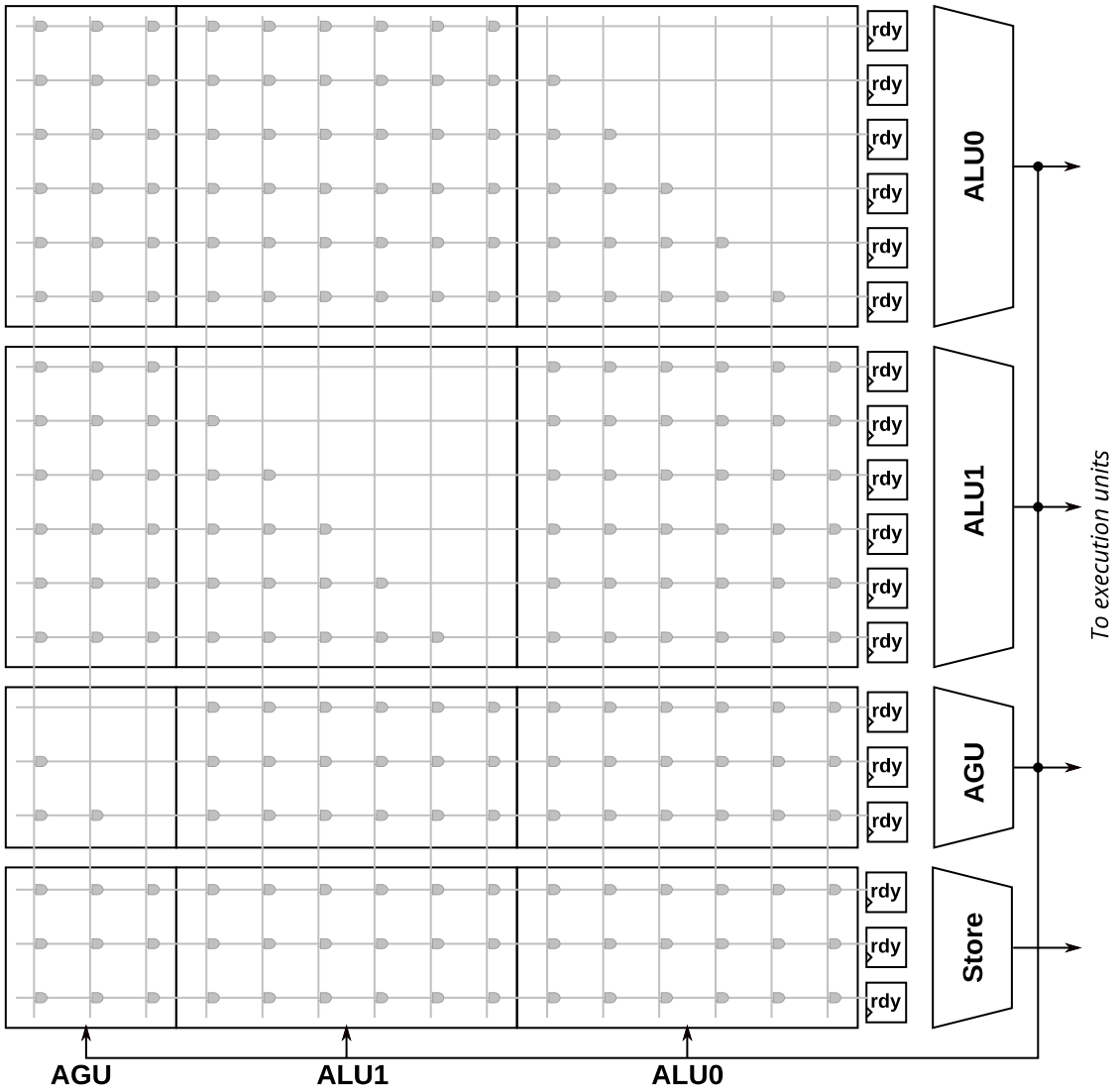
# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97

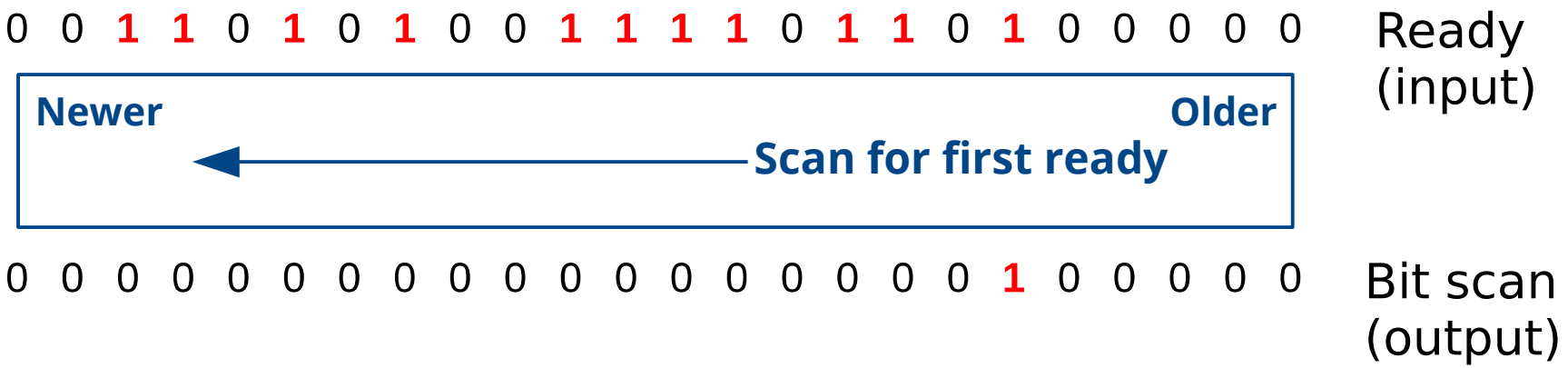


# Scheduler Circuit

- 4-way distributed matrix scheduler
- 32-entry (10, 10, 7, 5)
  - 275 MHz
- Comparison:
  - Pentium Pro: 20
  - Haswell: 60
  - Ryzen: 84 (6×14)
  - Skylake: 97

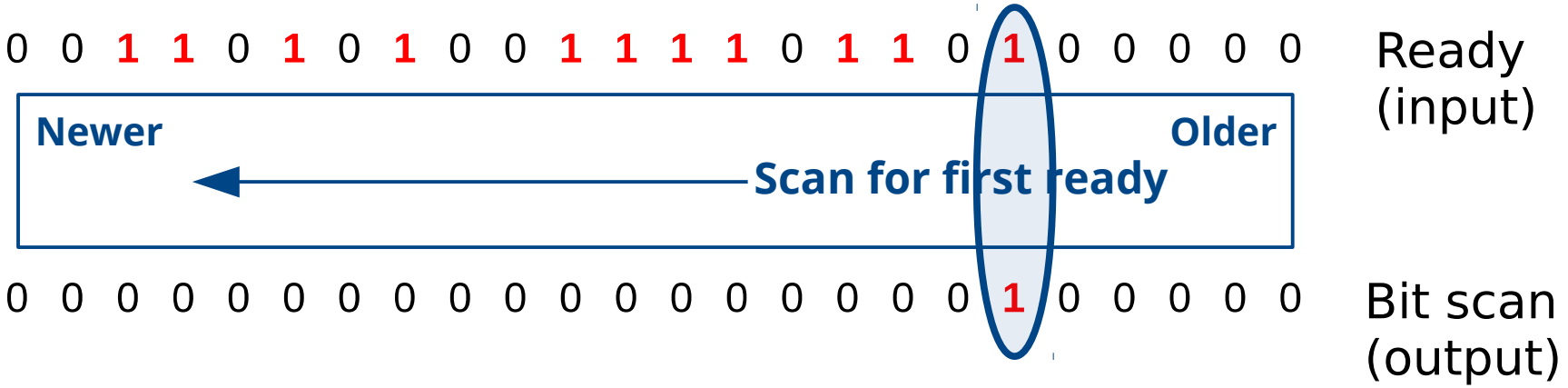


# Scheduler Picker: Bit Scan



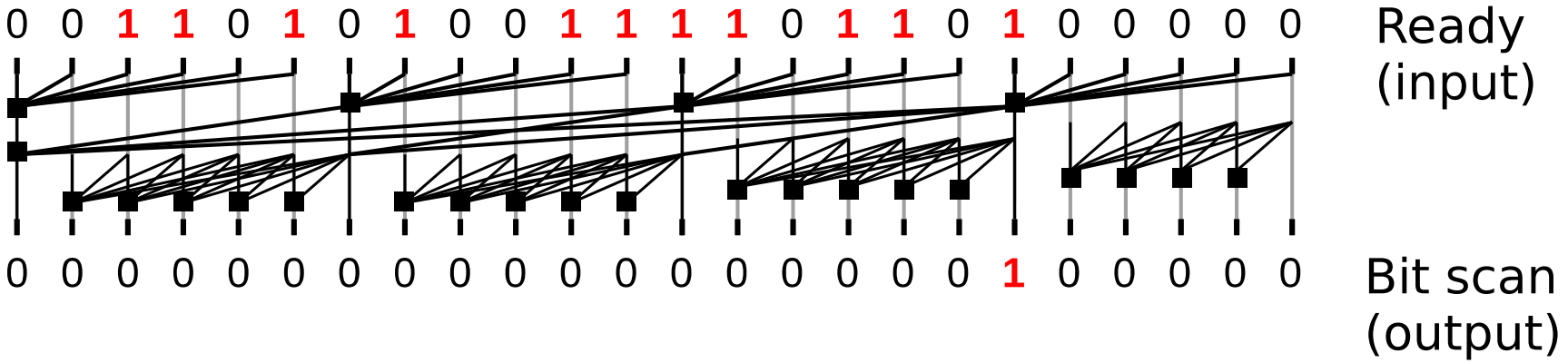

- Pick first ready instruction to execute

# Scheduler Picker: Bit Scan




- Pick first ready instruction to execute

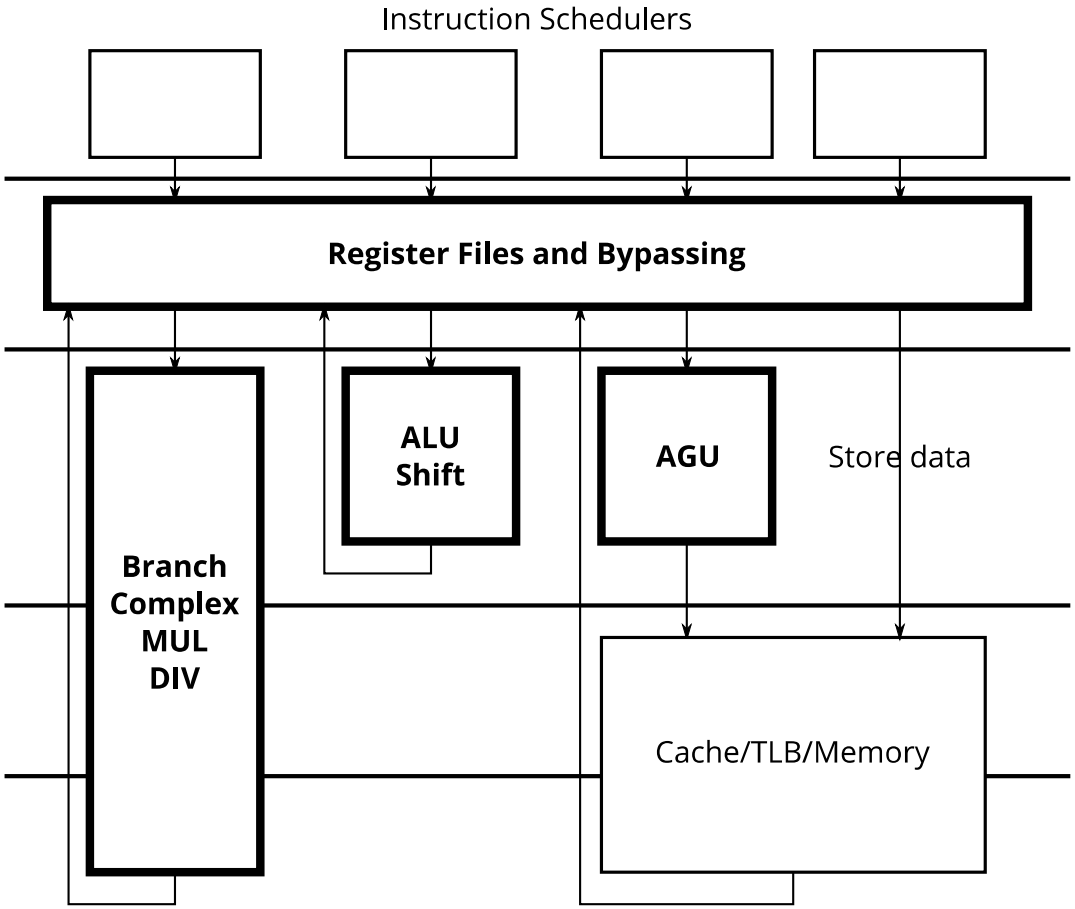
# Scheduler Picker: Bit Scan




- Pick first ready instruction to execute
- Logarithmic depth: radix-6
  - Han-Carlson prefix tree
  - Very difficult to code: Synthesizer makes it linear depth again

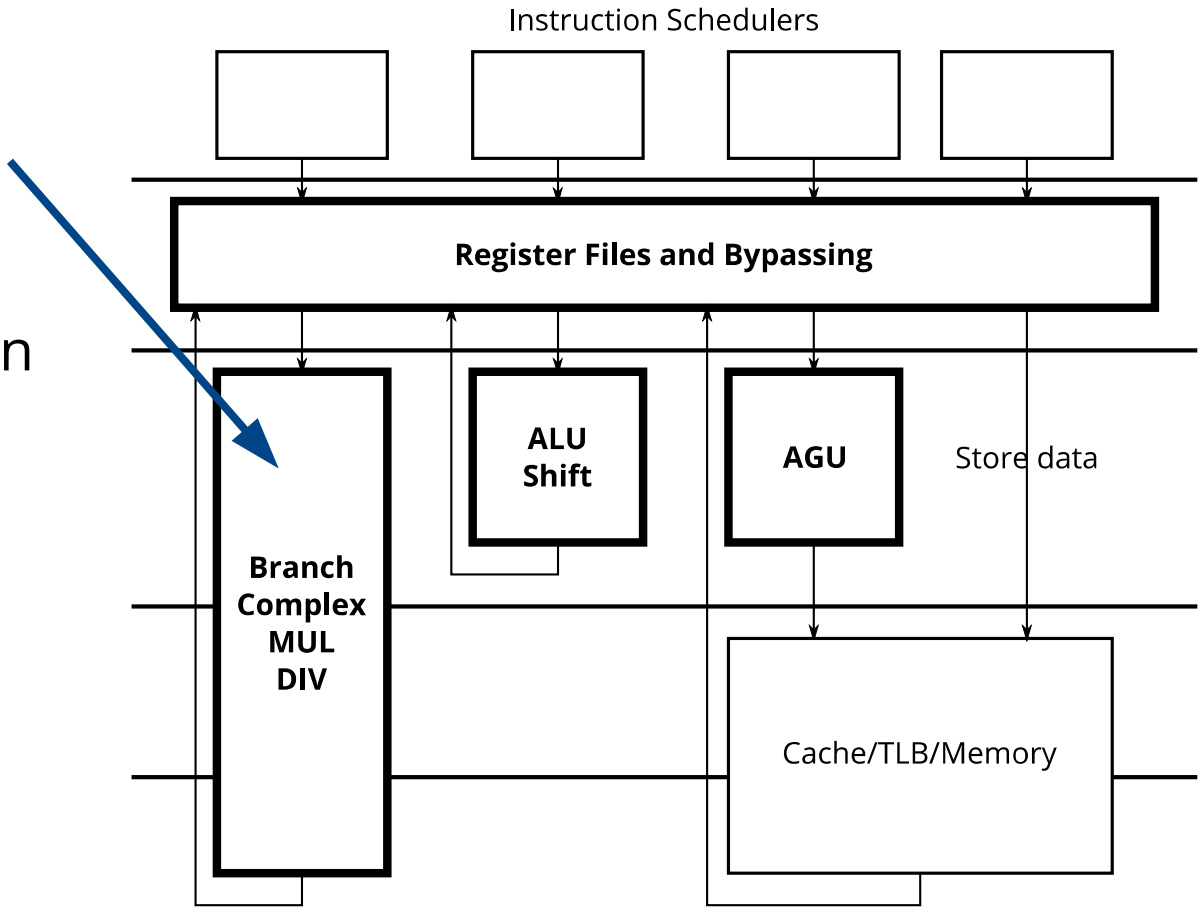
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



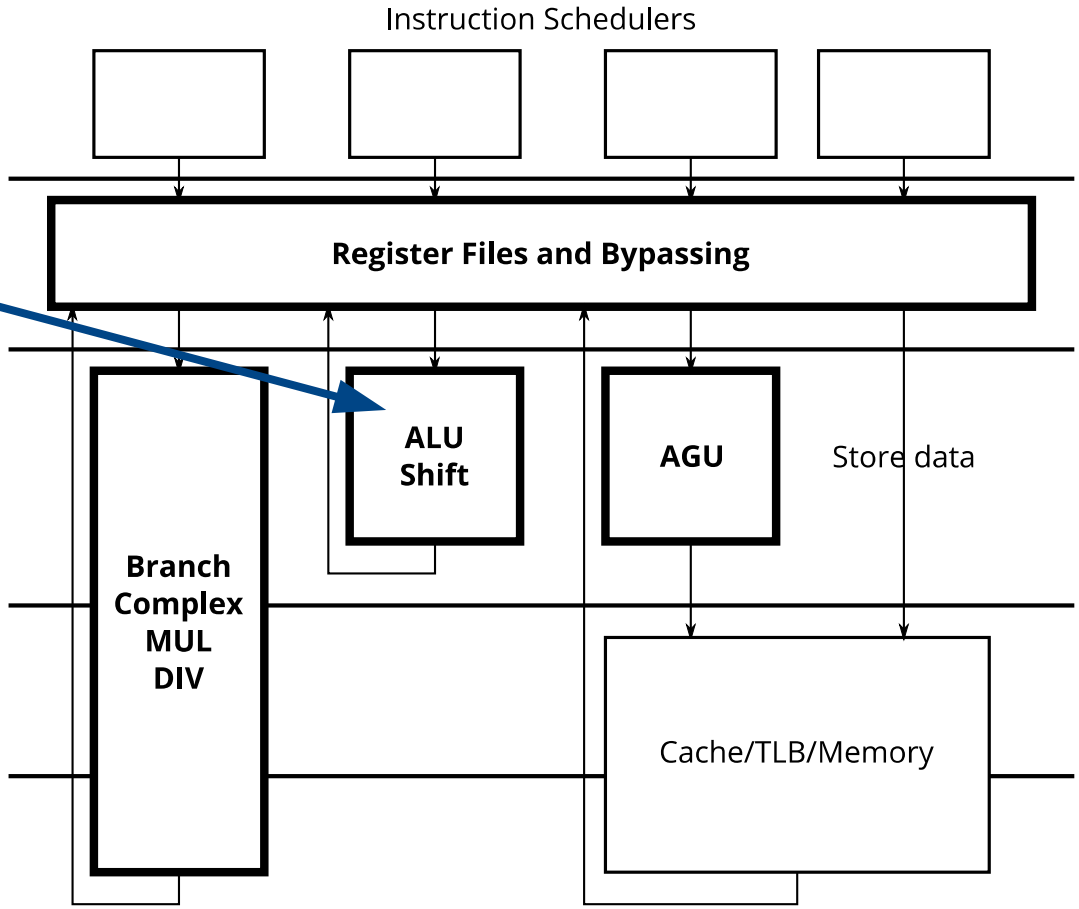
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



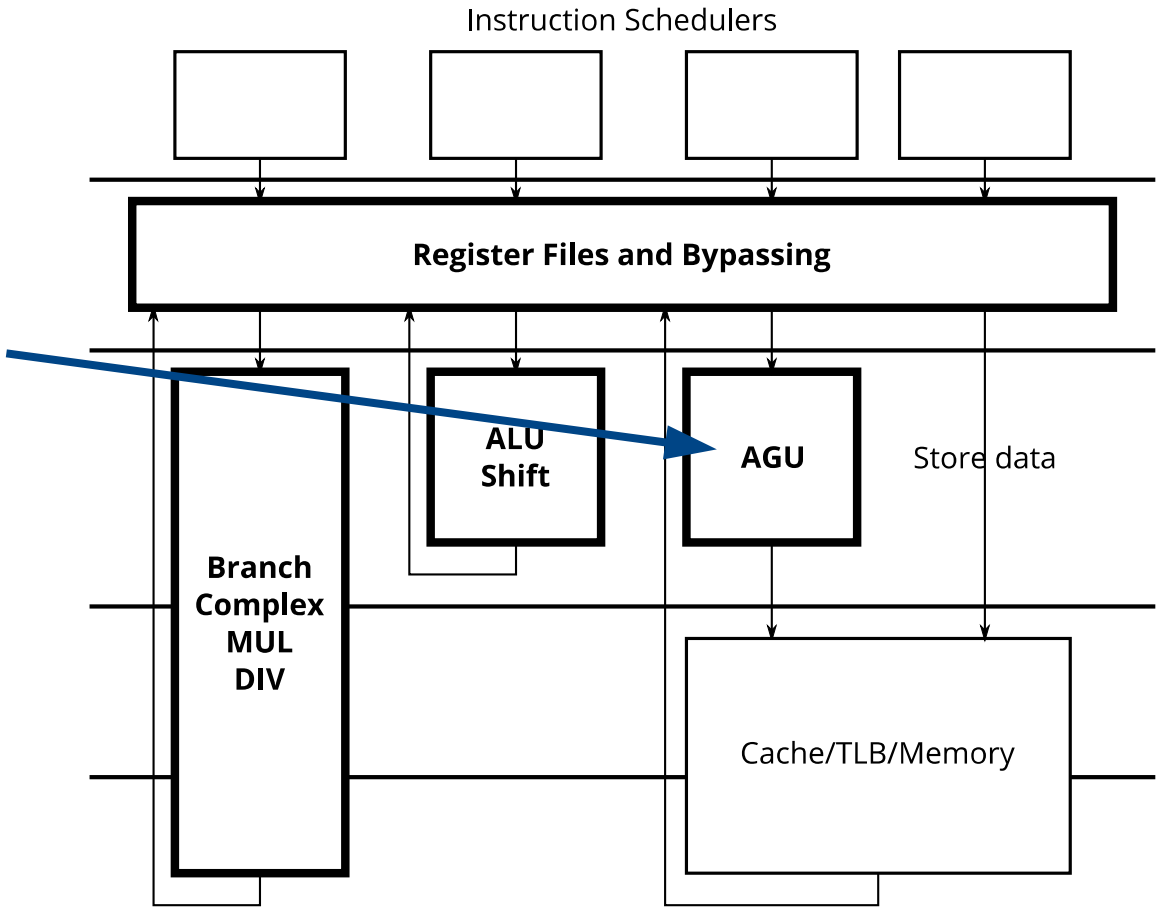
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



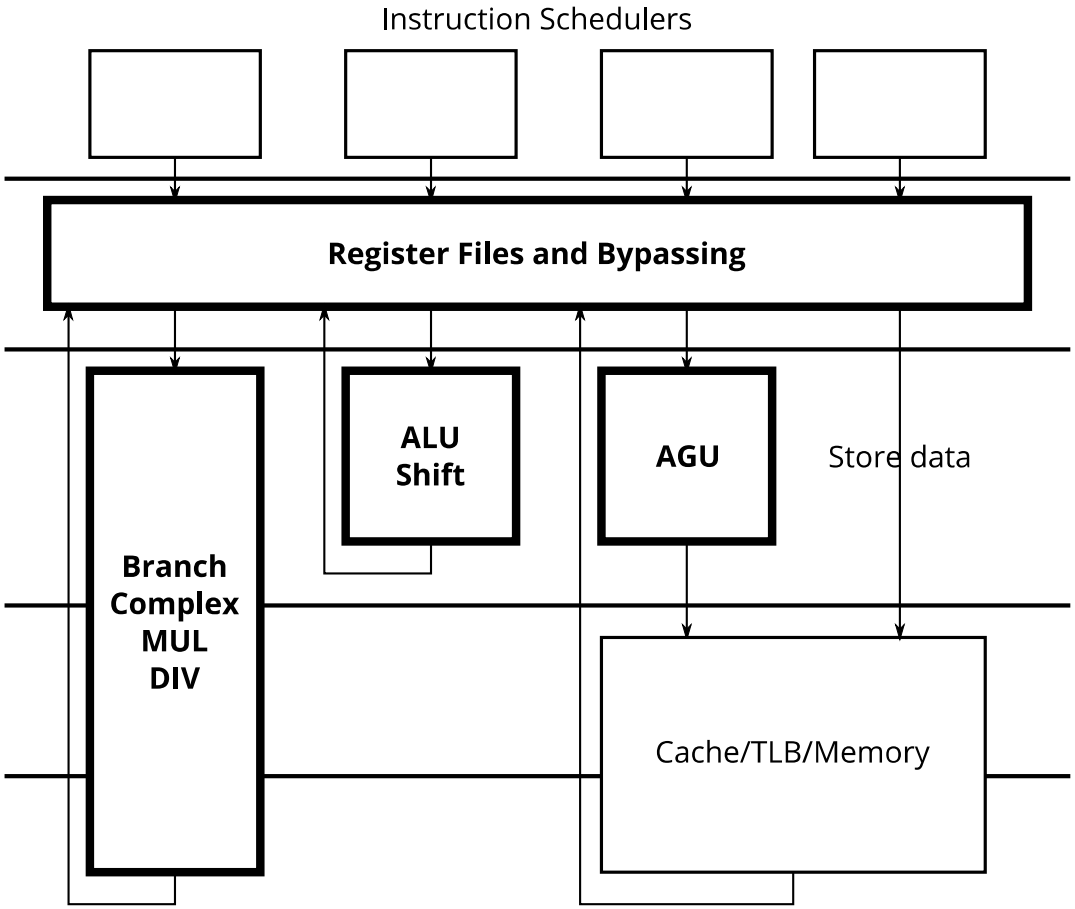
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



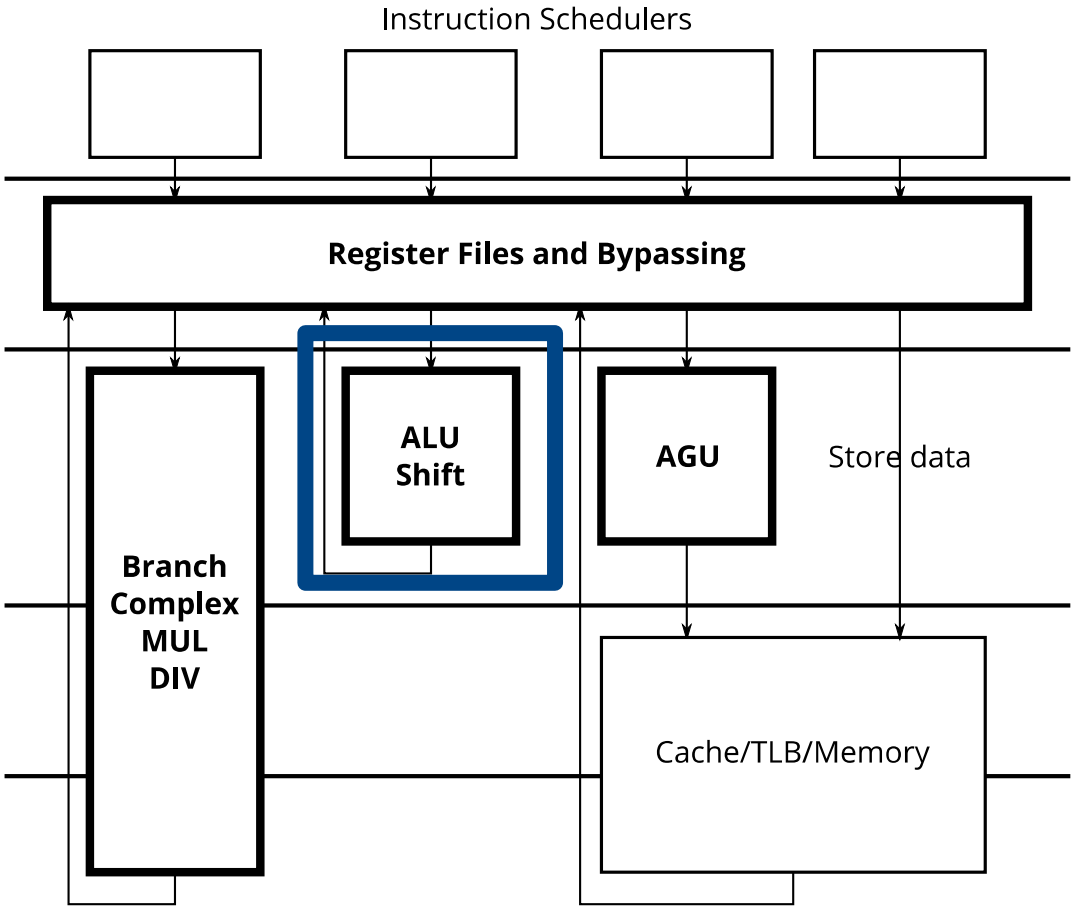
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



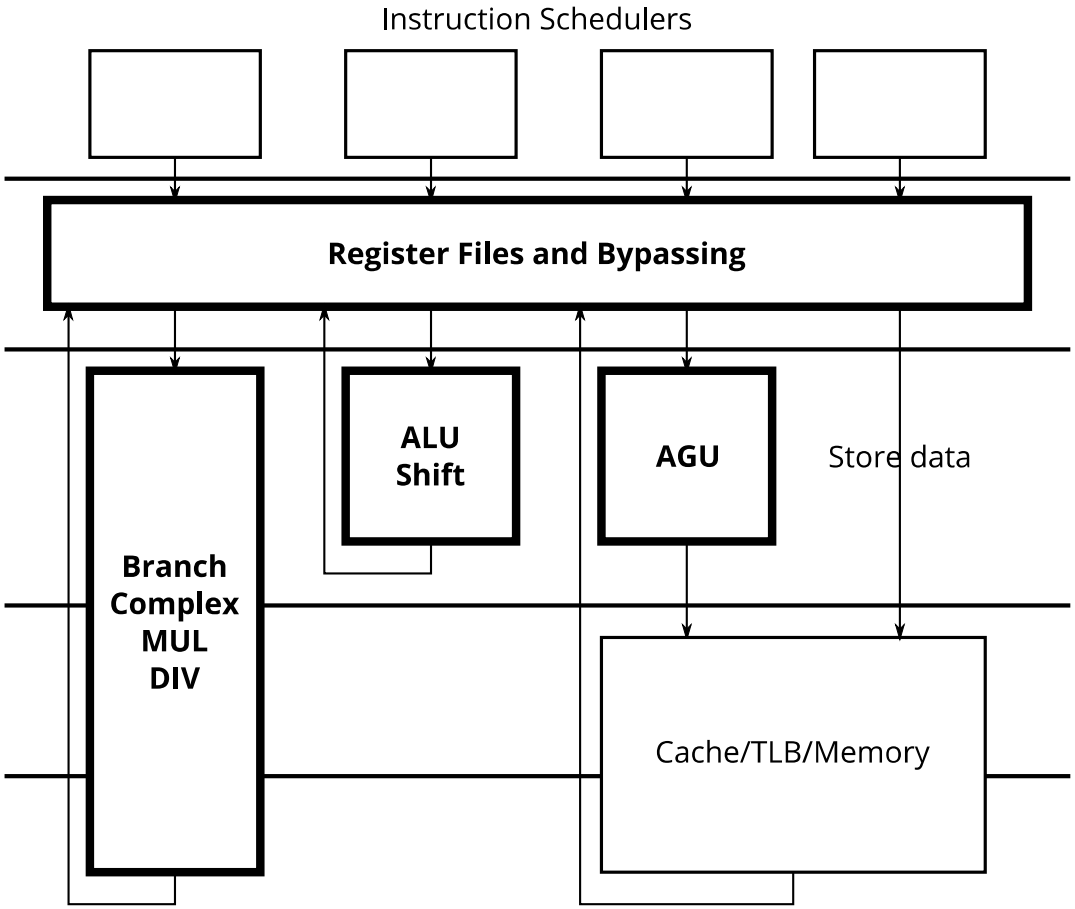
# Execution

- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



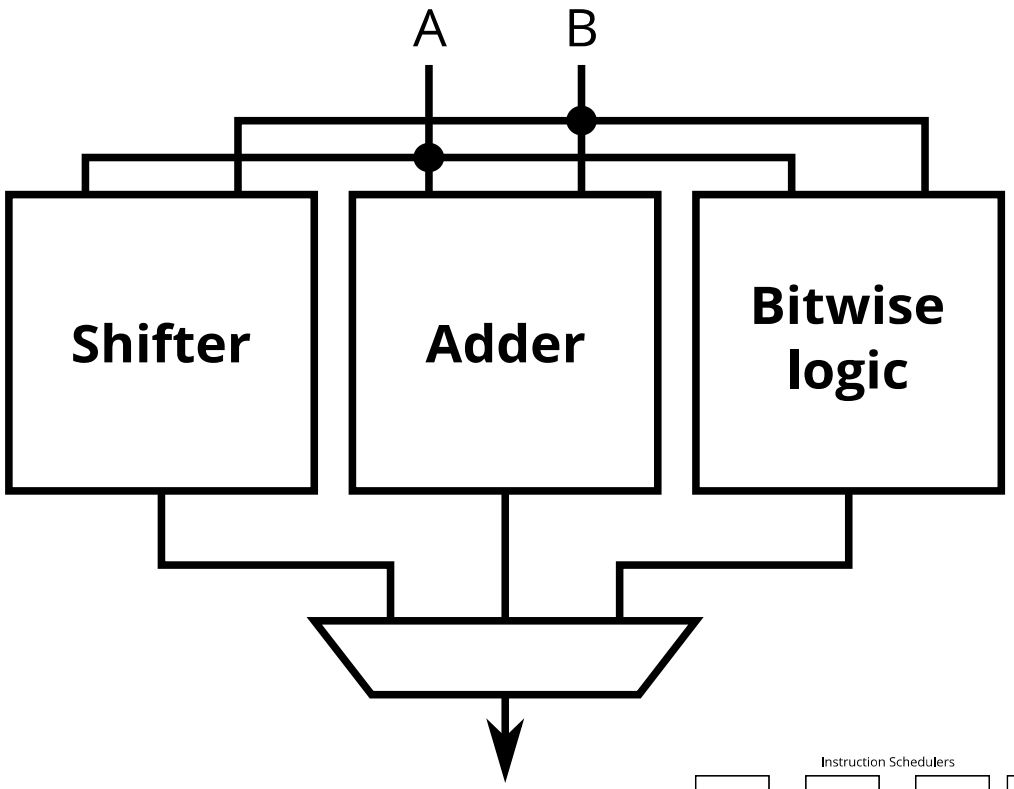
# Execution

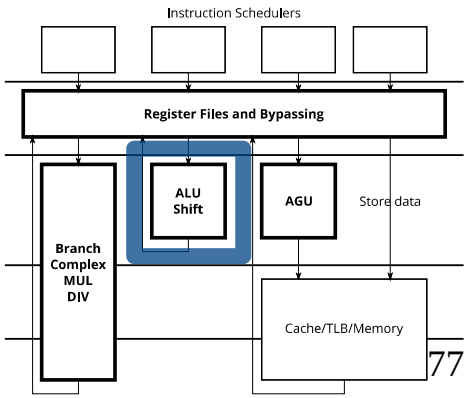
- Three different execution units
  - Complex, 3+ cycle
  - Simple, 1 cycle
  - Address generation
- Latency vs. delay circuit design problem



# Execution Circuits: Simple ALU

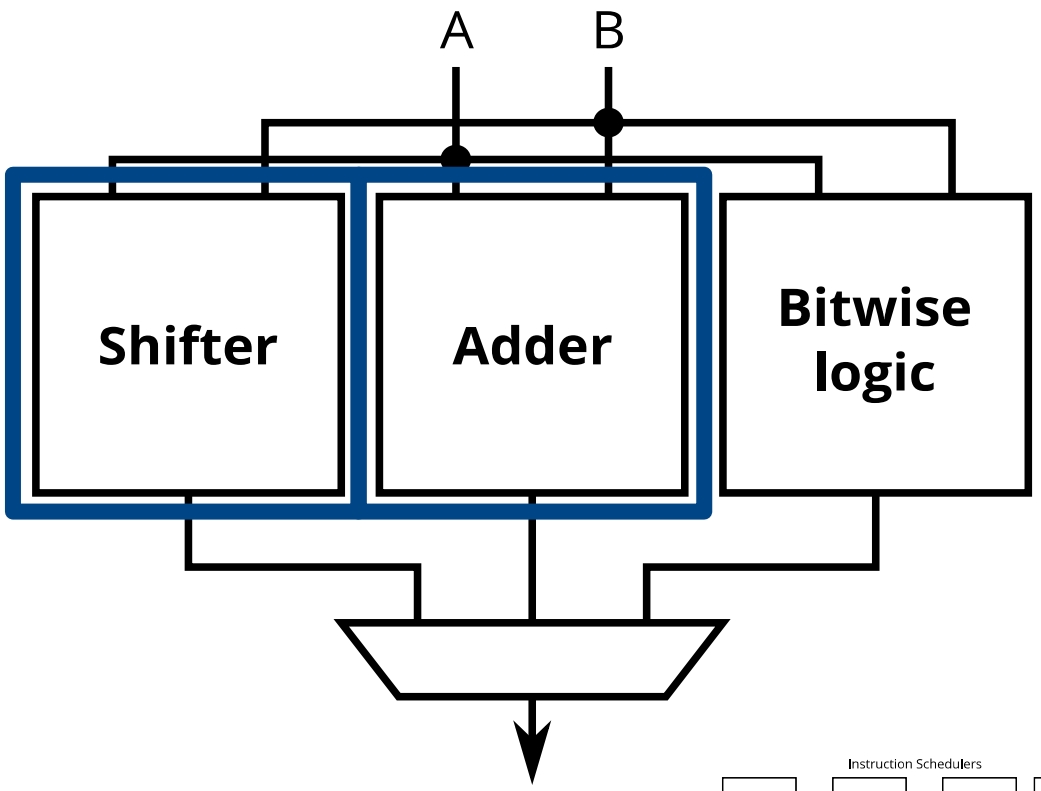
- Three parts:
  - Shifter
  - Adder
  - Bitwise logic
- We'll look at shifter and adder circuits

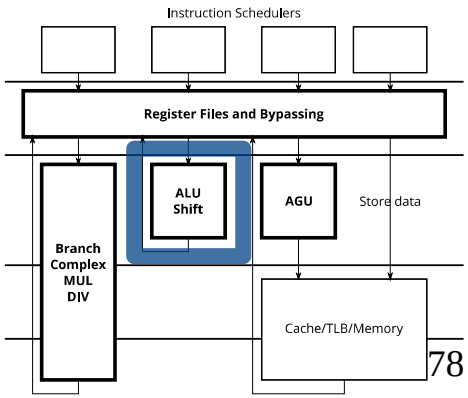


# Execution Circuits: Simple ALU

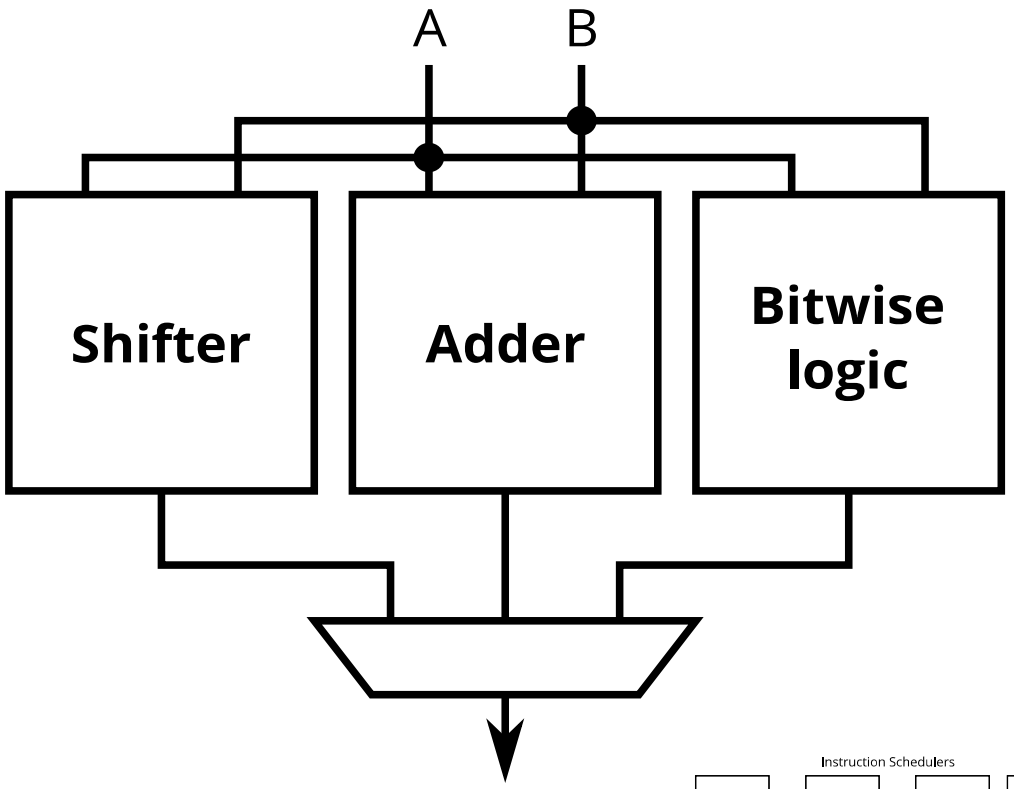
- Three parts:
  - Shifter
  - Adder
  - Bitwise logic
- We'll look at shifter and adder circuits

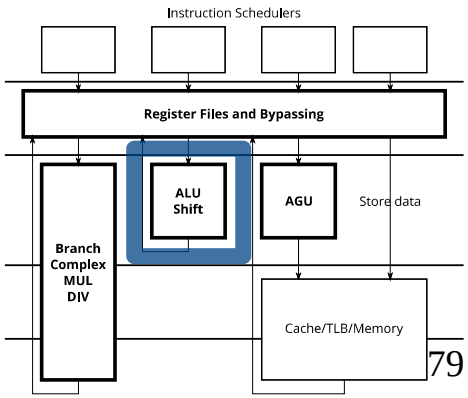


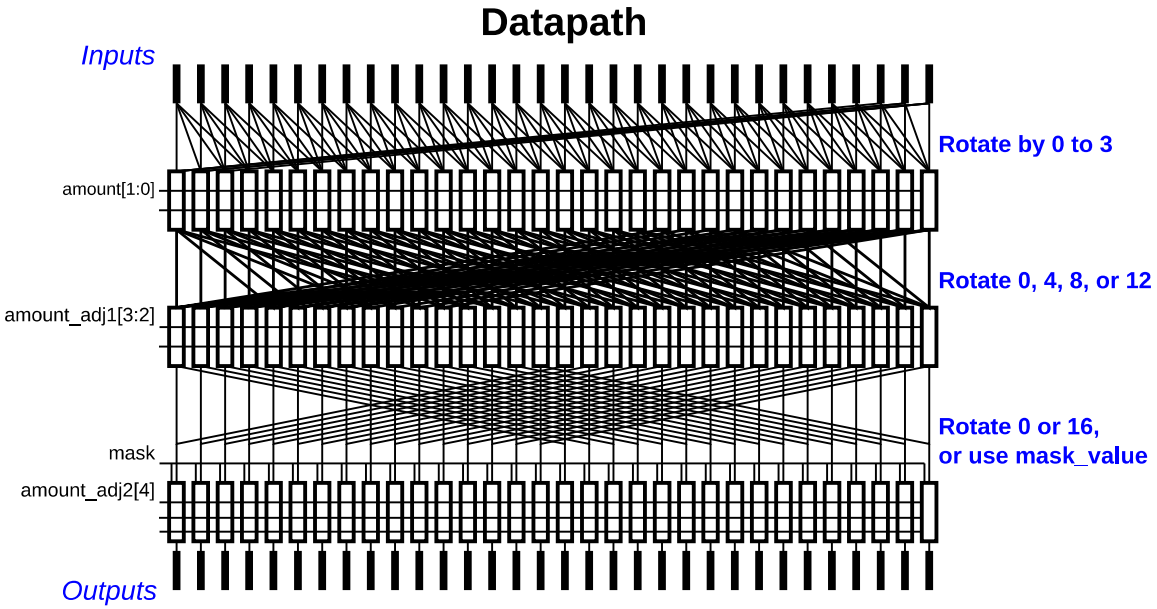
# Execution Circuits: Simple ALU

- Three parts:
  - Shifter
  - Adder
  - Bitwise logic
- We'll look at shifter and adder circuits

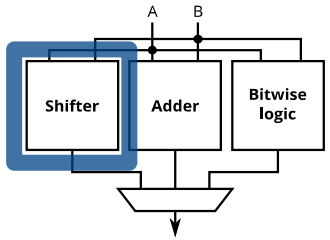


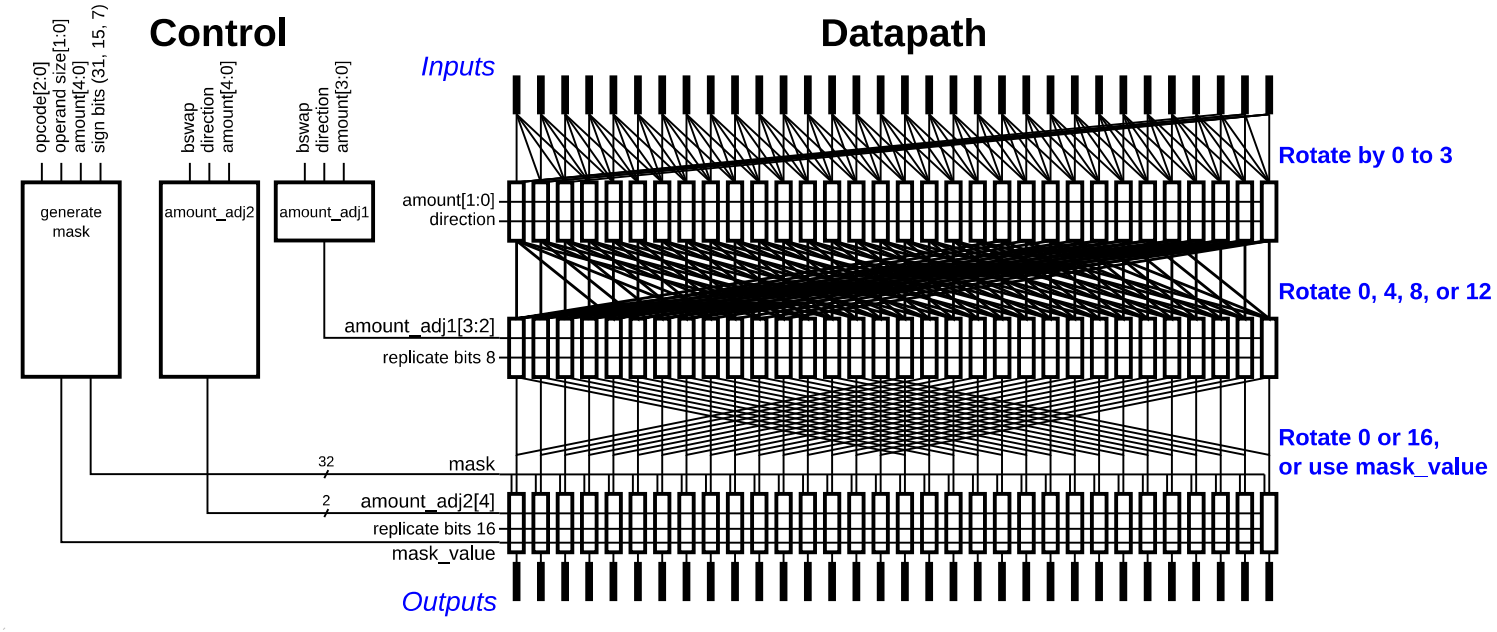
# Execution Circuit: Shifter



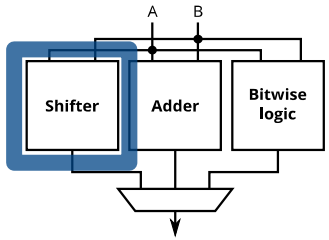

- Minimal 32-bit shifter: Needs 3 LUT levels (4-to-1 mux per level)
- We used a rotate + mask circuit: *Almost* 3 LUT levels



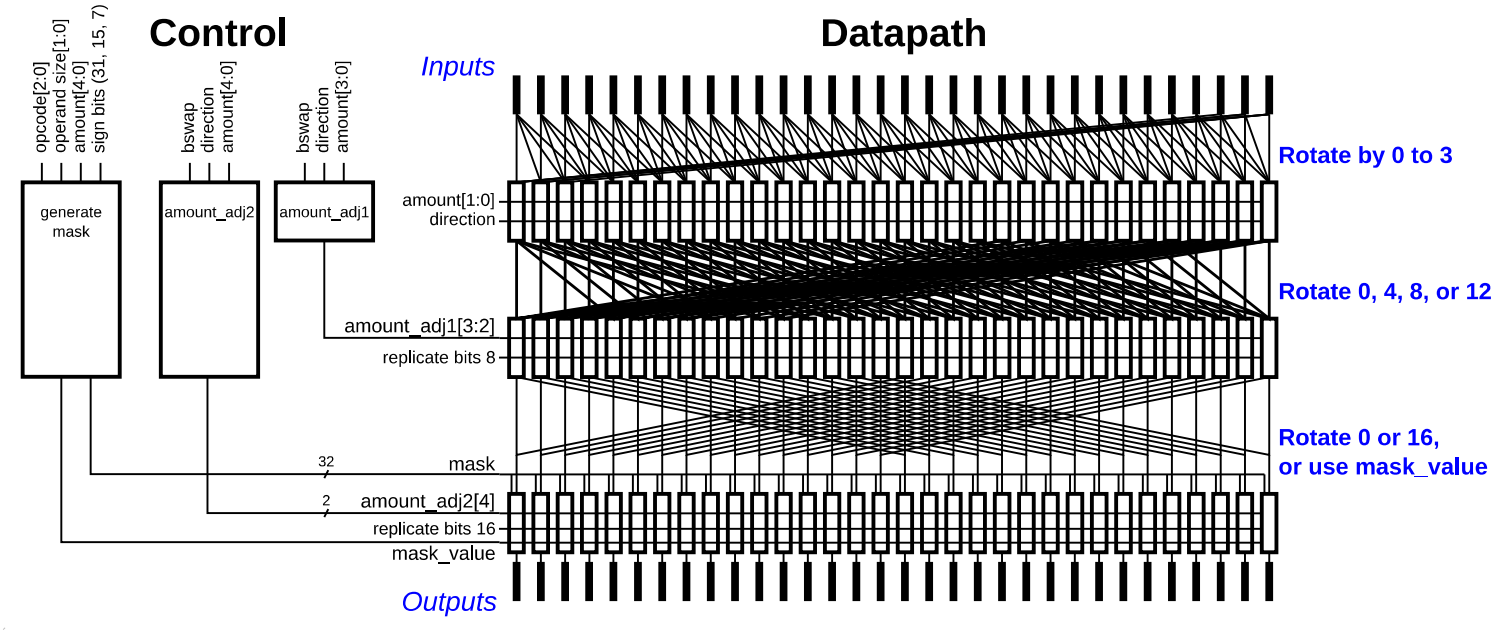
# Execution Circuit: Shifter



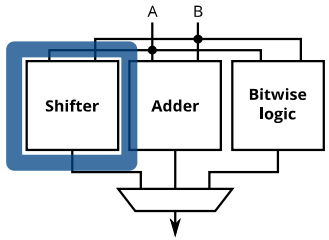

- Minimal 32-bit shifter: Needs 3 LUT levels (4-to-1 mux per level)
- We used a rotate + mask circuit: *Almost* 3 LUT levels
  - Left and right. 32-, 16-, and 8-bit operands
  - Rotate, shift, arithmetic shift
  - Rotate-through-carry-by-1
  - Byte swap (aa bb cc dd → dd cc bb aa)
  - Sign extension



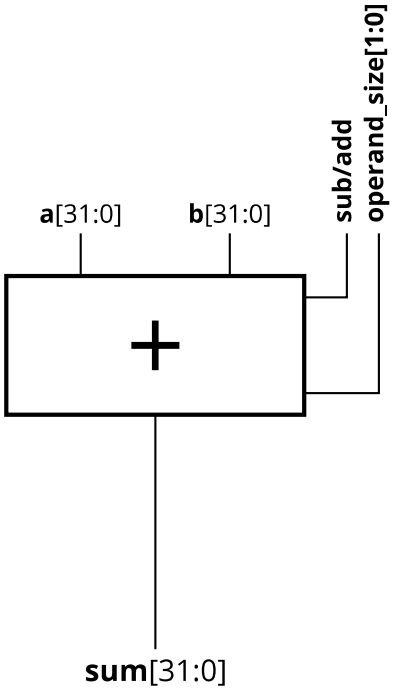
# Execution Circuit: Shifter



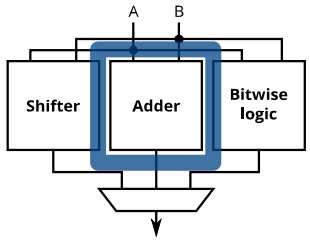

- Minimal 32-bit shifter: Needs 3 LUT levels (4-to-1 mux per level)
- We used a rotate + mask circuit: *Almost* 3 LUT levels
  - Left and right. 32-, 16-, and 8-bit operands
  - Rotate, shift, arithmetic shift
  - Rotate-through-carry-by-1
  - Byte swap (aa bb cc dd → dd cc bb aa)
  - Sign extension
- **2.9 ns, 54% faster (and 46% smaller) than HDL synthesis**



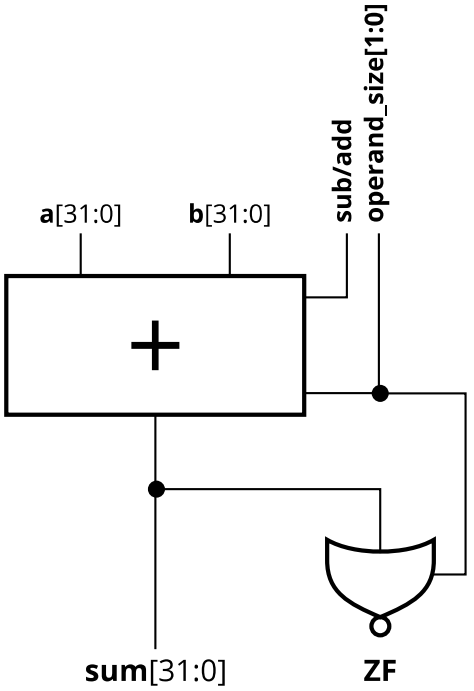
# Execution Circuit: Adder



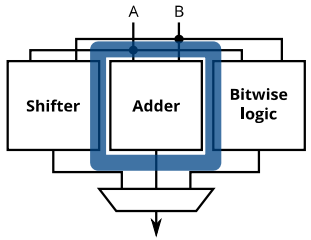

- FPGAs have carry chains: Can't improve on it by much



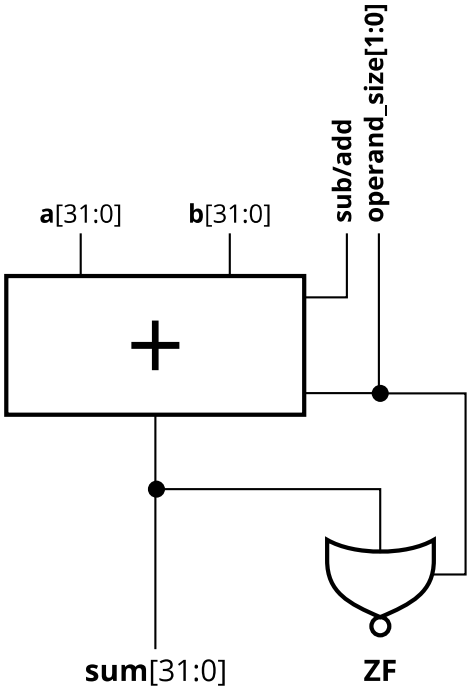
# Execution Circuit: Adder



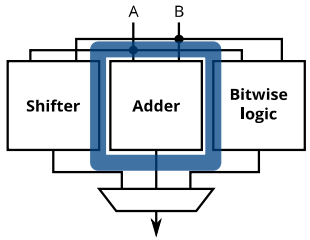

- FPGAs have carry chains: Can't improve on it by much
- Condition codes: ZF means "is the result zero?"
  - 32/16/8-bit NOR gate is 3 LUT levels *plus* the adder…



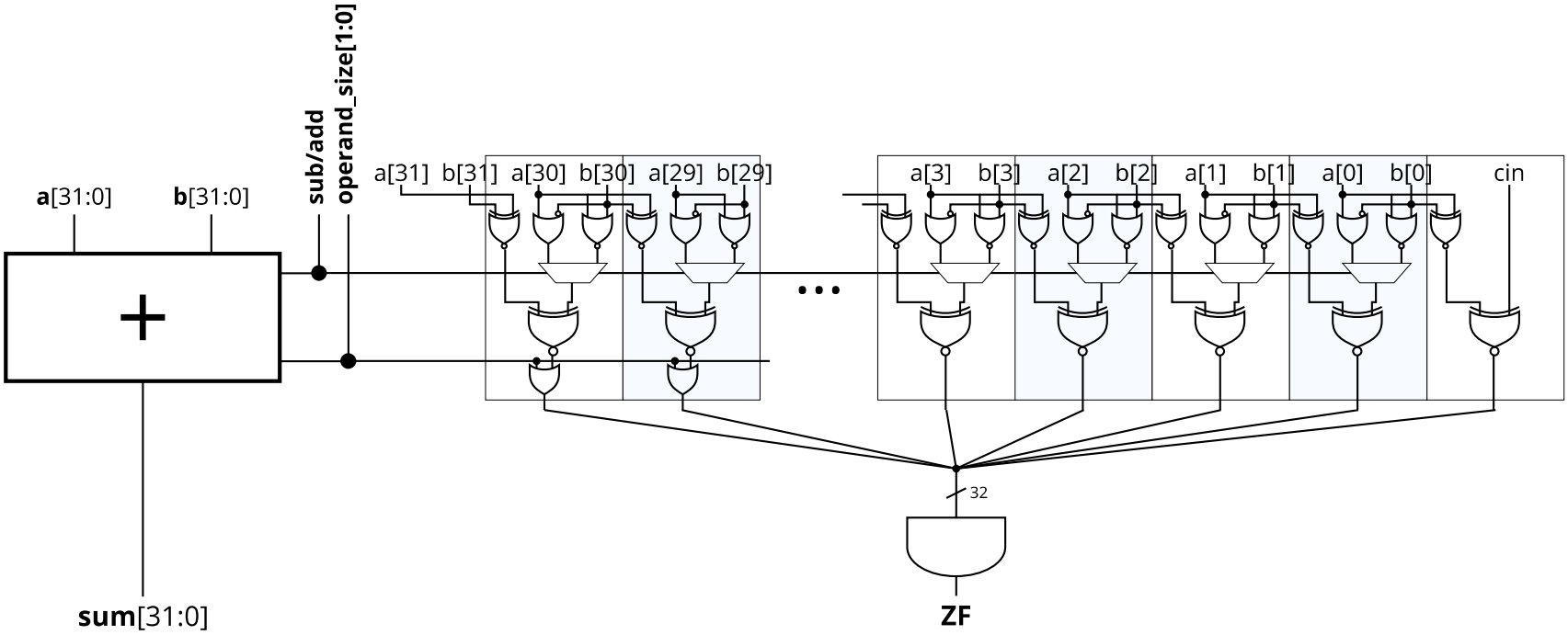
# Execution Circuit: Adder



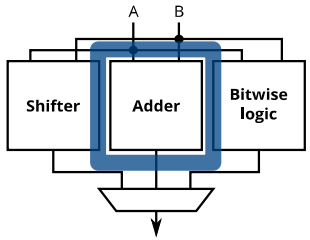

- FPGAs have carry chains: Can't improve on it by much
- Condition codes: ZF means "is the result zero?"
  - 32/16/8-bit NOR gate is 3 LUT levels *plus* the adder…
- Computing *a* + *b* = ***K*** does not need addition!
  - ZF: 3 LUT levels in *parallel* with adder



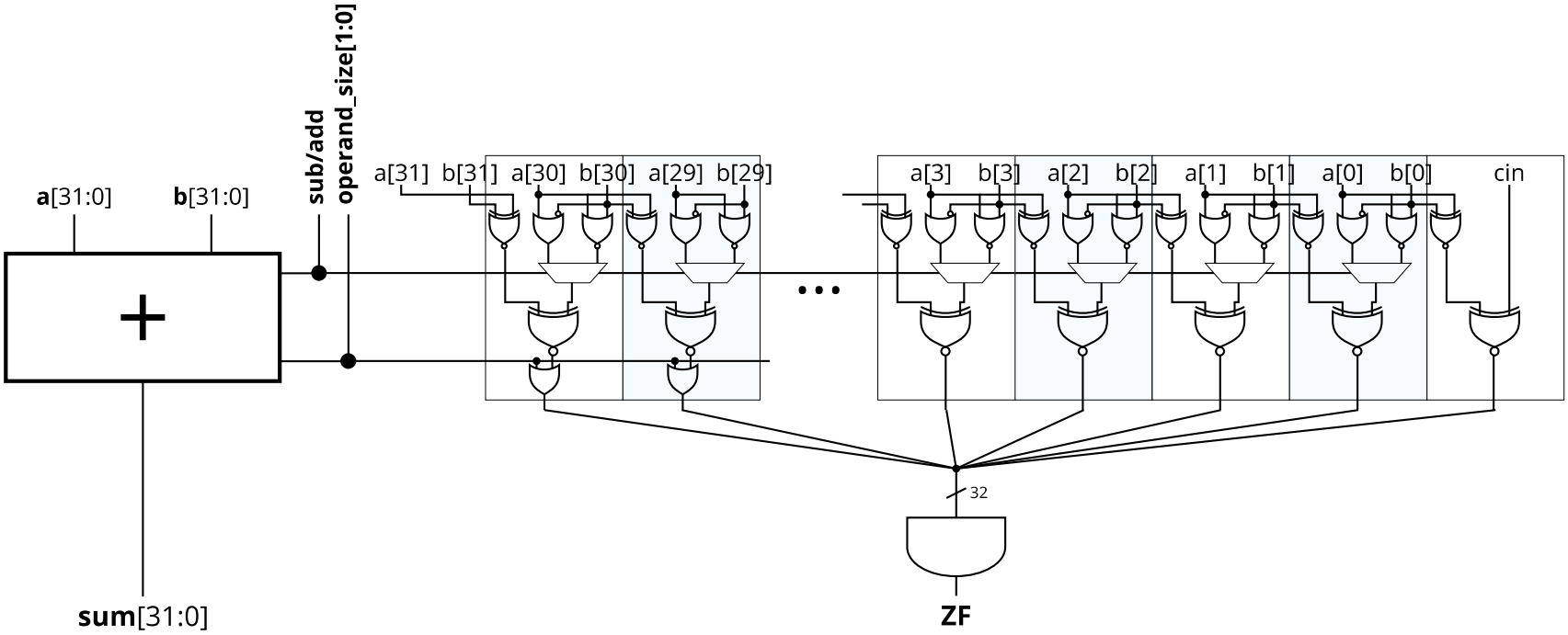
# Execution Circuit: Adder



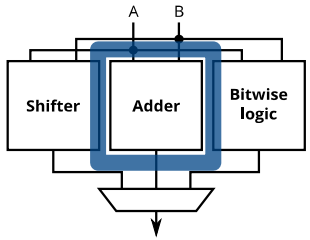

- FPGAs have carry chains: Can't improve on it by much
- Condition codes: ZF means "is the result zero?"
  - 32/16/8-bit NOR gate is 3 LUT levels *plus* the adder…
- Computing *a* + *b* = ***K*** does not need addition!
  - ZF: 3 LUT levels in *parallel* with adder



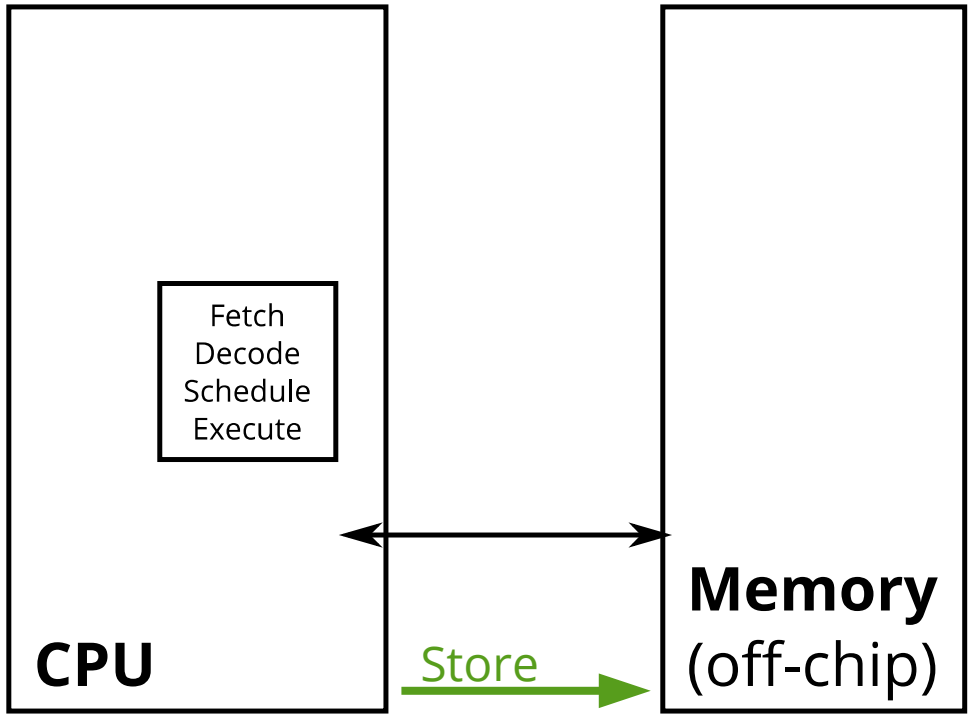
# Execution Circuit: Adder




- FPGAs have carry chains: Can't improve on it by much
- Condition codes: ZF means "is the result zero?"
  - 32/16/8-bit NOR gate is 3 LUT levels *plus* the adder…
- Computing *a* + *b* = ***K*** does not need addition!
  - ZF: 3 LUT levels in *parallel* with adder
- **2.3ns, 24% faster, +55% area (+30 ALM) vs. HDL synthesis**

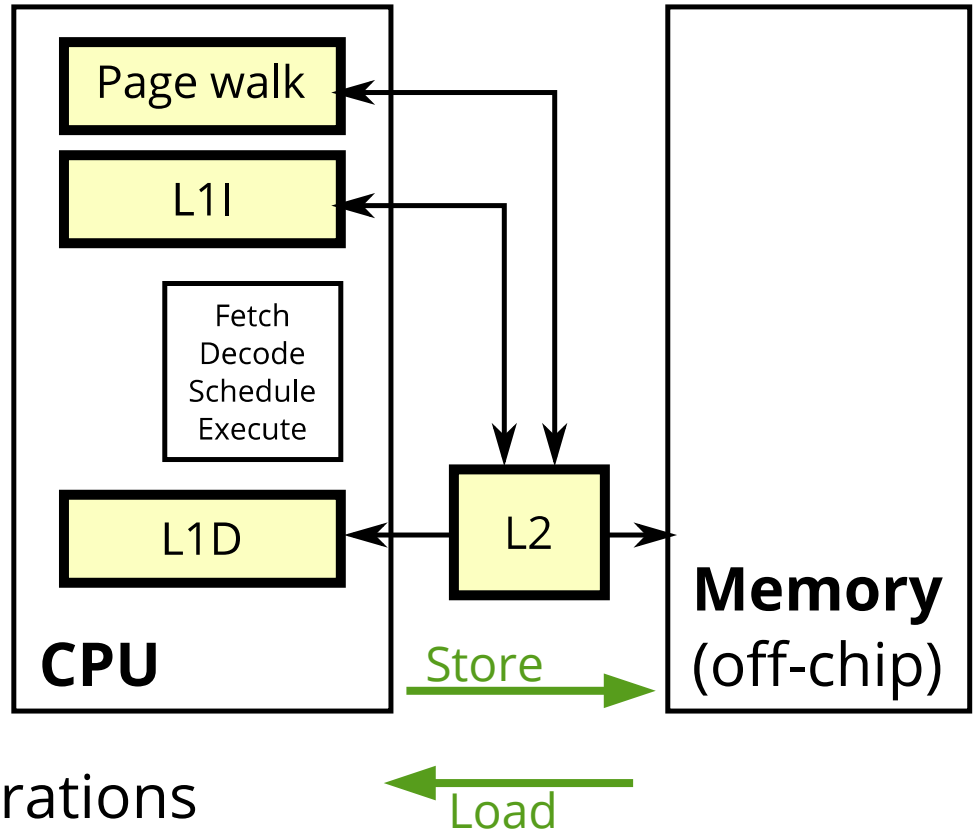


# Memory System Microarchitecture




- Memory operations
  - **Store:** `mov [ecx], eax`
  - **Load:** `mov eax, [ecx]`

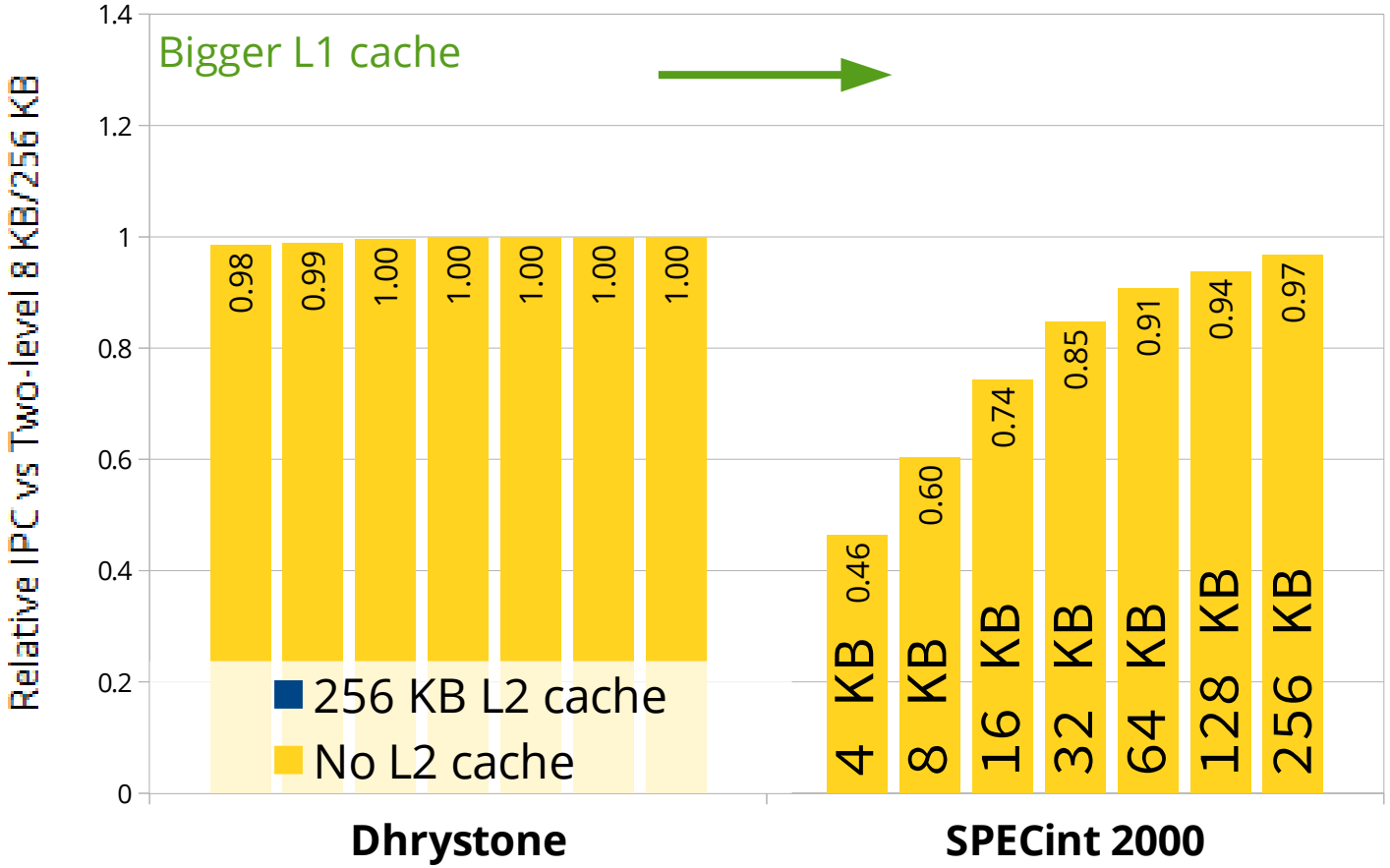
# Memory System Microarchitecture




- Memory operations
  - **Store:** `mov [ecx], eax`
  - **Load:** `mov eax, [ecx]`
- Caches
  - For performance (Instruction and data)
  - For OS support (TLB and page walking)

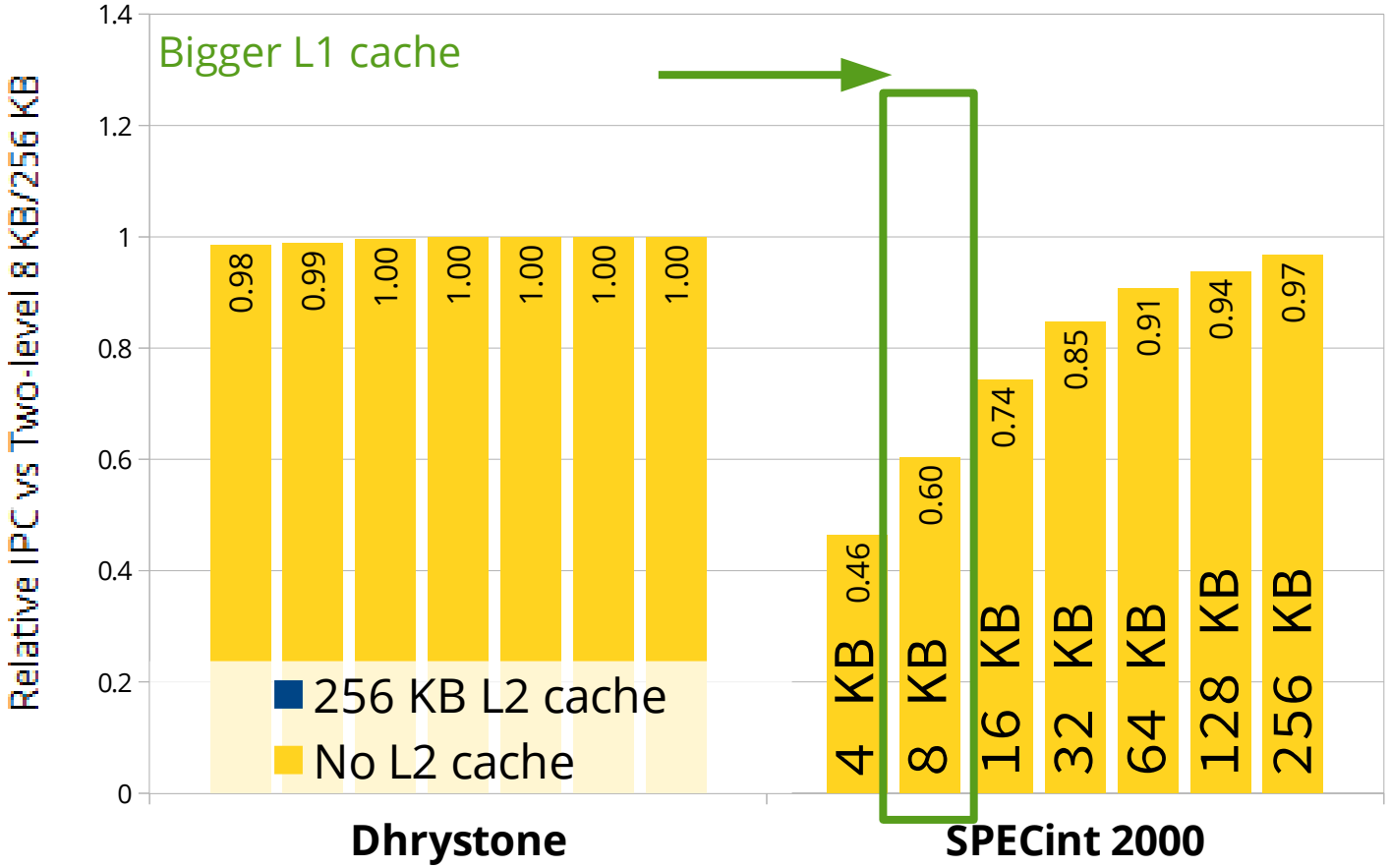
# Basic Cache Trade-offs

# Basic Cache Trade-offs



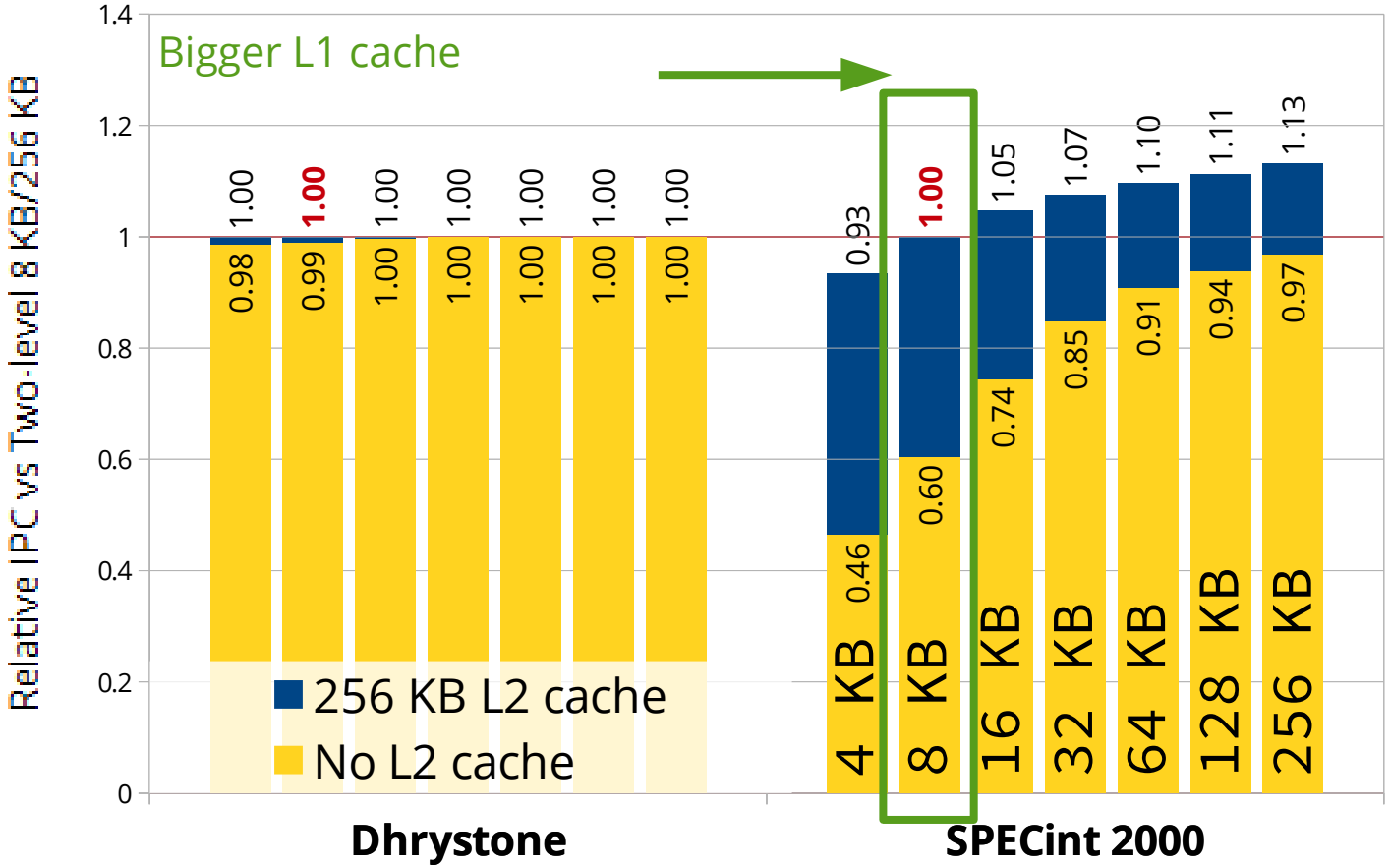

- Bigger cache → higher IPC
  - Sensitivity varies with workload

# Basic Cache Trade-offs



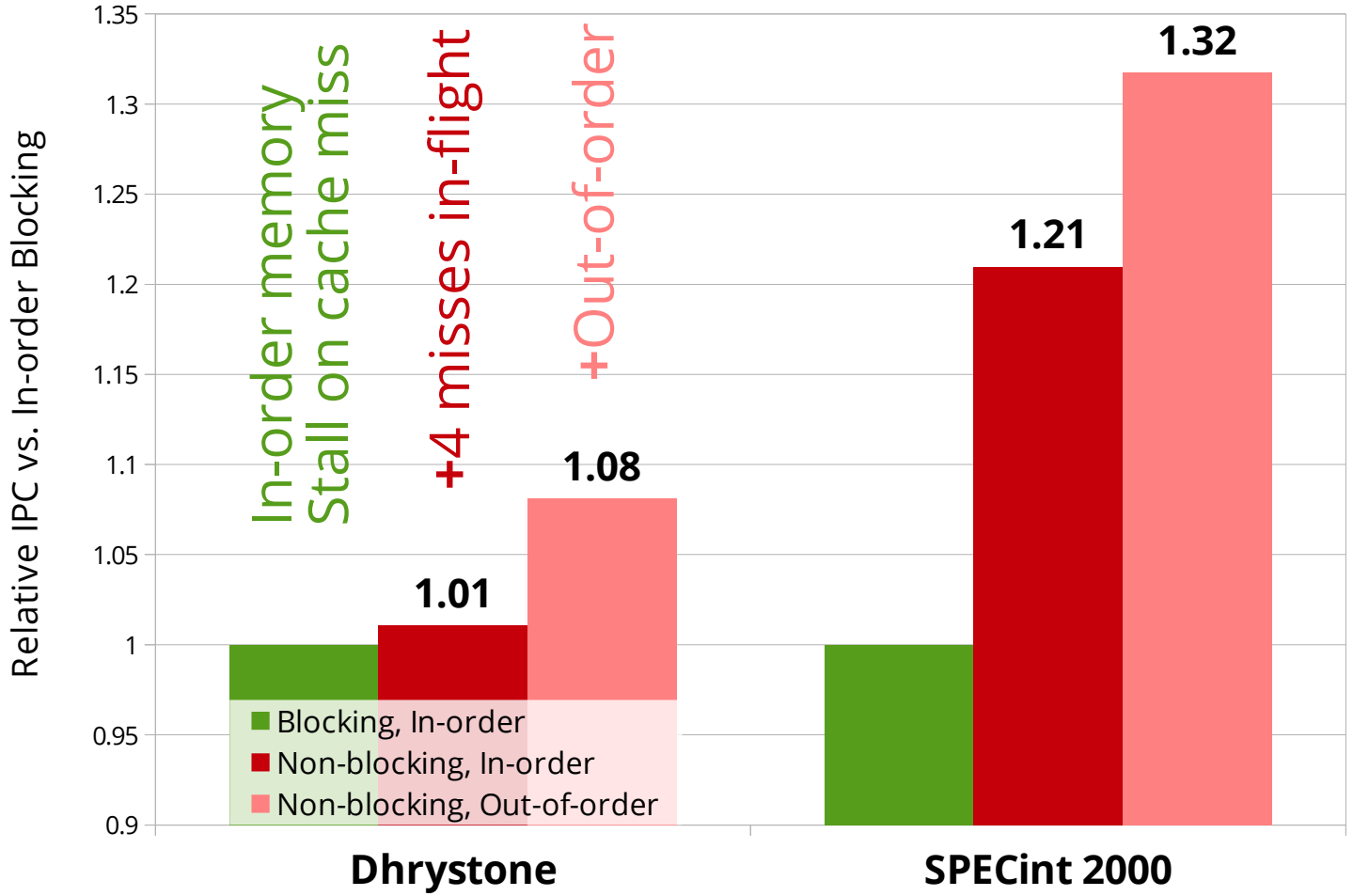

- Bigger cache → higher IPC
  - Sensitivity varies with workload
- L1 caches need to be small (We chose 8 KB)

# Basic Cache Trade-offs




- Bigger cache → higher IPC
  - Sensitivity varies with workload
- L1 caches need to be small (We chose 8 KB)
- L2 cache (256 KB) mostly makes up for this

# More Memory System Trade-offs



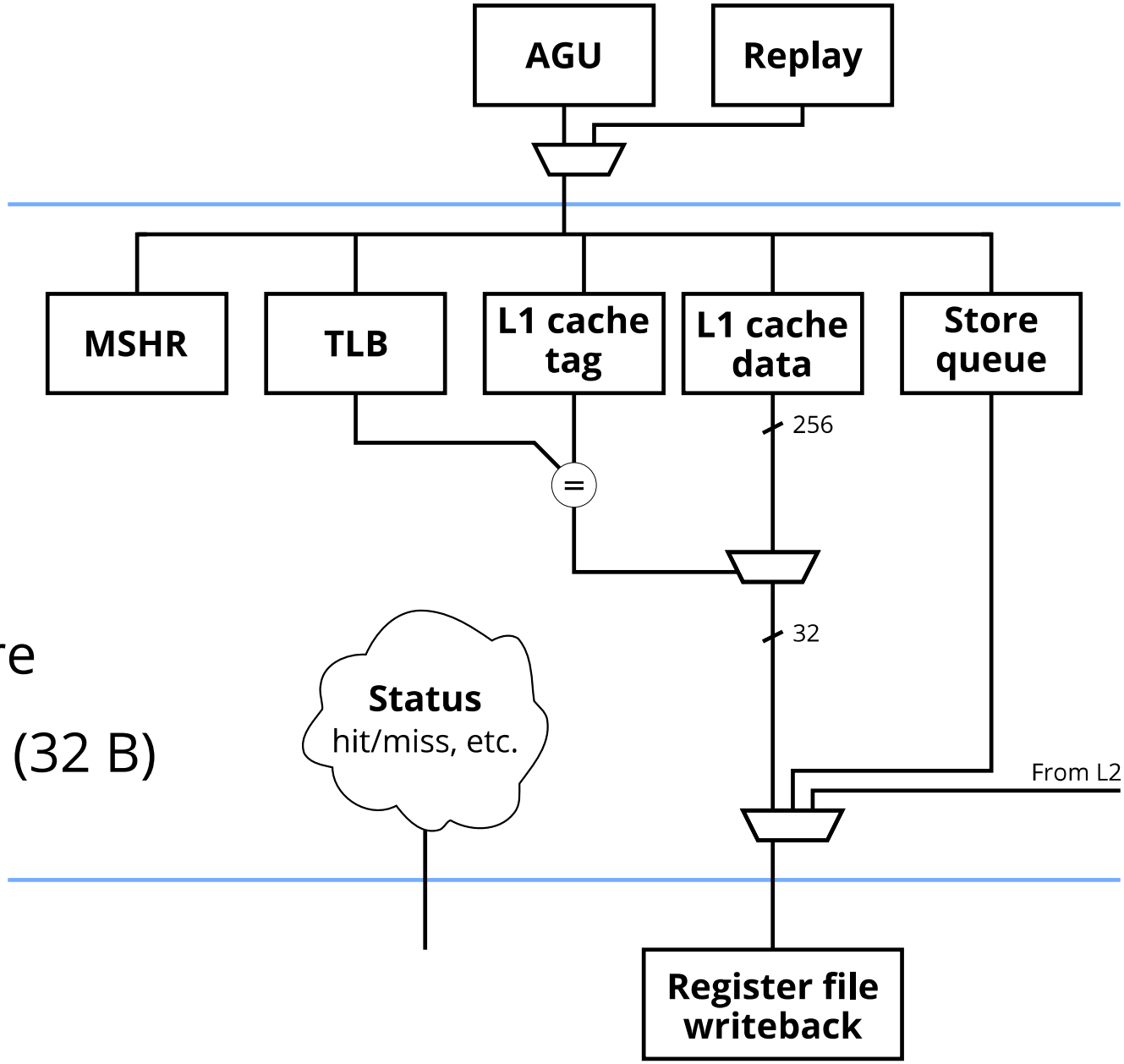

- Multiple in-flight misses
- Out-of-order
  - Memory dependence speculation

# L1 Memory System

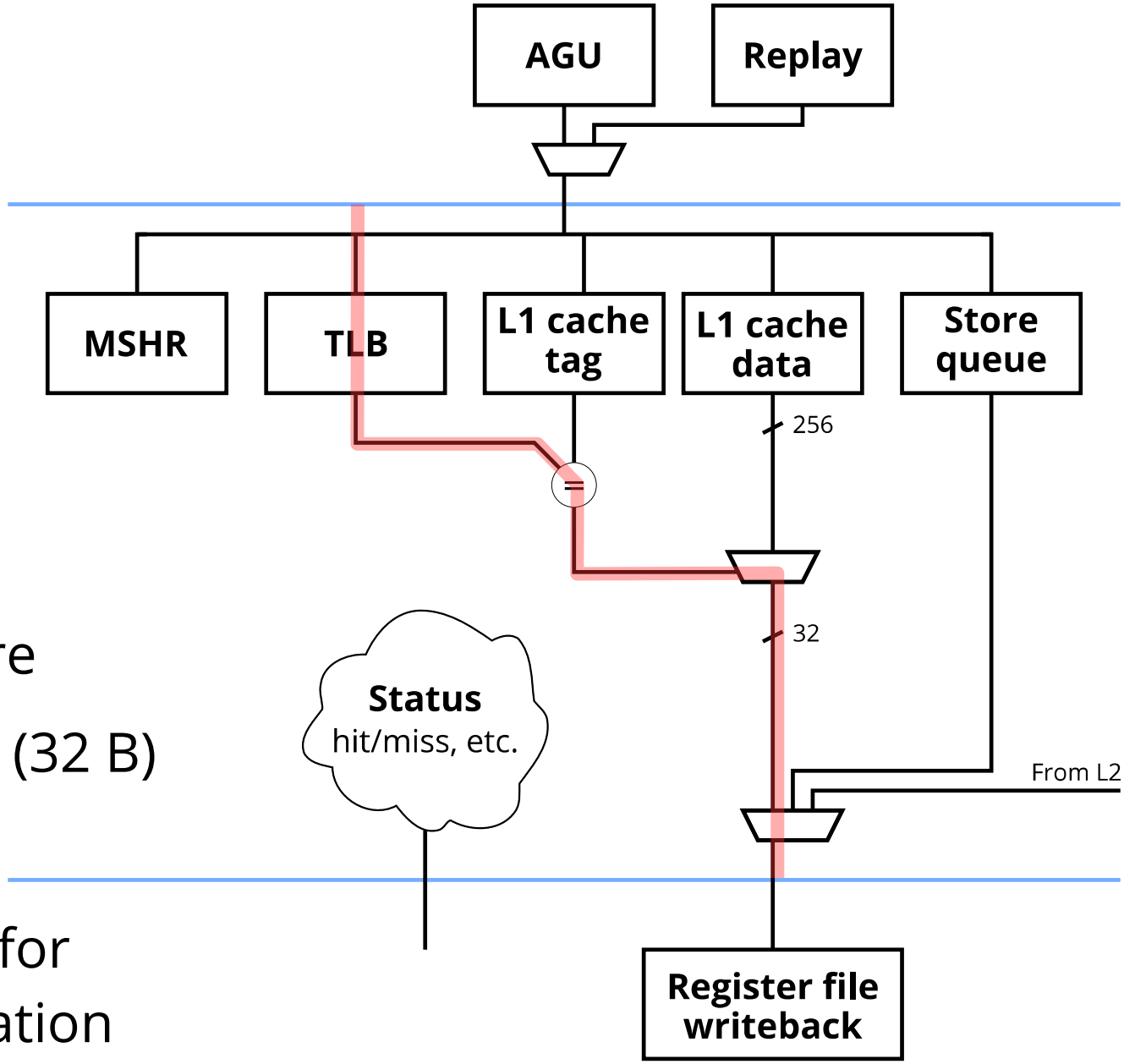
- TLB lookup
- Cache tag compare
- Cache data rotate (32 B)

# L1 Memory System

- TLB lookup
- Cache tag compare
- Cache data rotate (32 B)



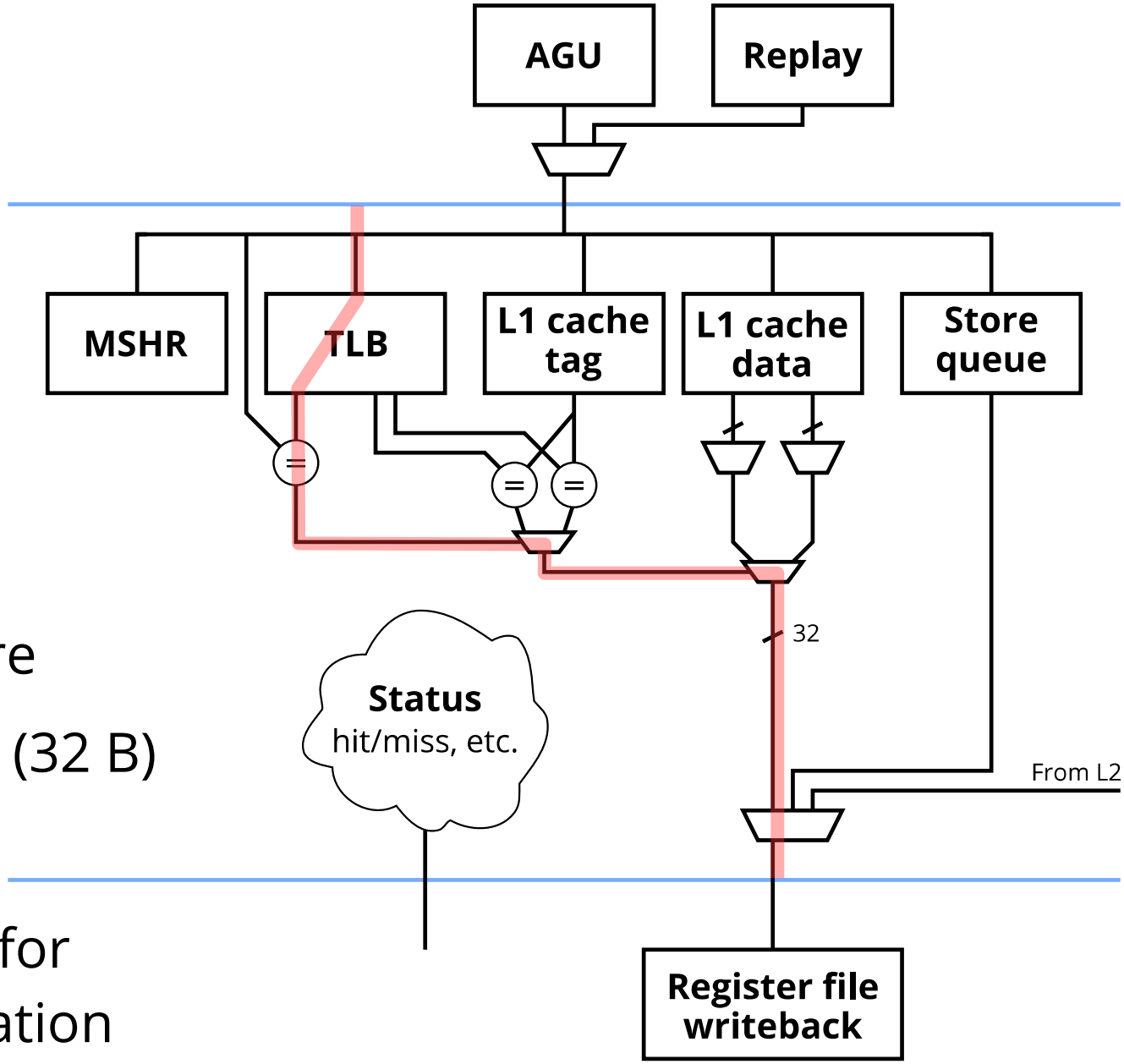
# L1 Memory System




- TLB lookup
- Cache tag compare
- Cache data rotate (32 B)

- Long critical path for direct implementation

# L1 Memory System

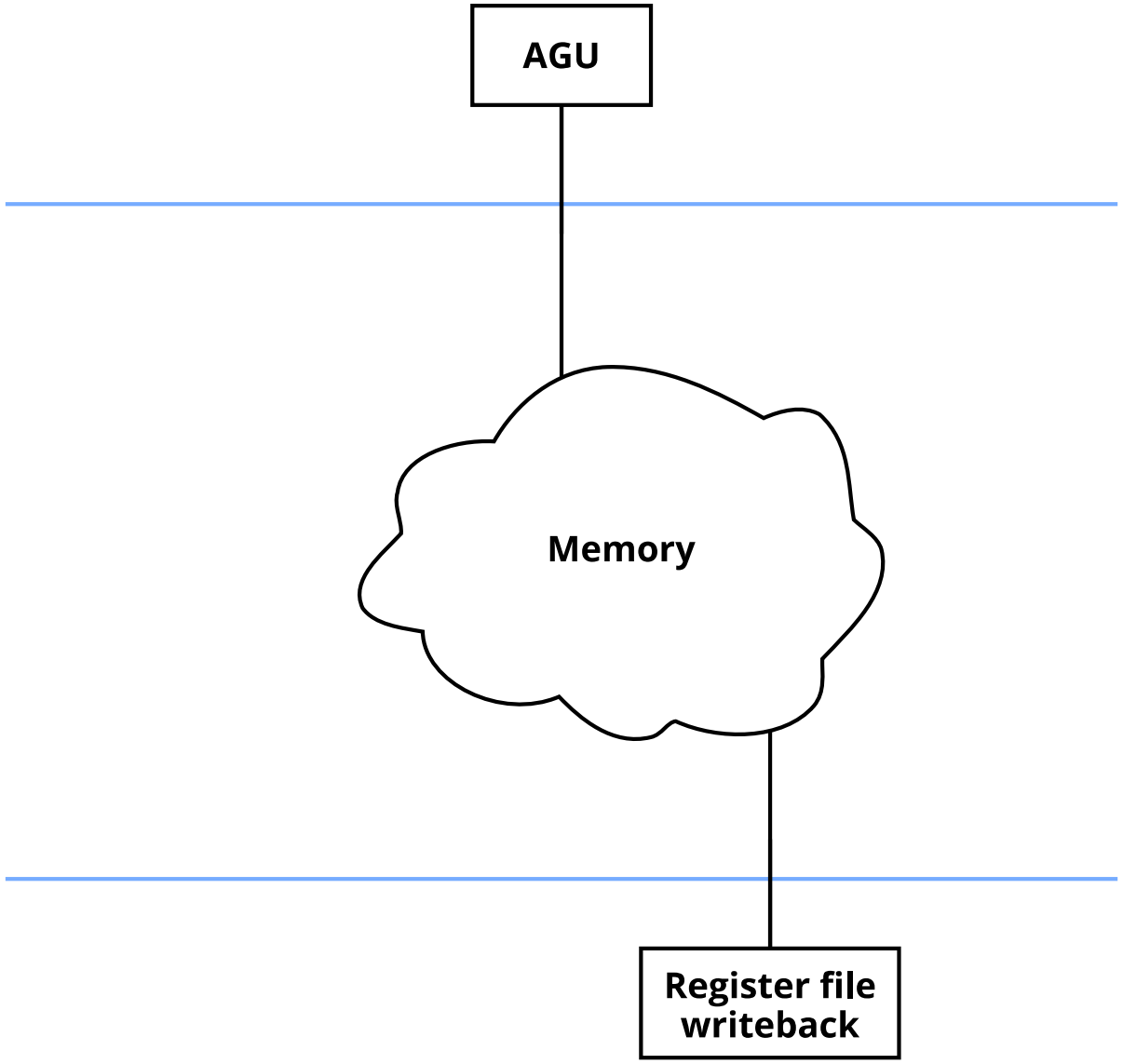



- TLB lookup
- Cache tag compare
- Cache data rotate (32 B)

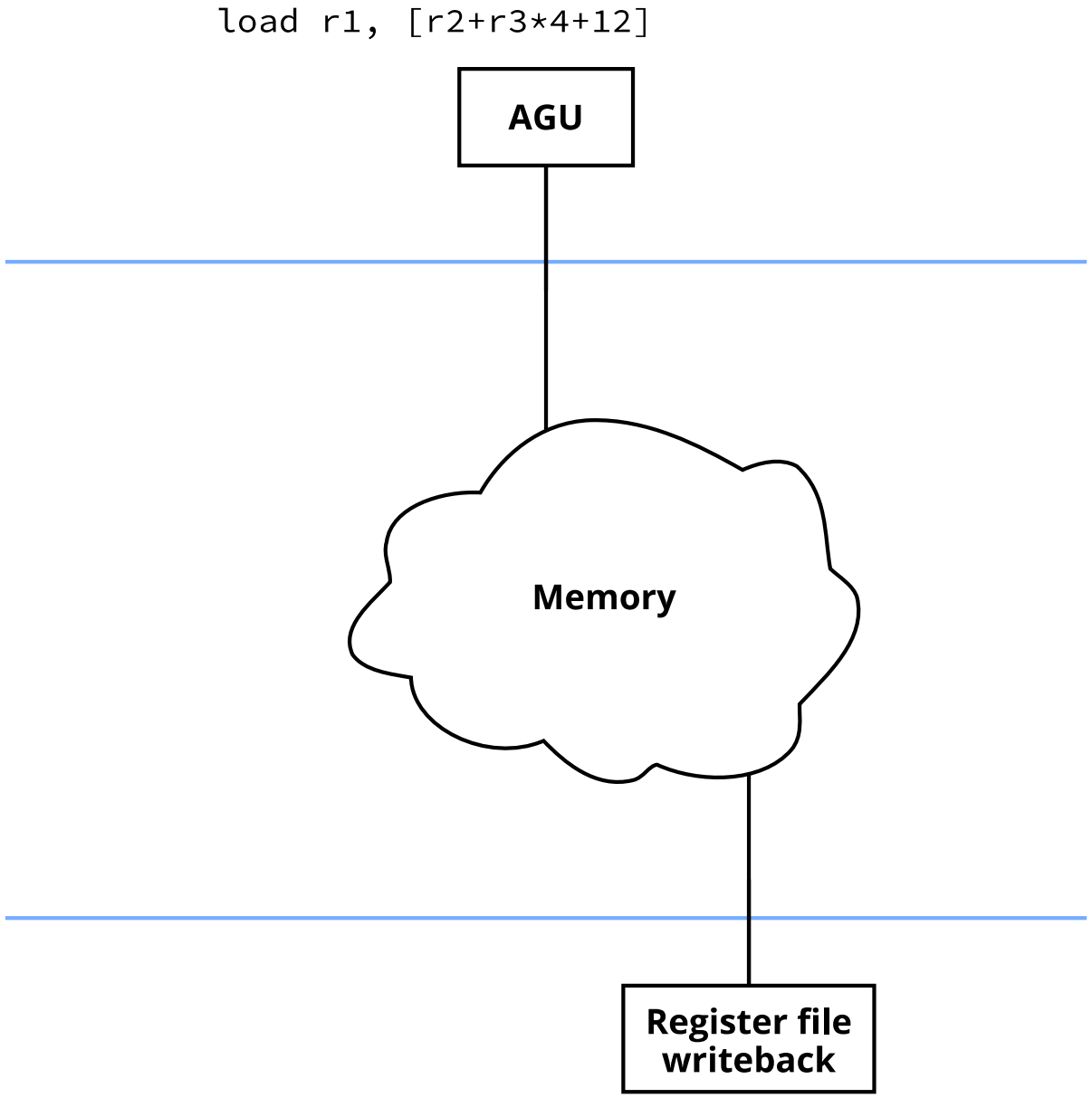
- Long critical path for direct implementation

# What happens to a load: simplified

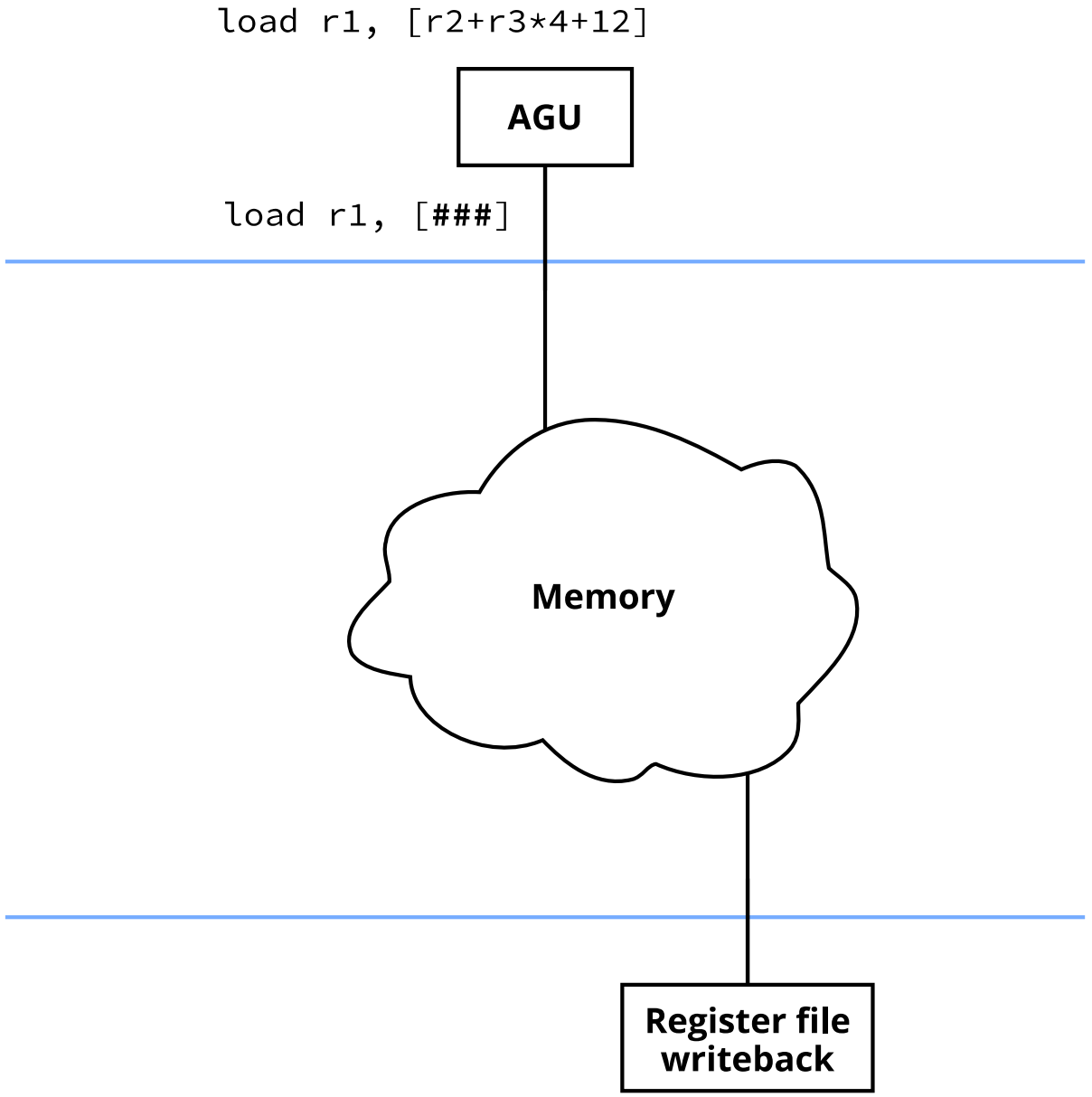
# What happens to a load: simplified



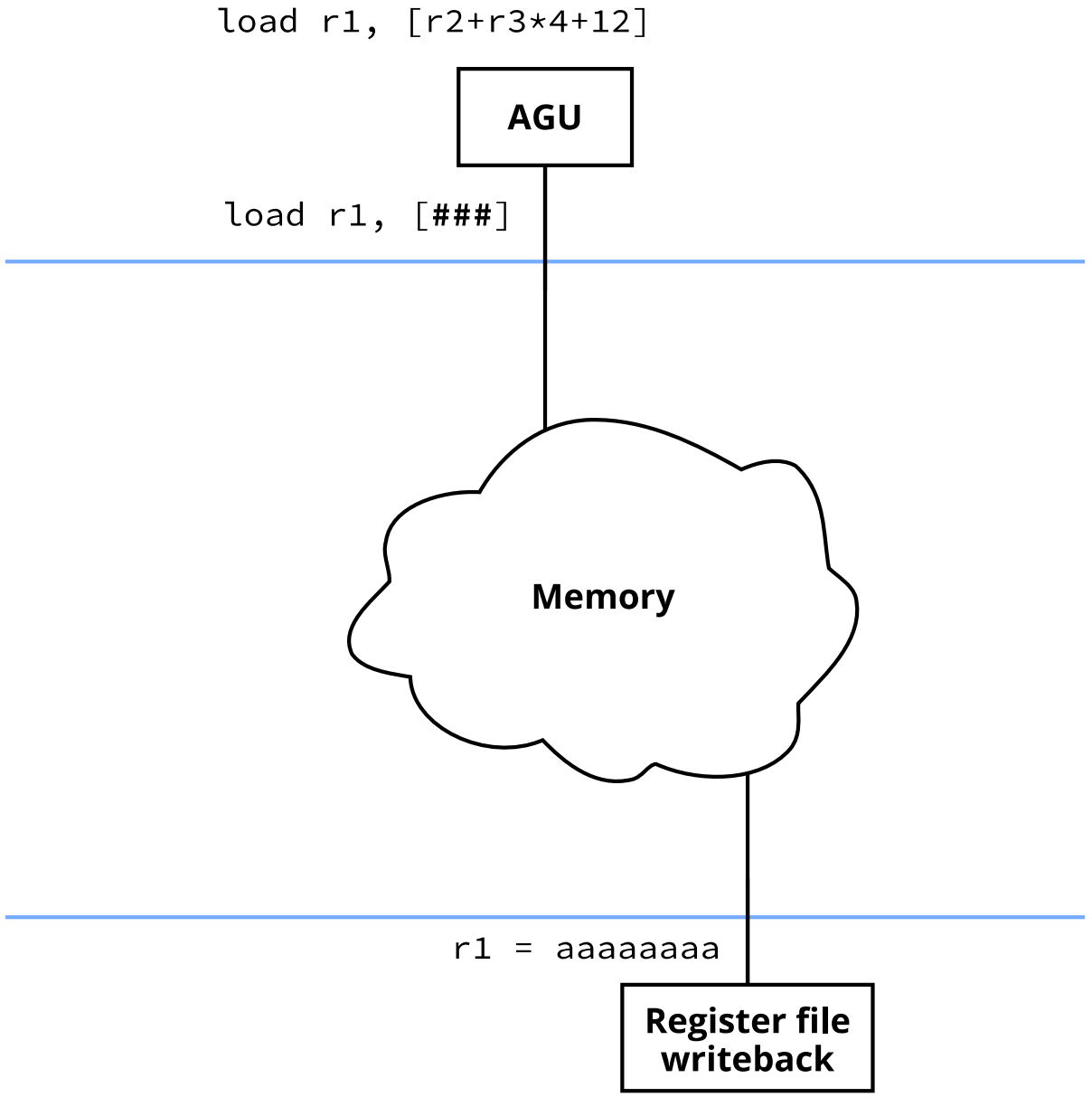
# What happens to a load: simplified



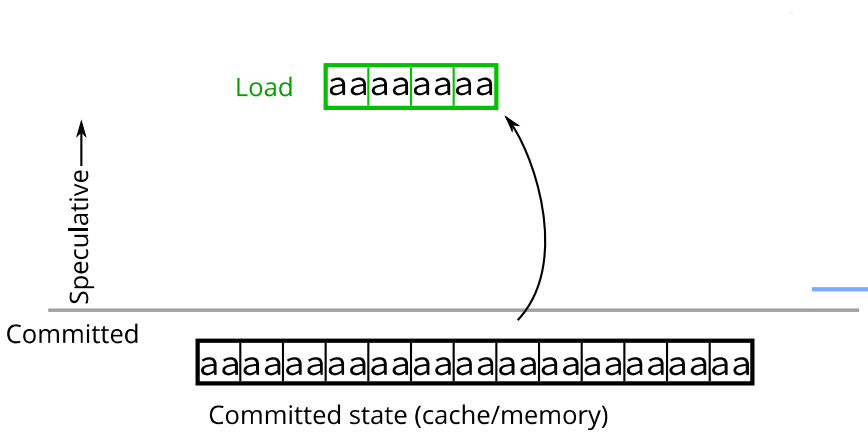
# What happens to a load: simplified

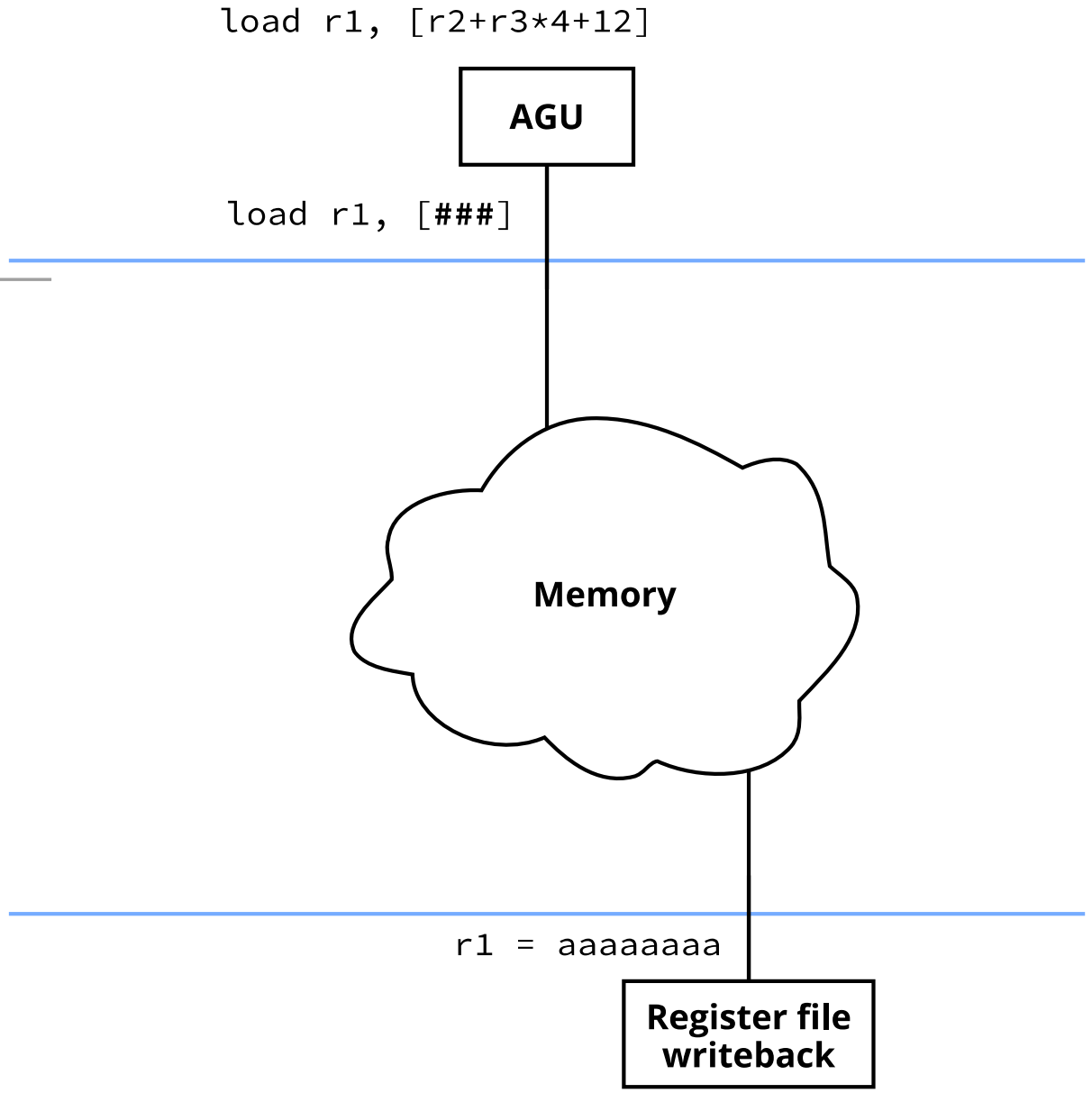


# What happens to a load: simplified

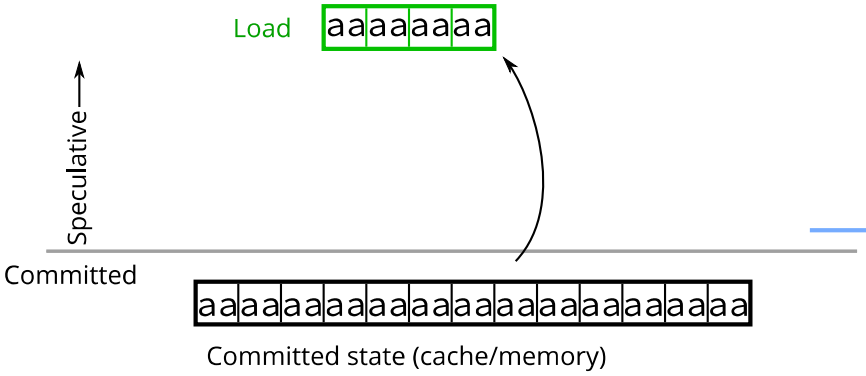


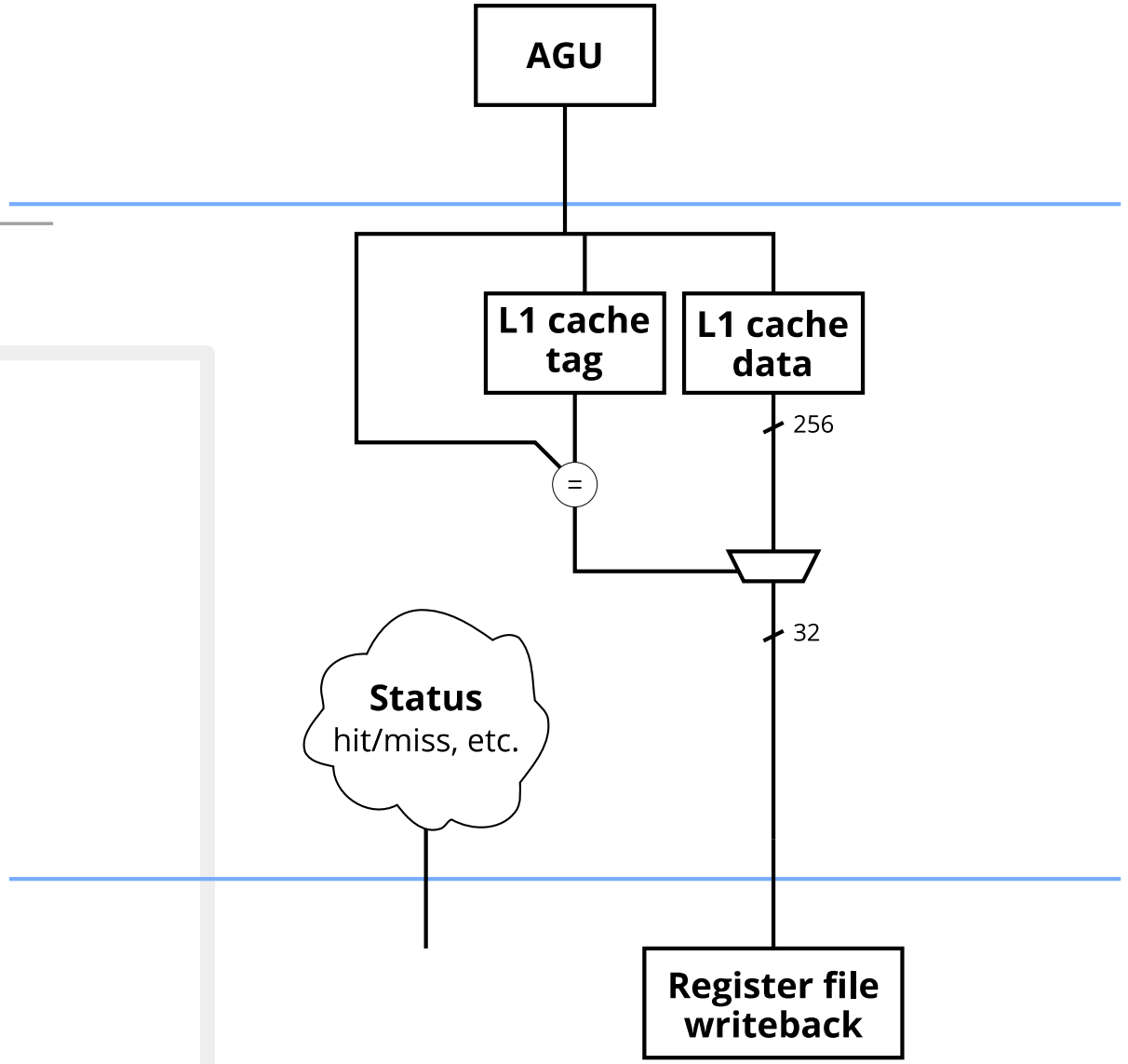
# What happens to a load: simplified



# What happens to a load: simplified



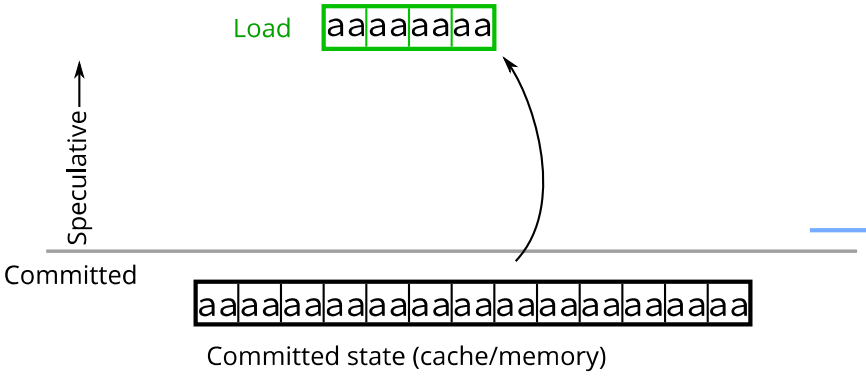



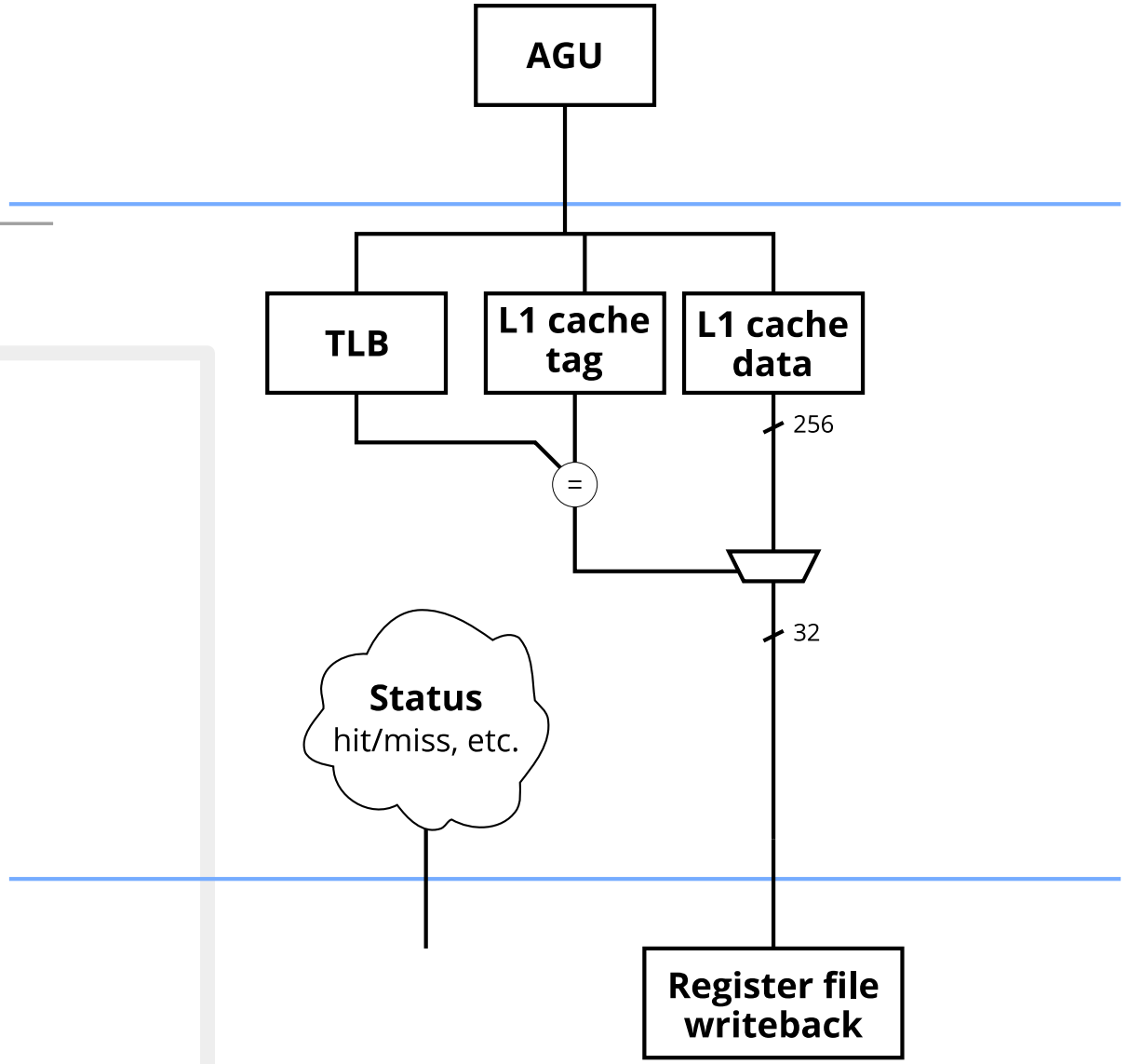

**Possible outcomes for a load:**

**Cache miss**

**Cache hit**

# What happens to a load: simplified



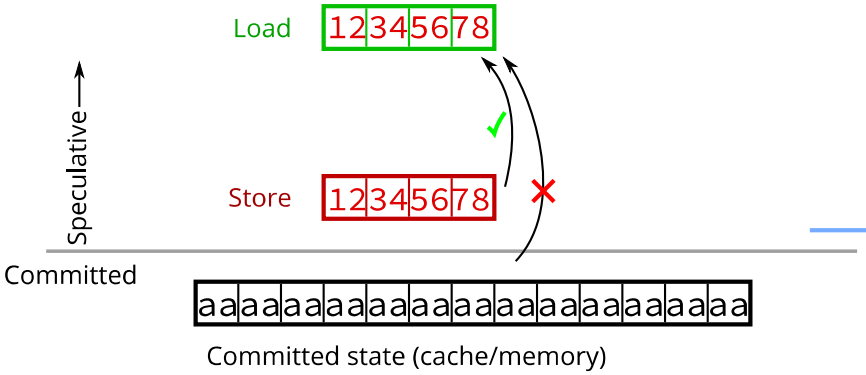

**Possible outcomes for a load:**

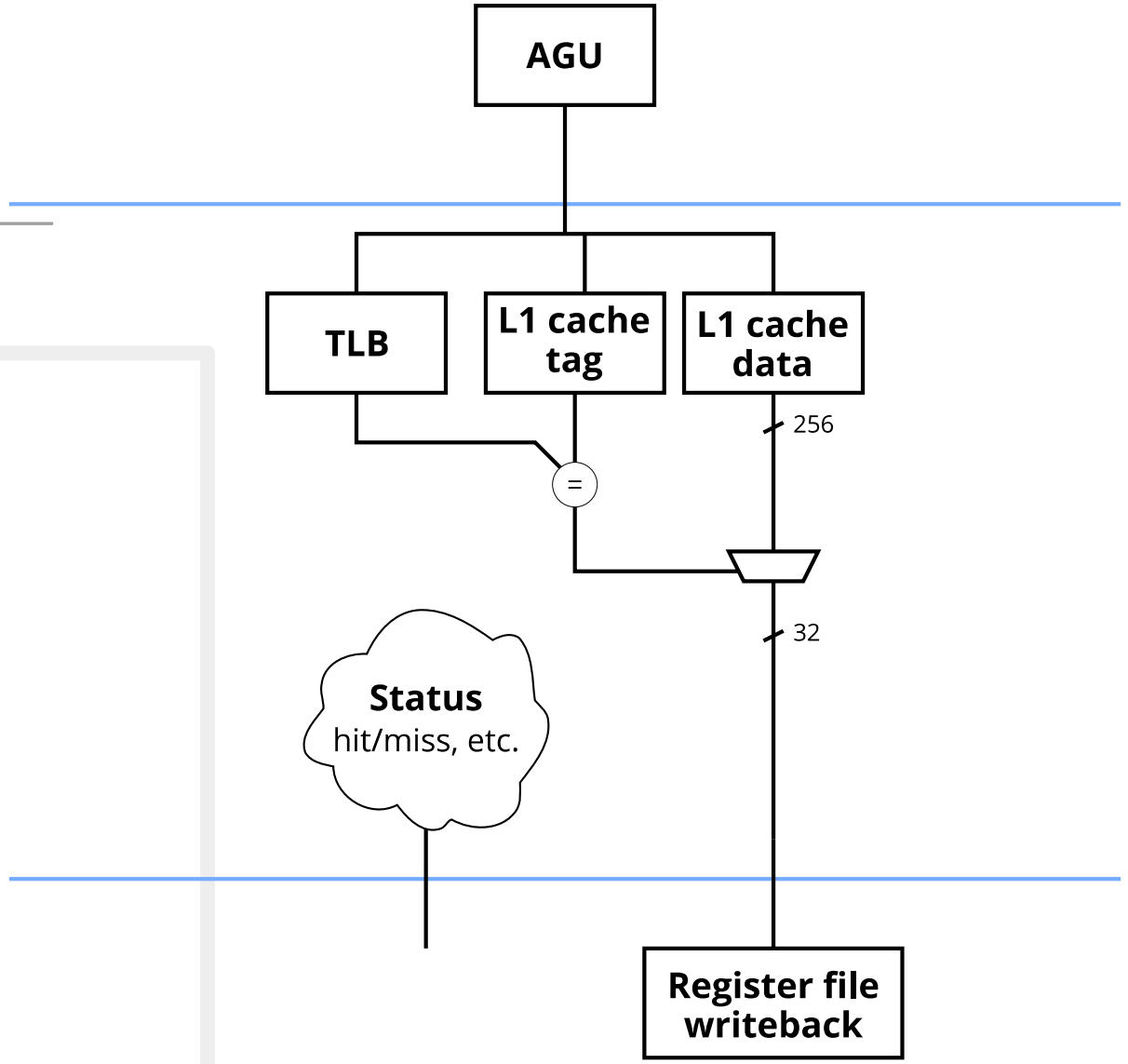
**TLB miss**

Cache miss

Cache hit

# What happens to a load: simplified



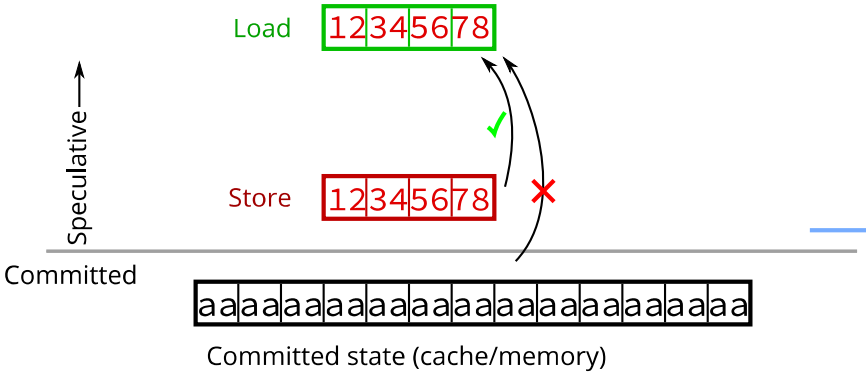

**Possible outcomes for a load:**

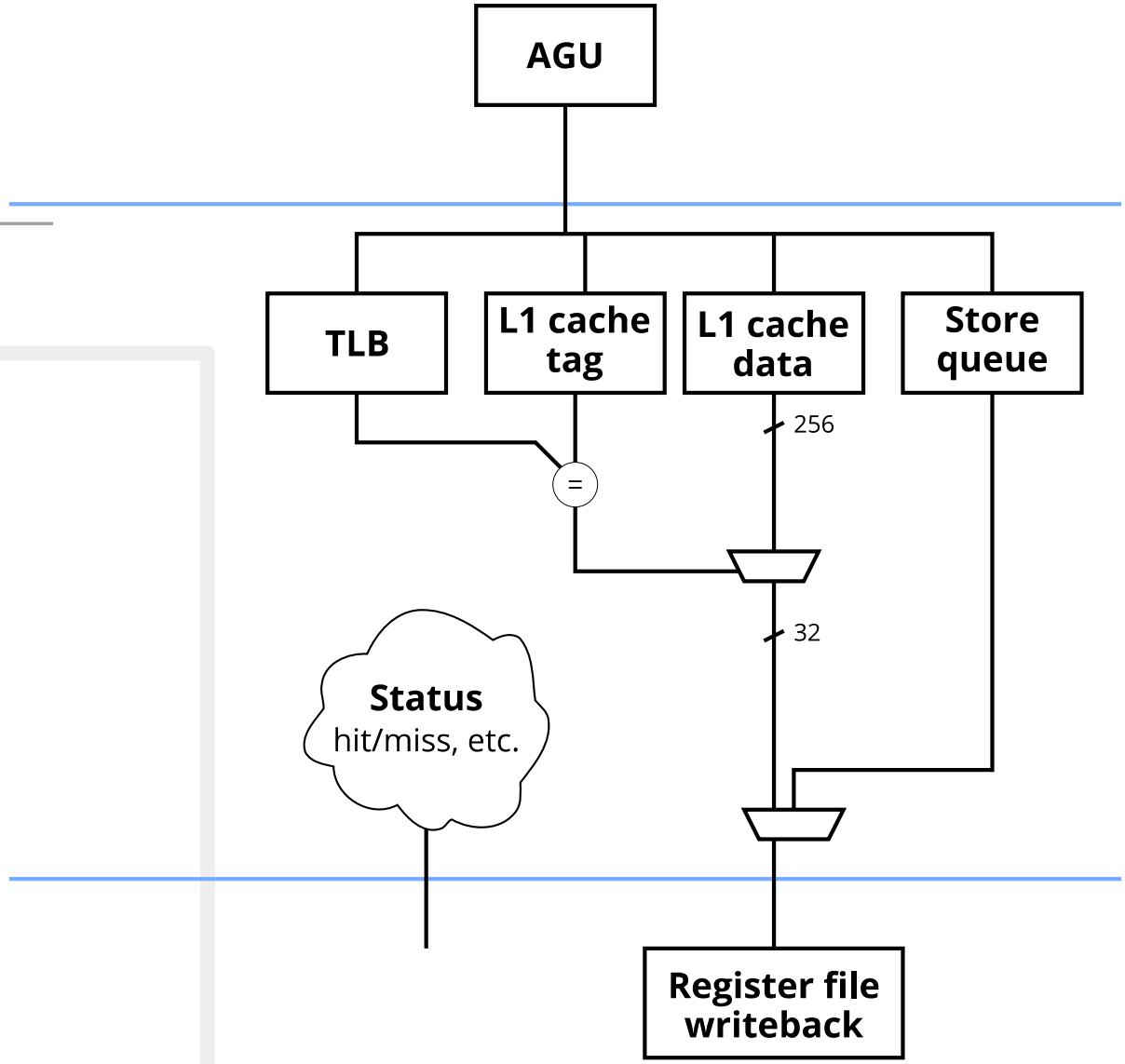
**TLB miss**

Cache miss

Cache hit

# What happens to a load: simplified




**Possible outcomes for a load:**

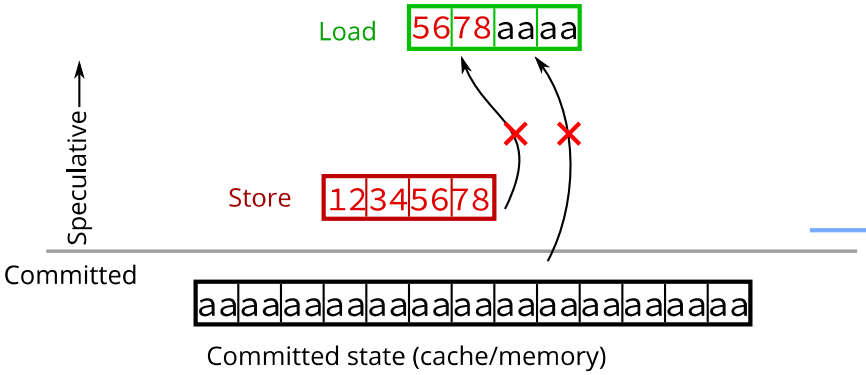
TLB miss

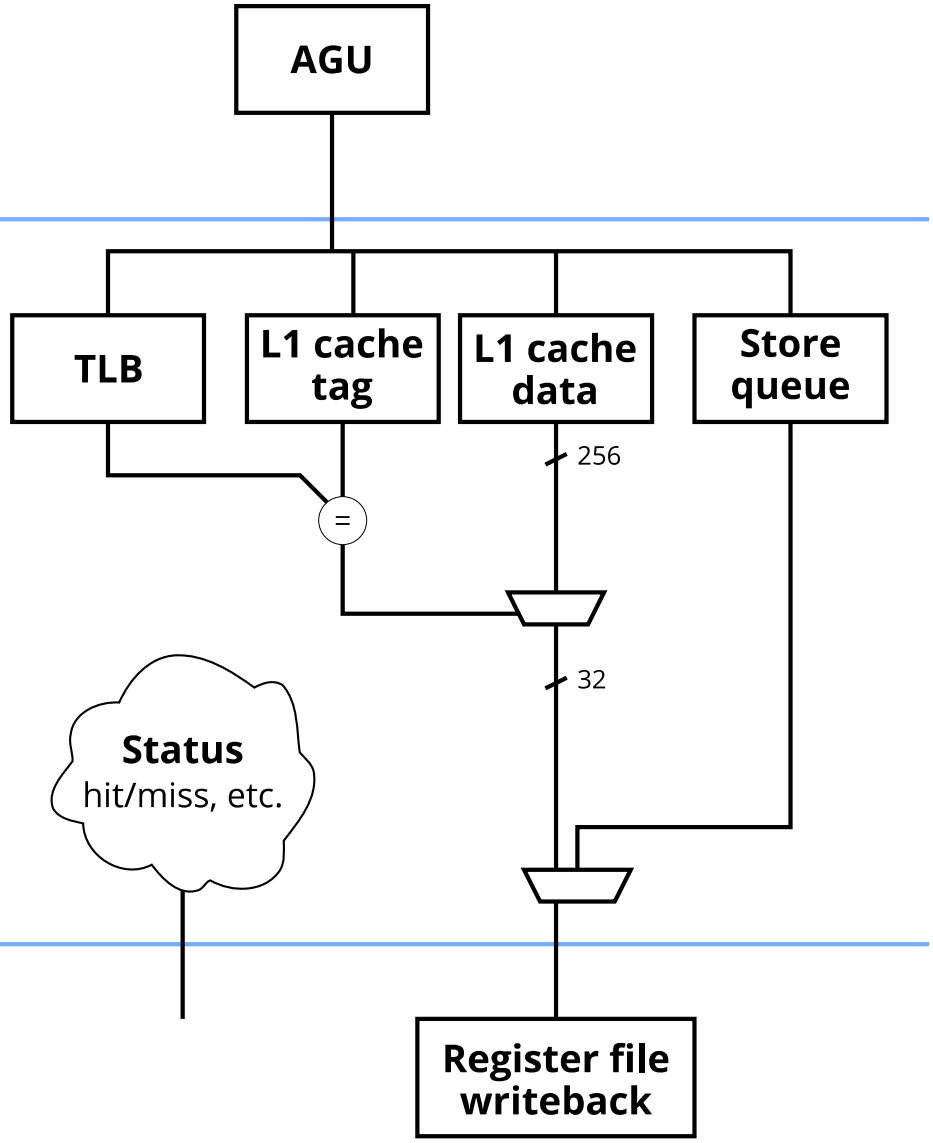
**Store-forward, success**

Cache miss

Cache hit

# What happens to a load: simplified




**Possible outcomes for a load:**

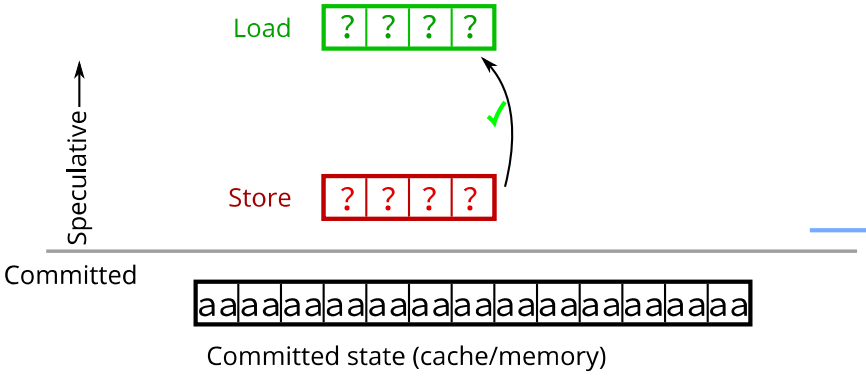
TLB miss

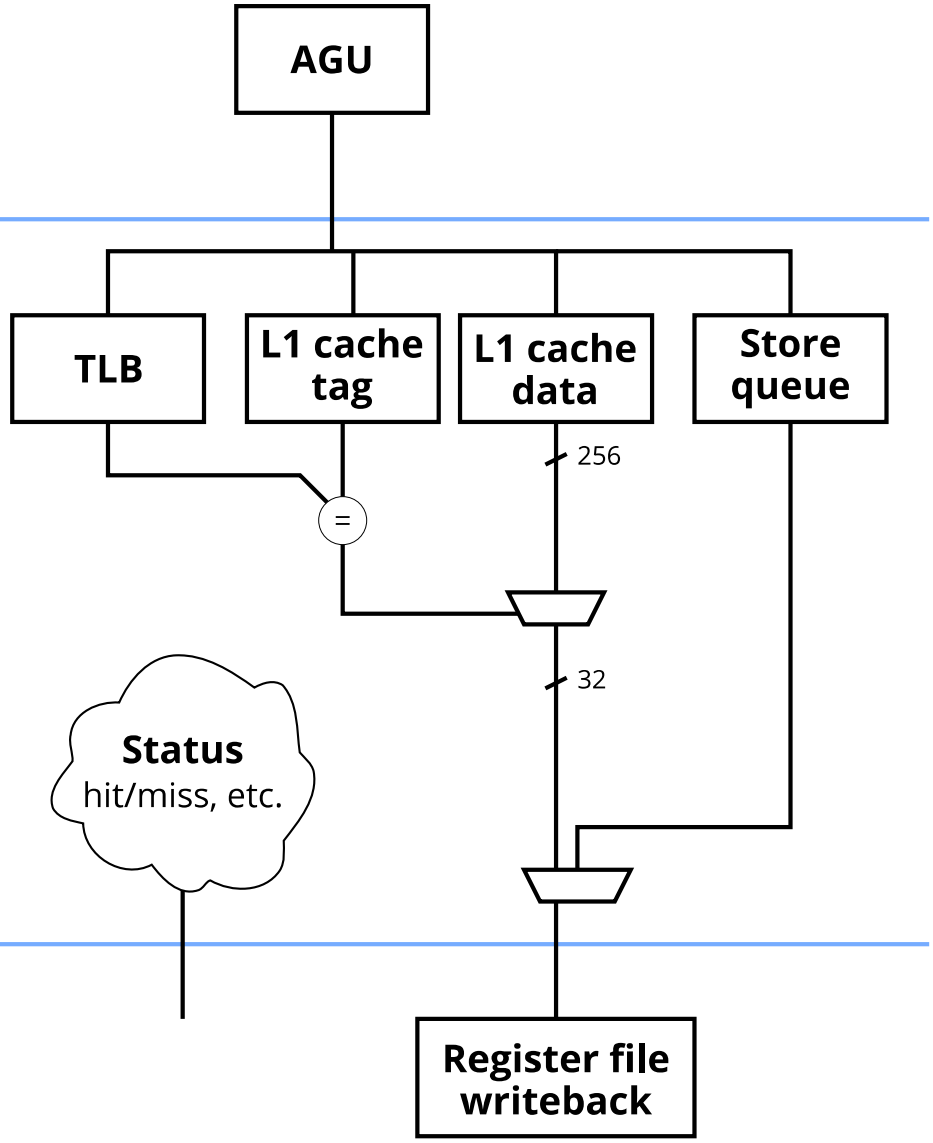
Store-forward, success

**Store-forward, failed**
Cache miss

Cache hit

# What happens to a load: simplified




**Possible outcomes for a load:**

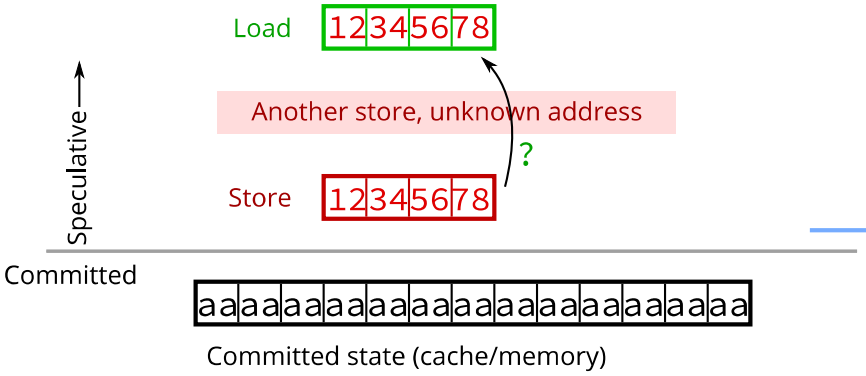
TLB miss

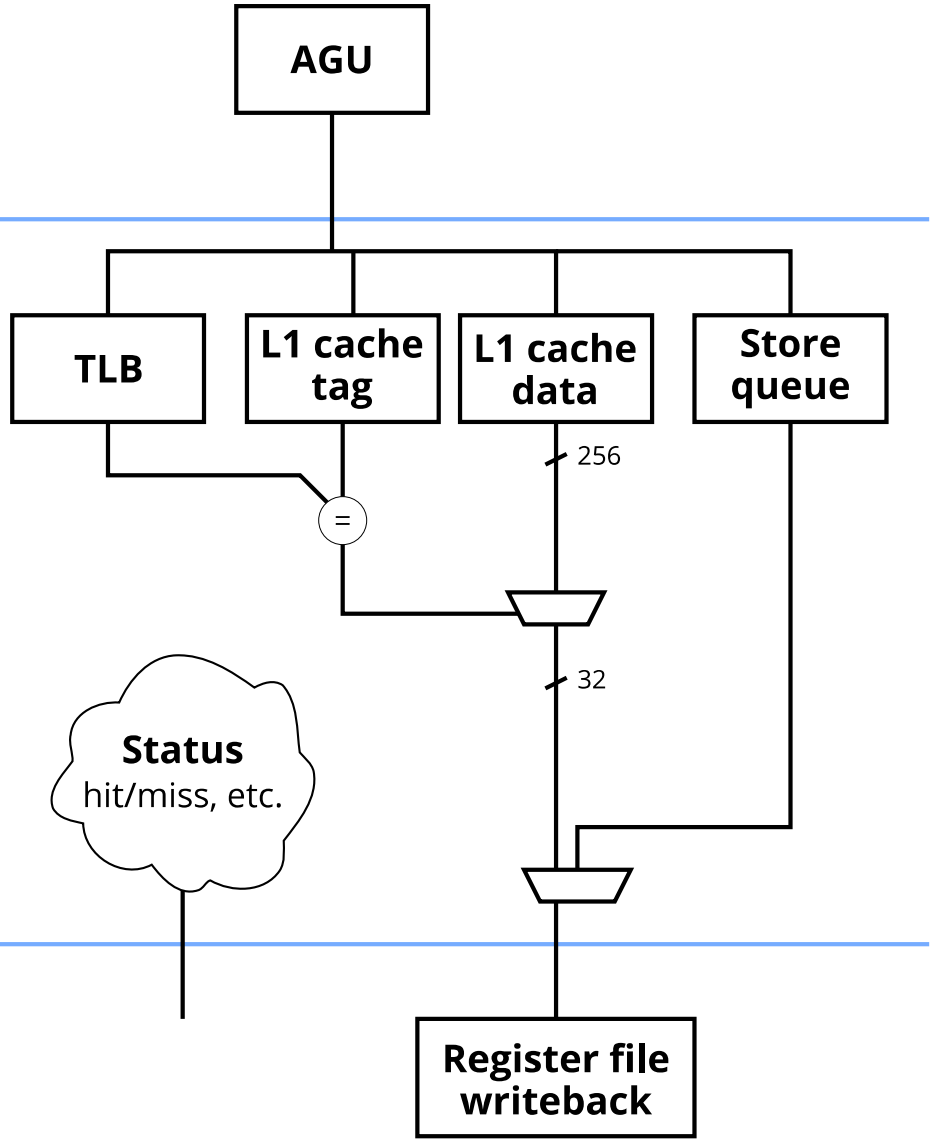
Store-forward, success
**Store-forward, not ready**

Store-forward, failed
Cache miss

Cache hit

# What happens to a load: simplified



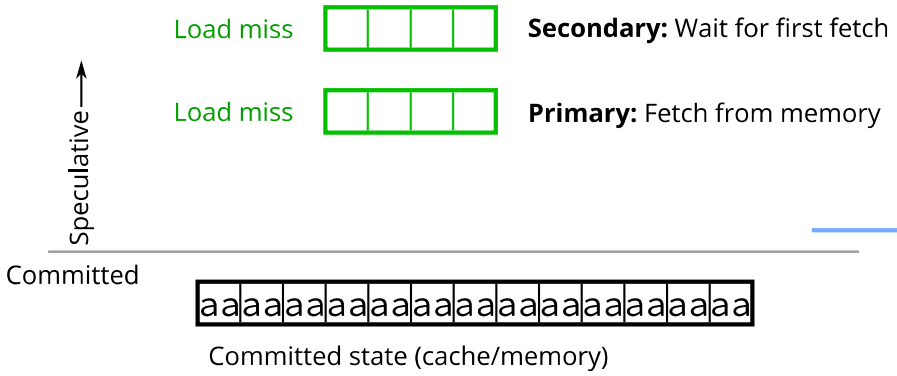

**Possible outcomes for a load:**

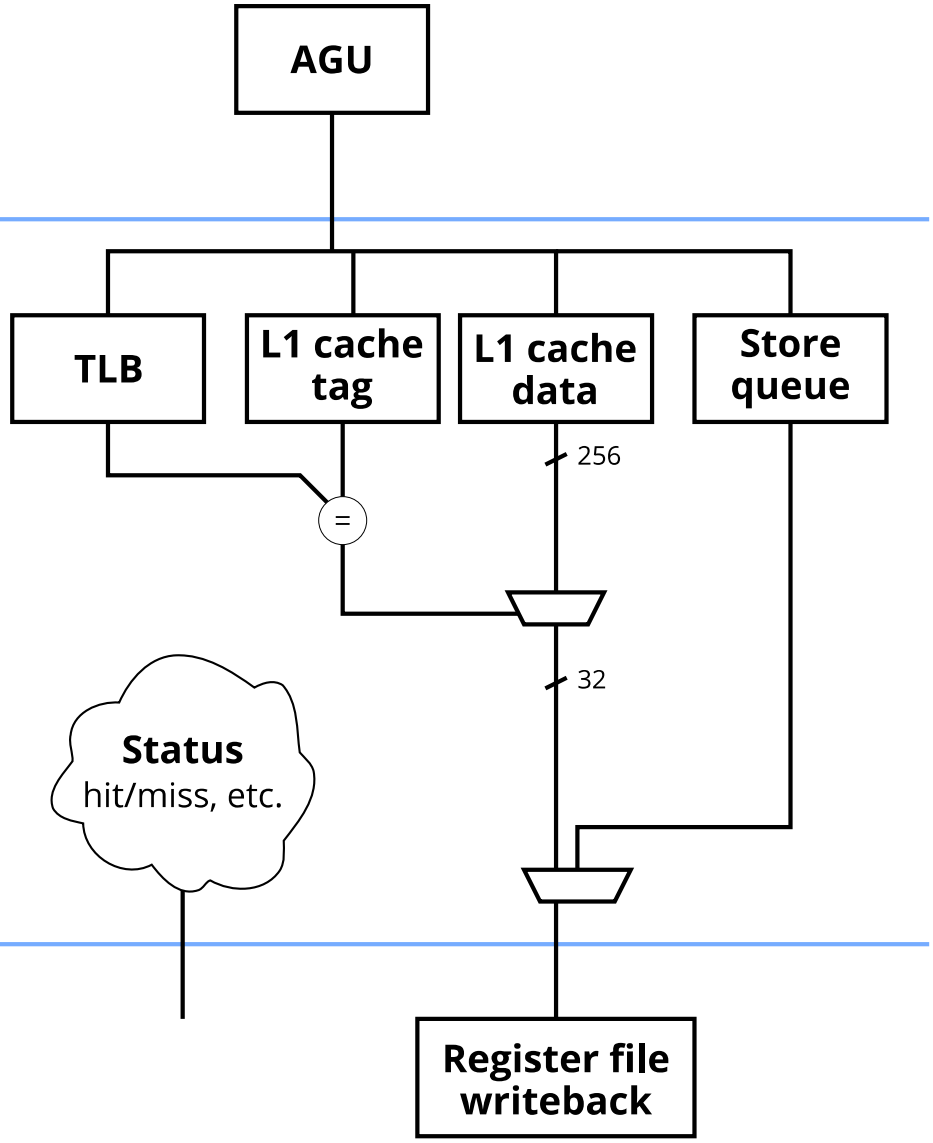
TLB miss

Store-forward, success
Store-forward, not ready
**Store-forward, ambiguous and wait**
Store-forward, failed
Cache miss

Cache hit

# What happens to a load: simplified



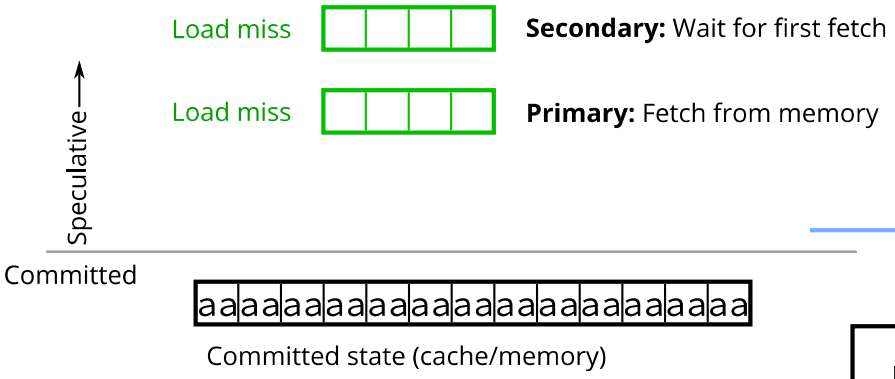

## Possible outcomes for a load:

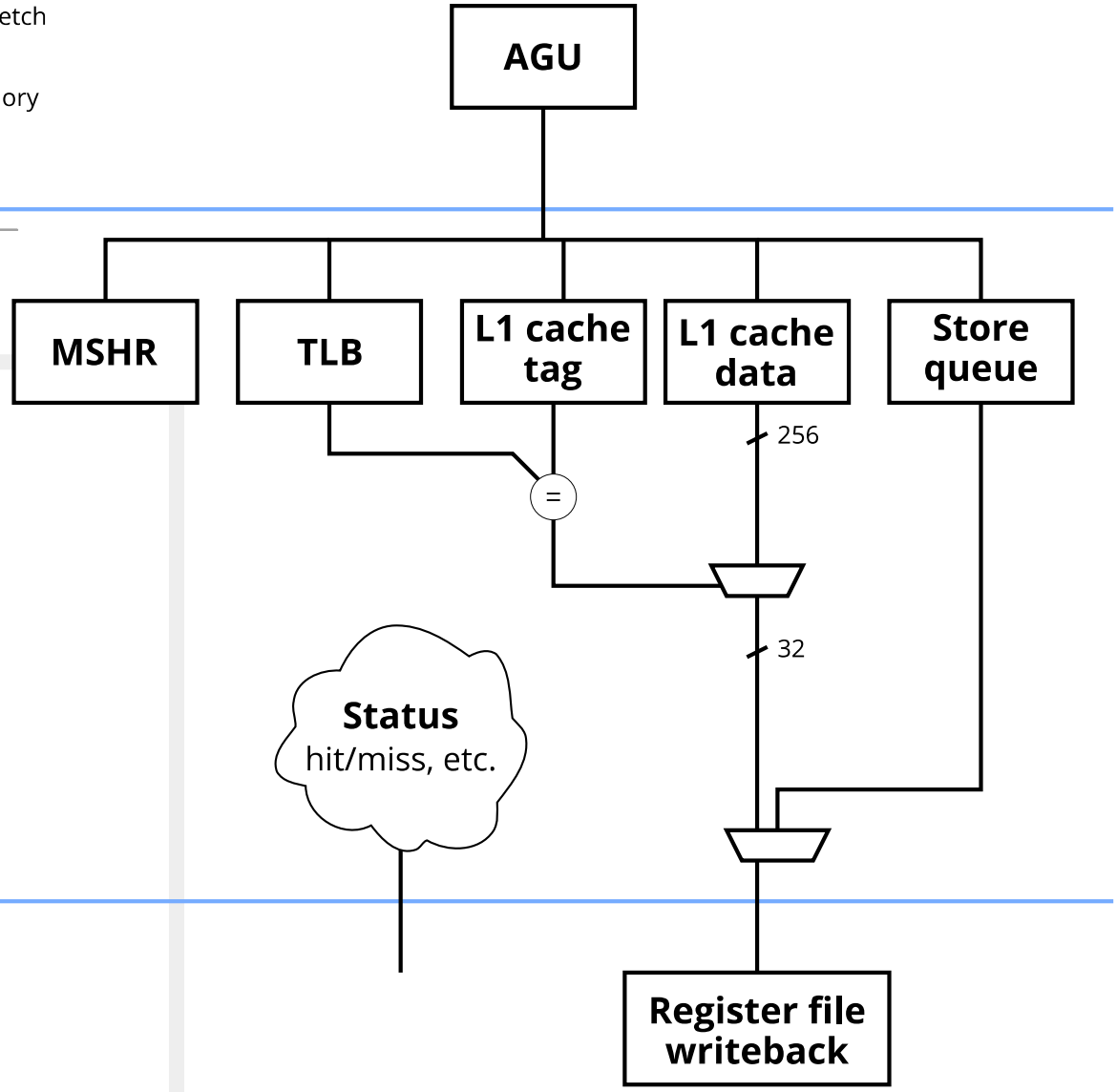
TLB miss

Store-forward, success
Store-forward, not ready
**Store-forward, ambiguous and wait**
Store-forward, failed
Cache miss

Cache hit

# What happens to a load: simplified



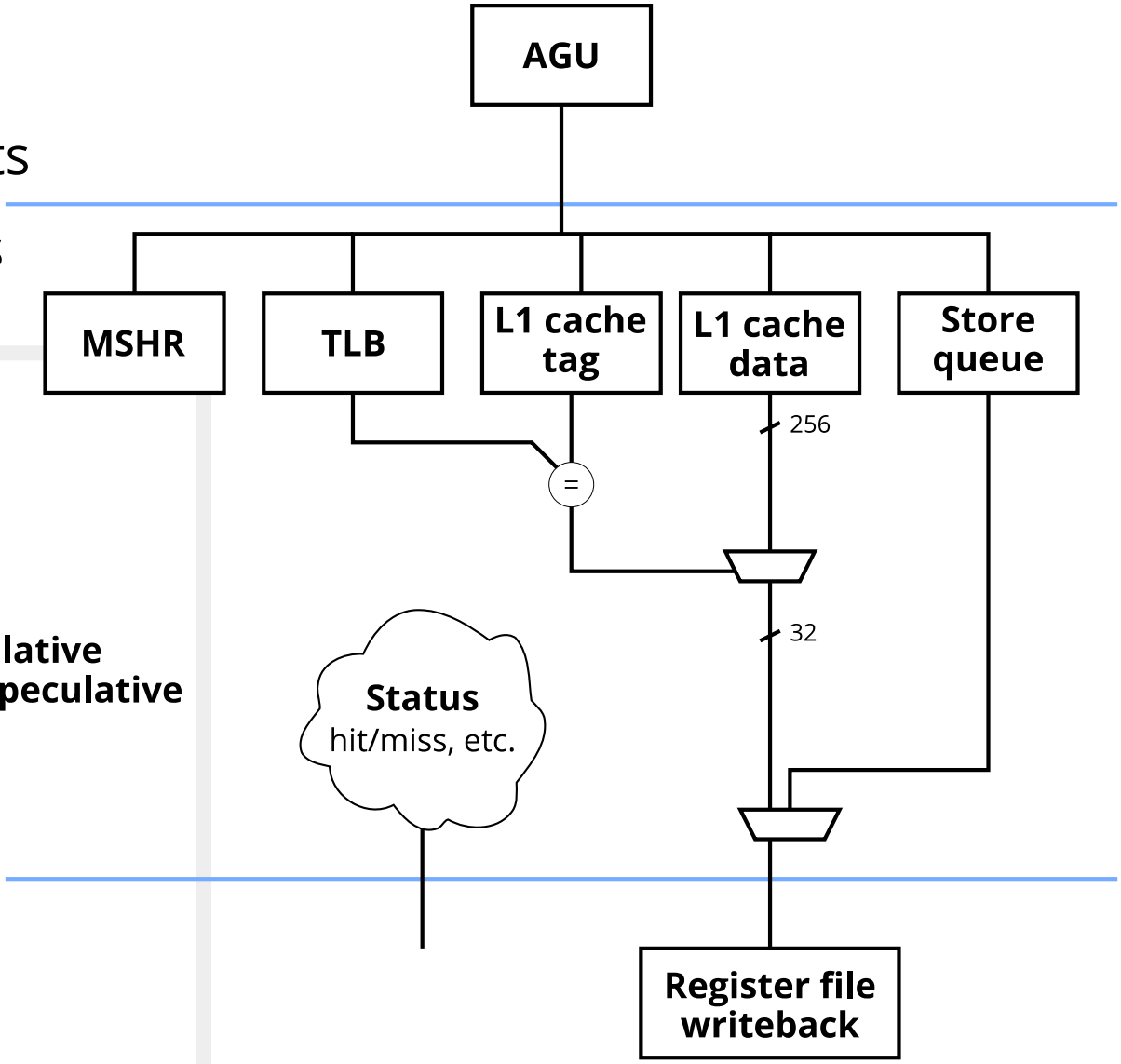

## Possible outcomes for a load:

TLB miss, primary miss
**TLB miss, secondary miss**
**TLB miss, not enough MSHR**

Store-forward, success
Store-forward, not ready
Store-forward, ambiguous and wait
Store-forward, failed
Cache miss, primary miss
**Cache miss, secondary miss (hit MSHR)**
**Cache miss, not enough MSHR**
Cache hit

# What happens to a load: simplified

- Other paging features:
  - Page faults
  - TLB Accessed/Dirty bits
  - Uncacheable accesses

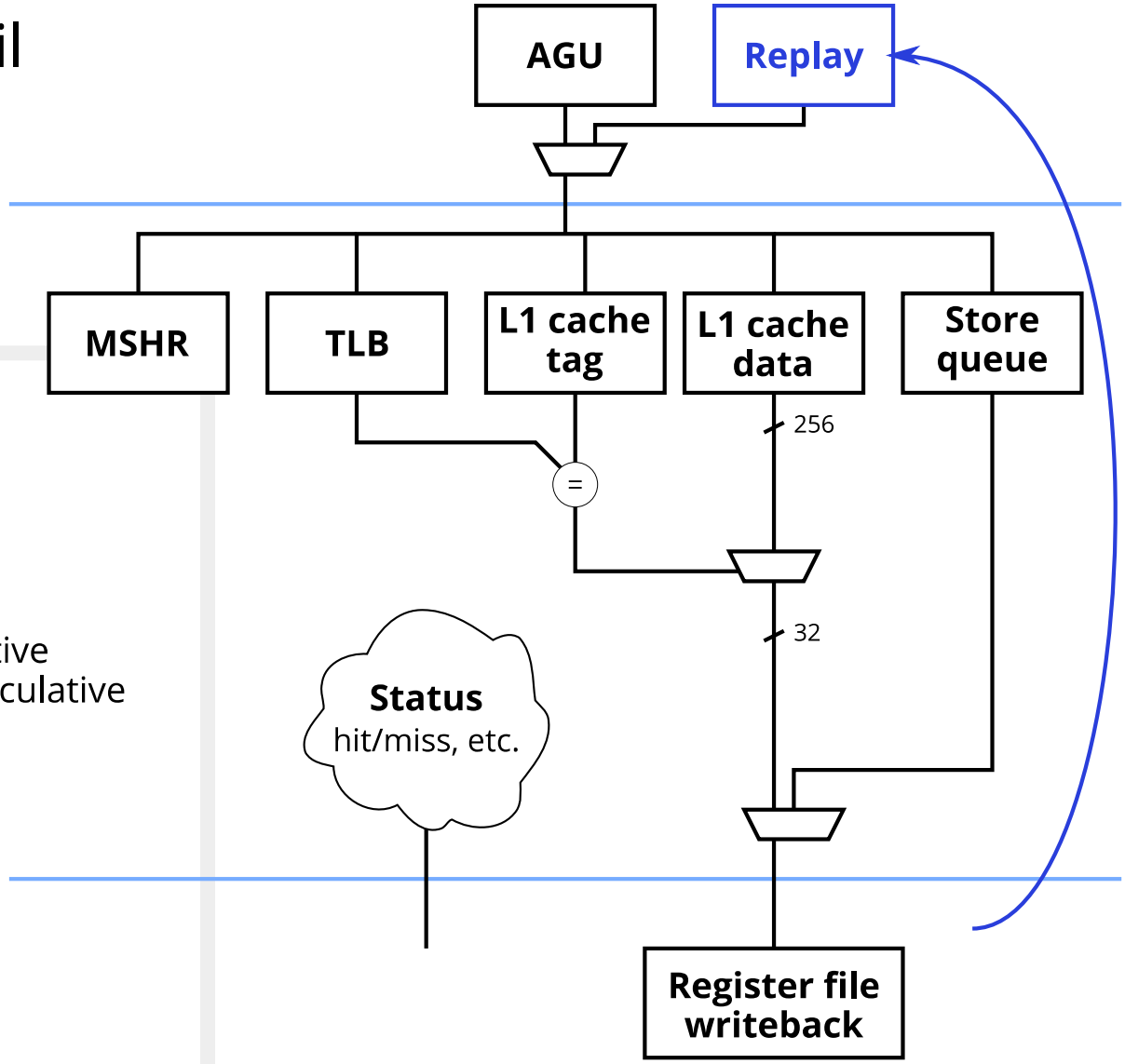



**Possible outcomes for a load:**

TLB miss, primary miss
TLB miss, secondary miss
TLB miss, not enough MSHR
**TLB page fault, speculative**
**TLB page fault, non-speculative**
**TLB hit, A or D bit needs setting, speculative**
**TLB hit, A or D bit needs setting, non-speculative**
**Uncacheable, speculative**
**Uncacheable, non-speculative**
Store-forward, success
Store-forward, not ready
Store-forward, ambiguous and wait
Store-forward, failed
Cache miss, primary miss
Cache miss, secondary miss (hit MSHR)
Cache miss, not enough MSHR
Cache hit

# What happens to a load: simplified

- Replay accesses that fail

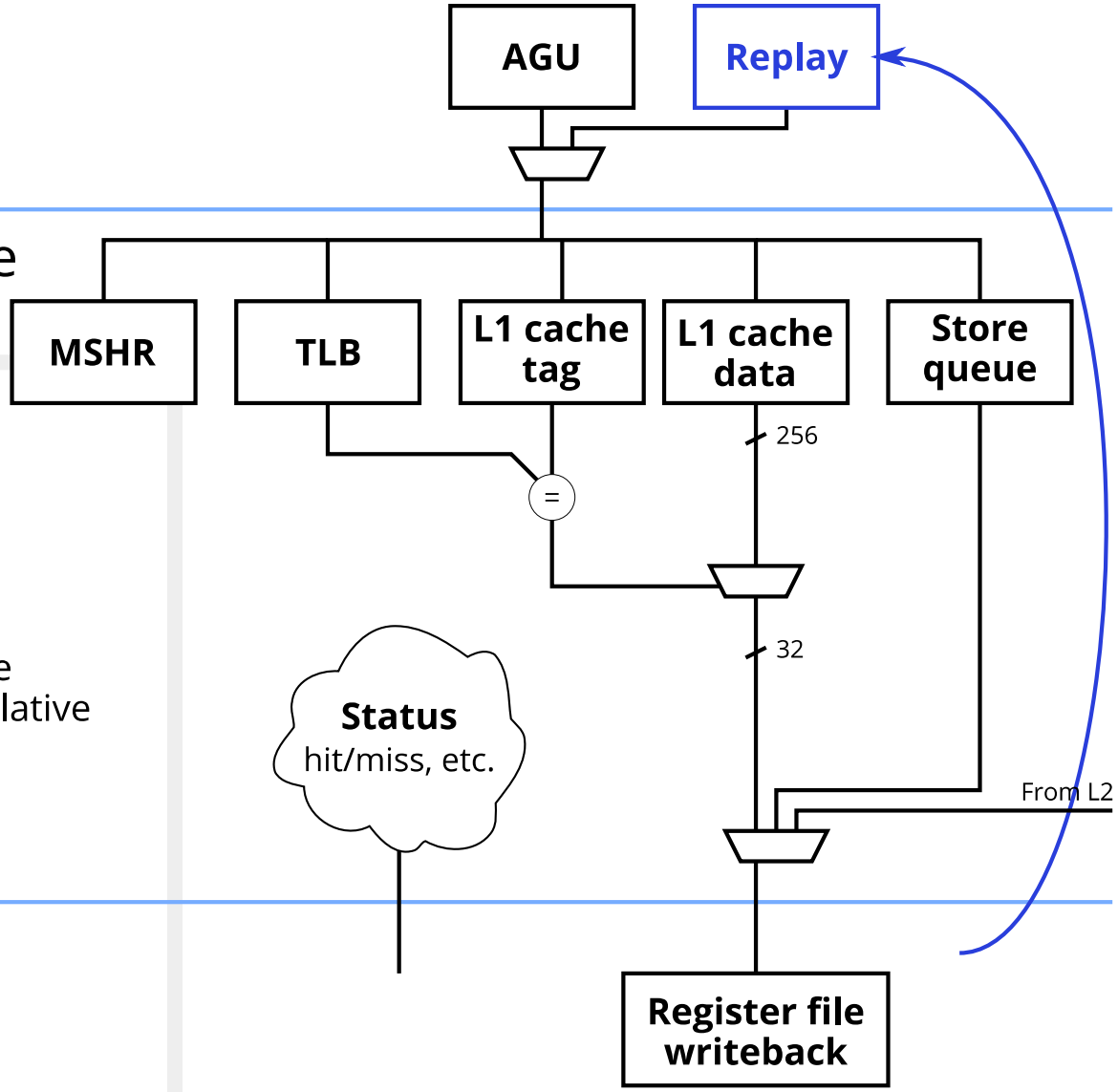



**Possible outcomes for a load:**

- TLB miss, primary miss
- TLB miss, secondary miss
- TLB miss, not enough MSHR
- TLB page fault, speculative
- TLB page fault, non-speculative
- TLB hit, A or D bit needs setting, speculative
- TLB hit, A or D bit needs setting, non-speculative
- Uncacheable, speculative
- Uncacheable, non-speculative
- Store-forward, success
- Store-forward, not ready
- Store-forward, ambiguous and wait
- Store-forward, failed
- Cache miss, primary miss
- Cache miss, secondary miss (hit MSHR)
- Cache miss, not enough MSHR
- Cache hit

# What happens to a load: simplified

- Also:
  - Split-cacheline/page
  - Stores
  - L2 and cache coherence

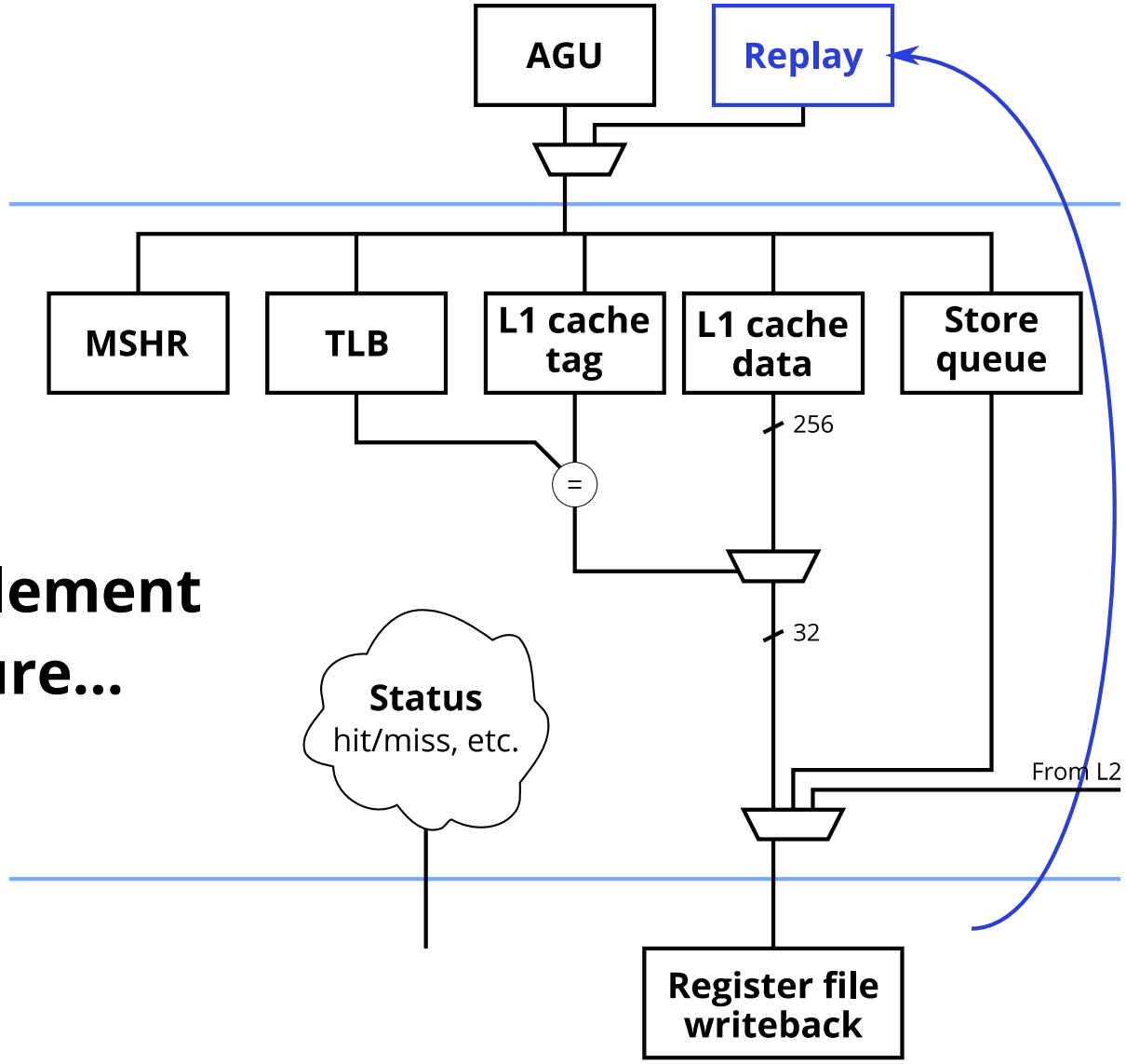



**Possible outcomes for a load:**

- TLB miss, primary miss
- TLB miss, secondary miss
- TLB miss, not enough MSHR
- TLB page fault, speculative
- TLB page fault, non-speculative
- TLB hit, A or D bit needs setting, speculative
- TLB hit, A or D bit needs setting, non-speculative
- Uncacheable, speculative
- Uncacheable, non-speculative
- Store-forward, success
- Store-forward, not ready
- Store-forward, ambiguous and wait
- Store-forward, failed
- Cache miss, primary miss
- Cache miss, secondary miss (hit MSHR)
- Cache miss, not enough MSHR
- Cache hit

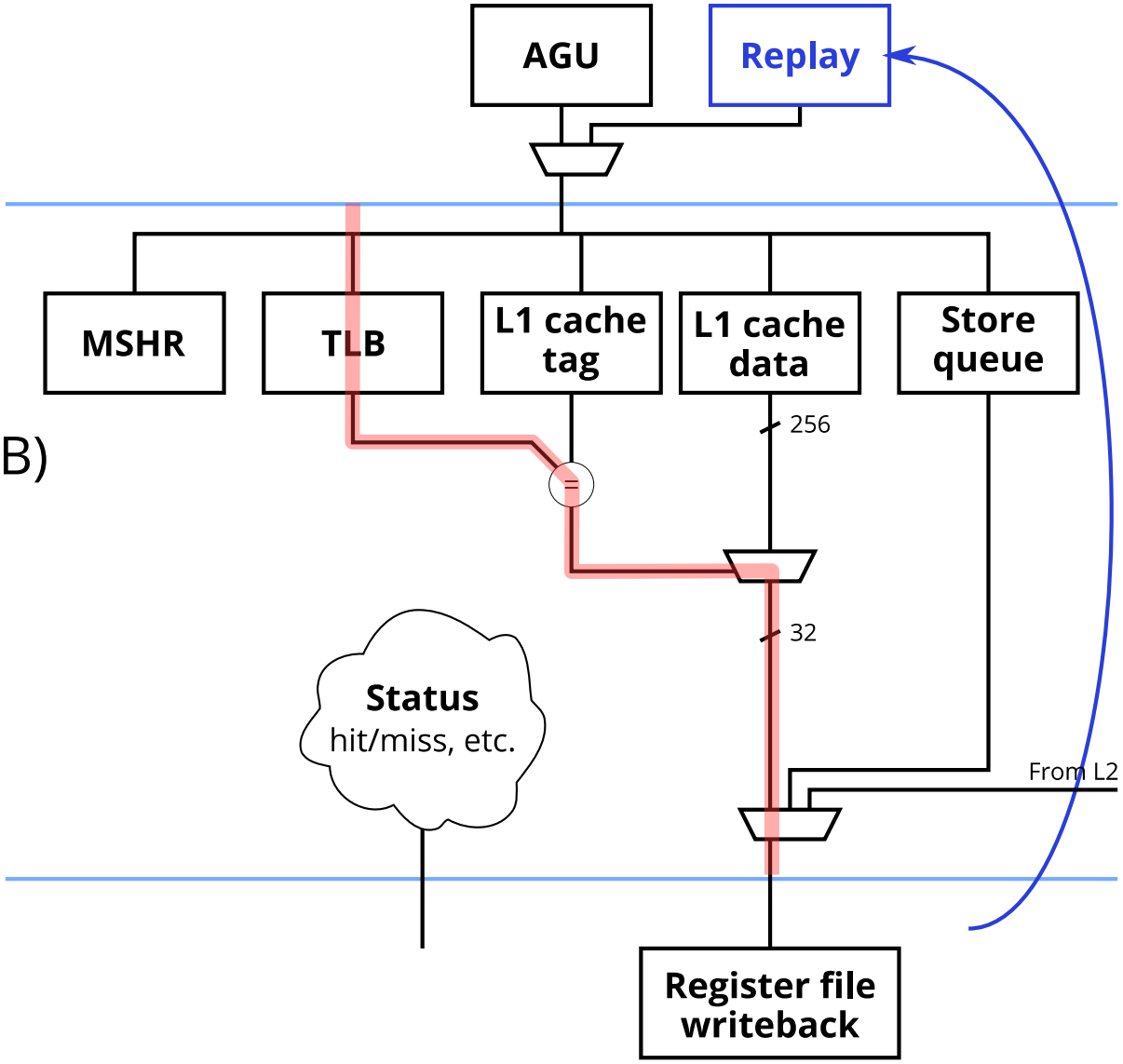
# What happens to a load: simplified

**Design a circuit to implement this microarchitecture…**



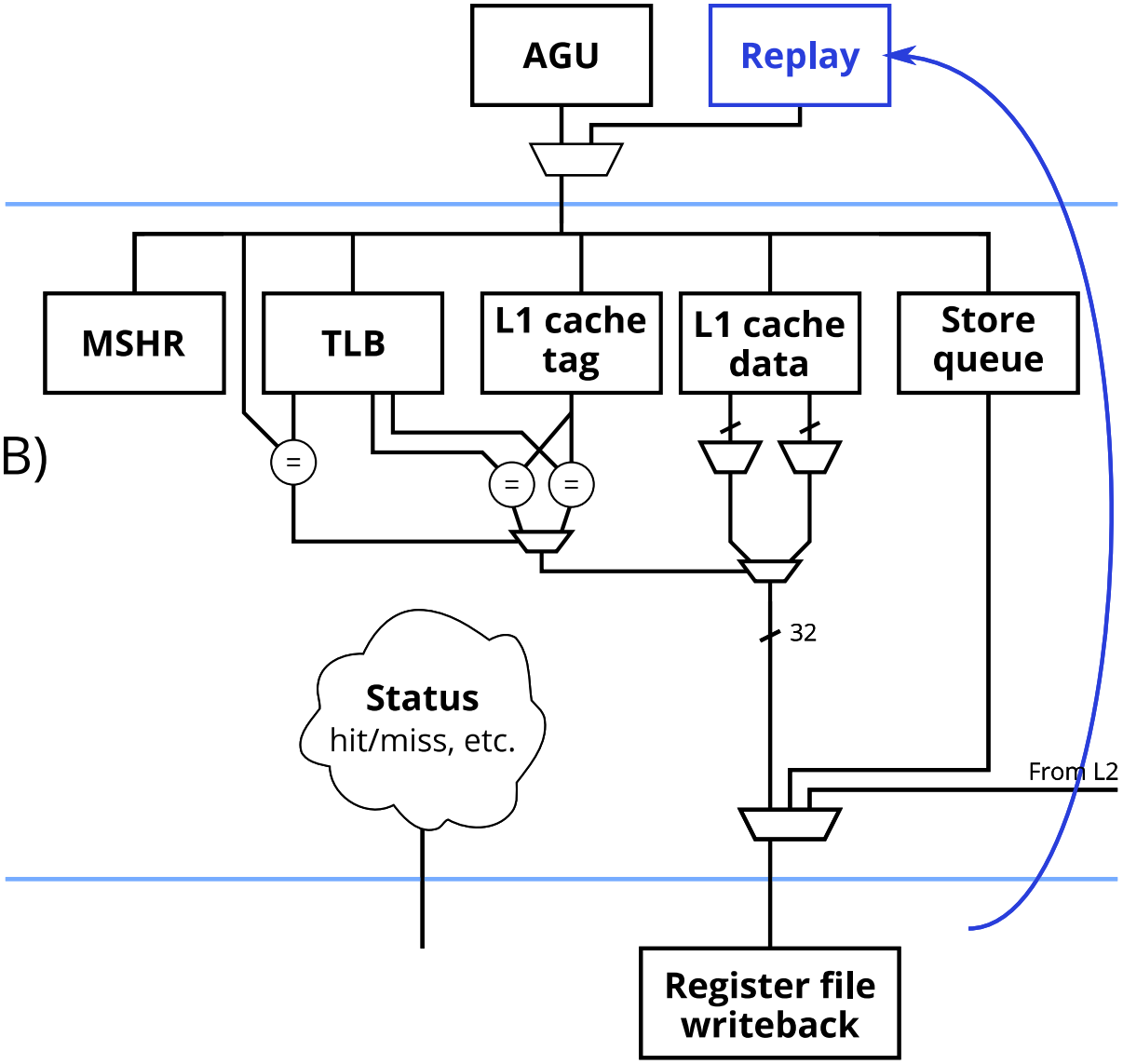
# What happens to a load: simplified

- Long critical path for direct implementation
  - TLB lookup
  - Cache tag compare
  - Cache data rotate (32 B)
- Shannon expansion: "TLB hit way 1?"
  - TLB lookup
  - 2-to-1 mux (1 bit)
  - 2-to-1 mux (32 bit)



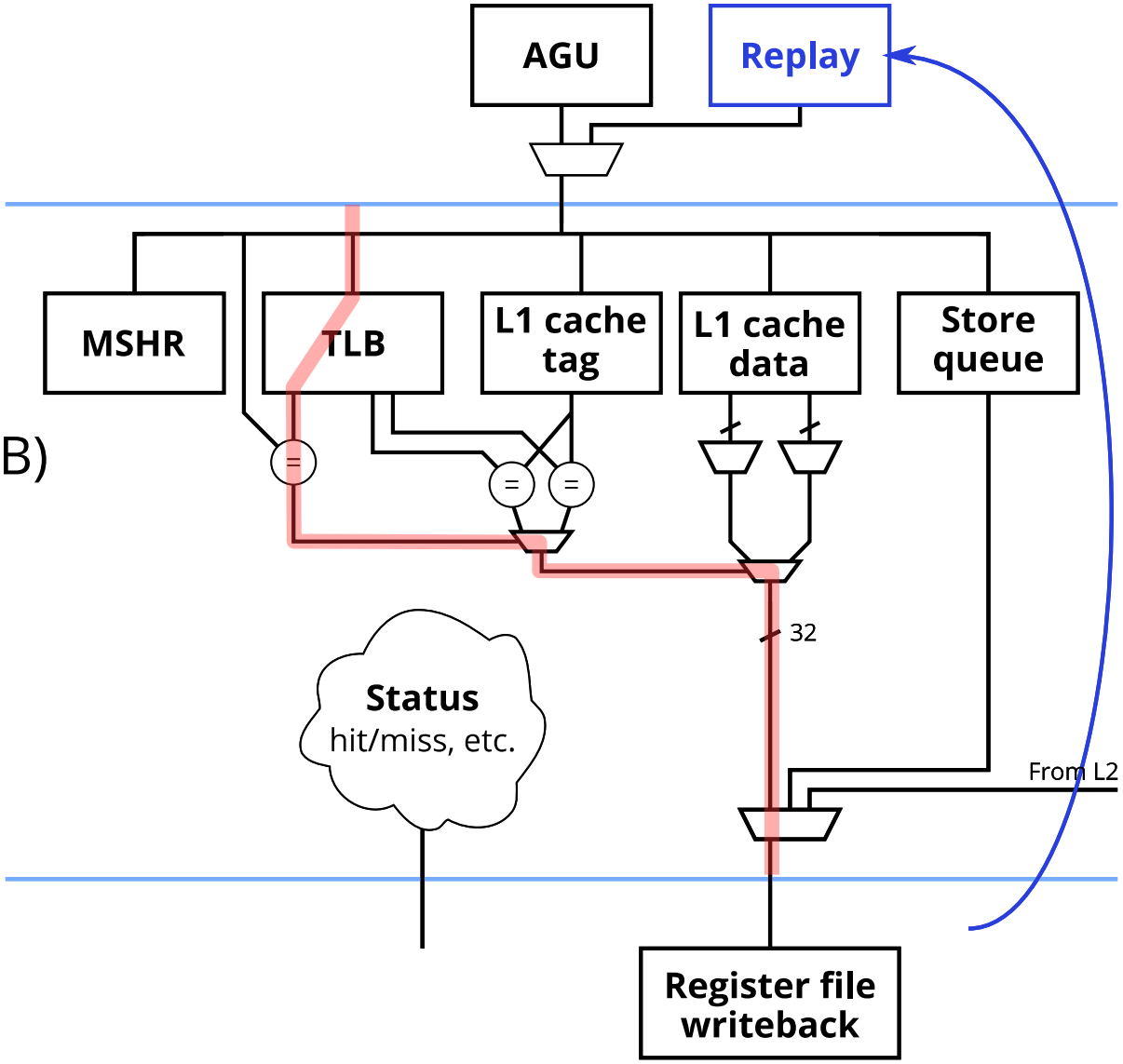
# What happens to a load: simplified

- Long critical path for direct implementation
  - TLB lookup
  - Cache tag compare
  - Cache data rotate (32 B)
- Shannon expansion: "TLB hit way 1?"
  - TLB lookup
  - 2-to-1 mux (1 bit)
  - 2-to-1 mux (32 bit)

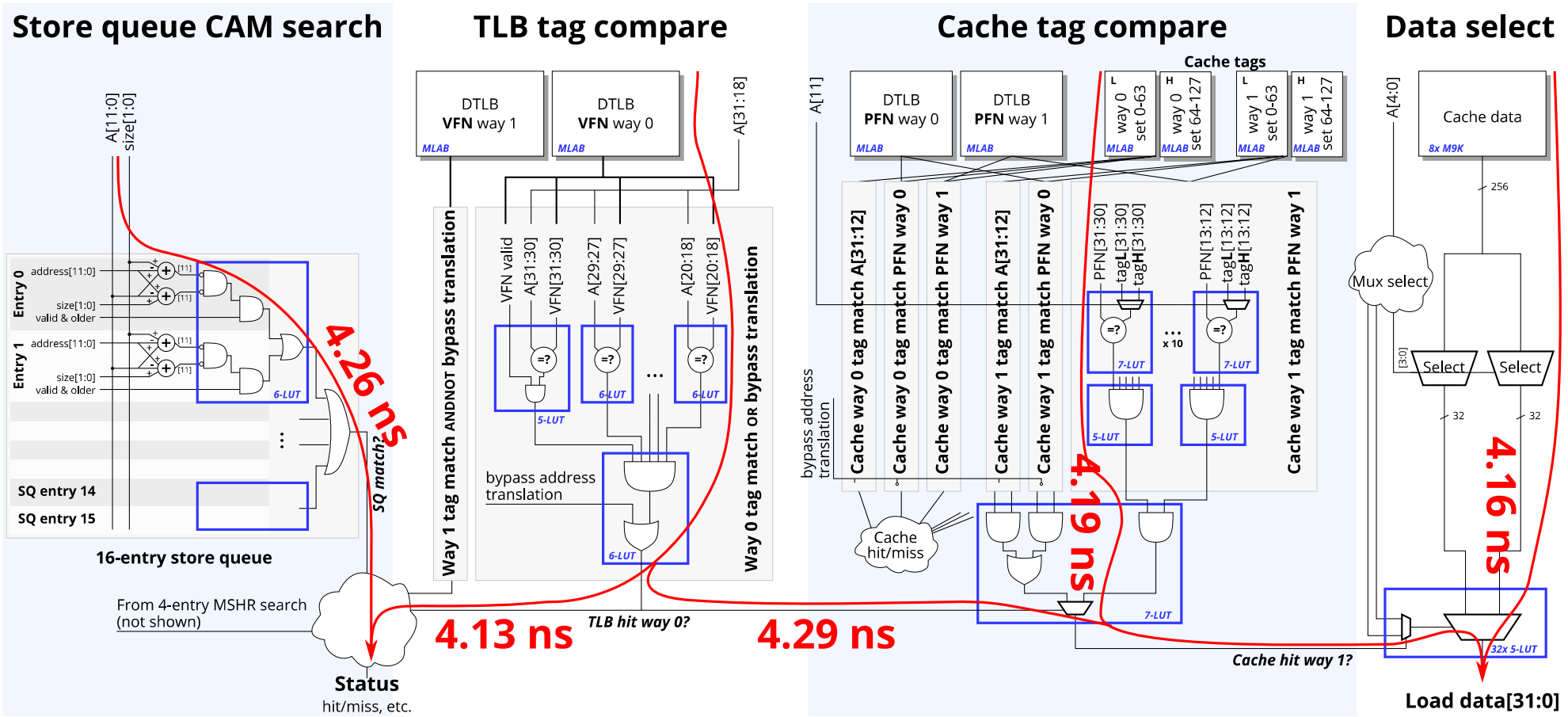


# What happens to a load: simplified

- Long critical path for direct implementation
  - TLB lookup
  - Cache tag compare
  - Cache data rotate (32 B)
- Shannon expansion: "TLB hit way 1?"
  - TLB lookup
  - 2-to-1 mux (1 bit)
  - 2-to-1 mux (32 bit)

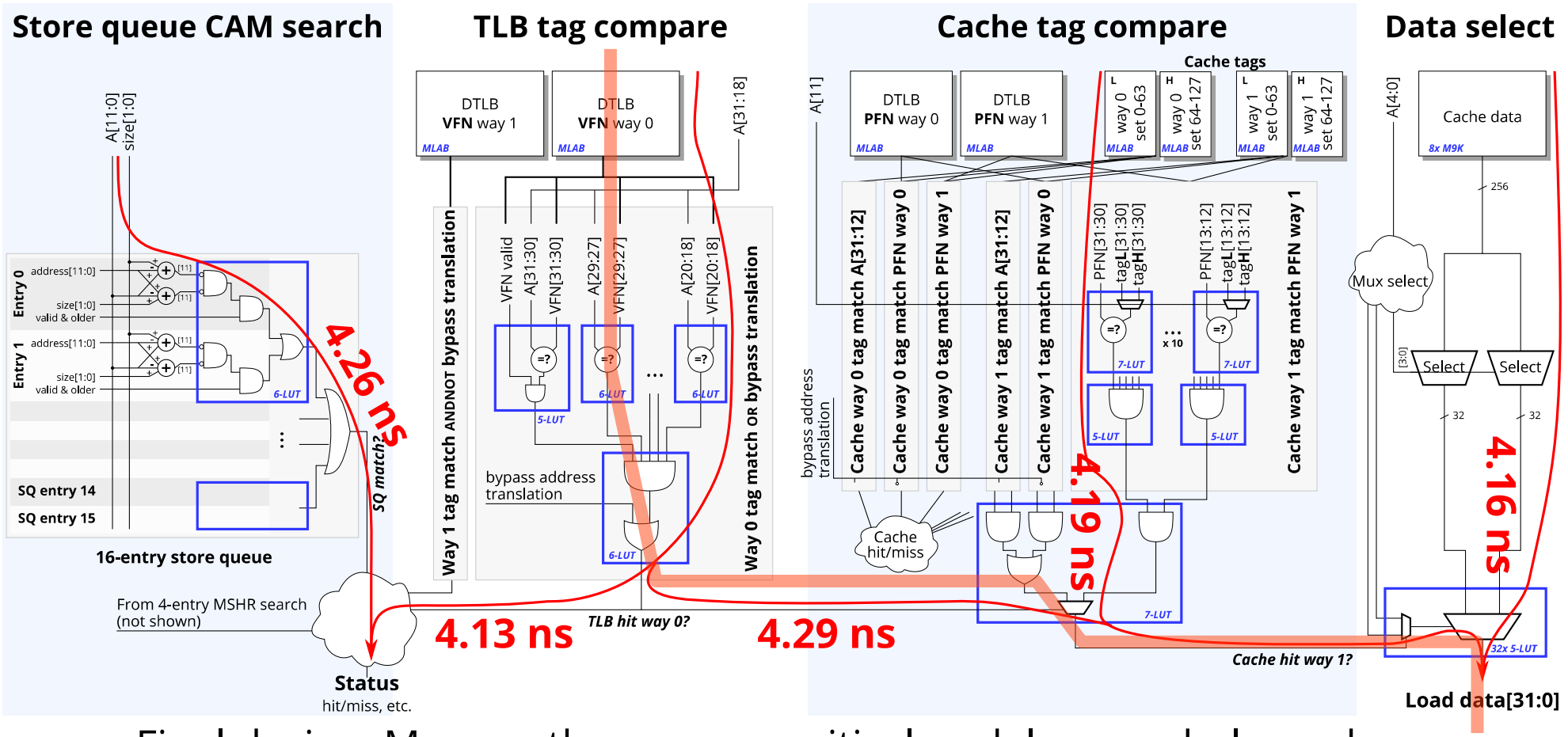


# Memory System: L1 Circuit




- Final design: Many paths are near-critical — delays are balanced
  - Critical portions hand-mapped to LUTs (blue boxes)
  - Many critical paths have logic depth of 5 LUTs
- 4.3 ns (~230 MHz)
  - Nios II/f: TLB+cache circuit is ~6.0 ns (~1.5 stages)

# Memory System: L1 Circuit




- Final design: Many paths are near-critical — delays are balanced
  - Critical portions hand-mapped to LUTs (blue boxes)
  - Many critical paths have logic depth of 5 LUTs
- 4.3 ns (~230 MHz)
  - Nios II/f: TLB+cache circuit is ~6.0 ns (~1.5 stages)

# Summary 2

- Out-of-order soft processors are useful and feasible
  - Area: **6.5×** Nios II/f, but affordable
  - Performance: **2.2×** Nios II/f on SPECint2000
- …if you pay attention to circuit design
  - Adapt microarchitecture to suit circuits
  - Or push circuits to make microarchitecture feasible
  - LUT-level design is often useful
    - Avoiding (n)optimization has been painful

# Future Work

- Hardware not yet functional
  - Some missing pieces
- More optimization
  - Frequency
- Add FPU

# End

# Thoughts on Complex Instructions

- There is little distinction between modern RISC and CISC
  - Abandoned “RISC” features
    - Branch delay slot
    - Register windows
  - “CISC” features in modern “RISC”
    - Variable-length instructions
    - Instructions with more than one micro-op
    - Unaligned memory access

# Complex Instructions 2

- It’s easier to lower than raise abstraction level
  - It’s easy to compile C to instructions.
  - It’s easy to crack complex instructions into many micro-ops.
- We’ve made a **lot** of instruction-level promises. Breaking them needs ISA changes
  - Fused Multiply-Add: Can’t synthesize from MUL and ADD
  - REP MOVS: Must maintain memory ordering
  - Binary translation is really hard: Need to keep low-level promises
- Can we really execute 5, 10, 20 IPC while keeping all instruction-level ISA promises?

# Simulator ISA Support

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0x00** | ADD | ADD | ADD | ADD | ADD | ADD | PUSH ES | POP ES | OR | OR | OR | OR | OR | OR | PUSH CS | 0x0F |
| **0x10** | ADC | ADC | ADC | ADC | ADC | ADC | PUSH SS | POP SS | SBB | SBB | SBB | SBB | SBB | SBB | PUSH DS | POP DS |
| **0x20** | AND | AND | AND | AND | AND | AND | ES | DAA | SUB | SUB | SUB | SUB | SUB | SUB | CS | DAS |
| **0x30** | XOR | XOR | XOR | XOR | XOR | XOR | SS | AAA | CMP | CMP | CMP | CMP | CMP | CMP | DS | AAS |
| **0x40** | INC | | | | | | | | DEC | | | | | | | |
| **0x50** | PUSH | | | | | | | | POP | | | | | | | |
| **0x60** | PUSHA | POPA | BOUND | ARPL | FS | GS | OS | AS | PUSH | IMUL | PUSH | IMUL | INS | INS | OUTS | OUTS |
| **0x70** | JO | JNO | JB | JNB | JZ | JNZ | JBE | JNBE | JS | JNS | JP | JNP | JL | JNL | JLE | JNLE |
| **0x80** | ALU | ALU | ALU | ALU | TEST | TEST | XCHG | XCHG | MOV | | | | MOV SREG | LEA | MOV SREG | POP |
| **0x90** | NOP | XCHG | | | | | | | CWDE | CDQ | CALL far | WAIT | PUSHF | POPF | SAHF | LAHF |
| **0xa0** | MOV | | MOV | | MOVS | MOVS | CMPS | CMPS | TEST | TEST | STOS | STOS | LODS | LODS | SCAS | SCAS |
| **0xb0** | MOV | | | | | | | | MOV | | | | | | | |
| **0xc0** | RO/SH | RO/SH | RET | RET | LES | LDS | MOV | MOV | ENTER | LEAVE | RET far | RET far | INT3 | INT | INTO | IRET |
| **0xd0** | RO/SH | RO/SH | RO/SH | RO/SH | AAM | AAD | SALC | XLAT | FPU | | | | | | | |
| **0xe0** | LOOPNZ | LOOPZ | LOOP | JCXZ | IN | IN | OUT | OUT | CALL | JMP | JMP far | JMP | IN | IN | OUT | OUT |
| **0xf0** | LOCK | INT1 | REP/NZ | REP/Z | HLT | CMC | ALU | ALU | CLC | STC | CLI | STI | CLD | STD | INC/DEC | messy |
| **0f 00** | yucky | yucky++ | LAR | LSL | | | CLTS | | INVD | WBINVD | | UD2 | | NOP | | |
| **0f 10** | SSE | | | | | | | | prefetch/hints | | | | | | | |
| **0f 20** | MOV CR | MOV DR | MOV CR | MOV DR | | | | | SSE | | | | | | | |
| **0f 30** | WRMSR | RDTSC | RDMSR | RDPMC | SYSENTER | SYSEXIT | | | SSE | | SSE | | | | | |
| **0f 40** | CMOVO | CMOVNO | CMOVB | CMOVNB | CMOVZ | CMOVNZ | CMOVBE | CMOVNBE | CMOVS | CMOVNS | CMOVP | CMOVNP | CMOVL | CMOVNL | CMOVLE | CMOVNLE |
| **0f 50** | SSE | | | | | | | | | | | | | | | |
| **0f 60** | MMX/SSE | | | | | | | | | | | | | | | |
| **0f 70** | SSE | MMX/SSE | | | | | | | VMREAD | VMWRITE | | | SSE | | MMX/SSE | |
| **0f 80** | JO | JNO | JB | JNB | JZ | JNZ | JBE | JNBE | JS | JNS | JP | JNP | JL | JNL | JLE | JNLE |
| **0f 90** | SETO | SETNO | SETB | SETNB | SETZ | SETNZ | SETBE | SETNBE | SETS | SETNS | SETP | SETNP | SETL | SETNL | SETLE | SETNLE |
| **0f a0** | PUSH FS | POP FS | CPUID | BT | SHLD | SHLD | | | PUSH GS | POP GS | RSM | BTS | SHRD | SHRD | FENCE | IMUL |
| **0f b0** | CMPXCHG | CMPXCHG | LSS | BTR | LFS | LGS | MOVZX | MOVZX | | UD | BT | BTC | BSF | BSR | MOVSX | MOVSX |
| **0f c0** | XADD | XADD | MMX/SSE | | | | | CX8B | BSWAP | | | | | | | |
| **0f d0** | MMX/SSE | | | | | | | | | | | | | | | |
| **0f e0** | MMX/SSE | | | | | | | | | | | | | | | |
| **0f f0** | MMX/SSE | | | | | | | | | | | | | | | |

# Publications

- **Comparing FPGA to custom CMOS**
  - Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture (*FPGA 2011*)
  - Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design (*TVLSI, 2013*)
- **Out-of-order memory system**
  - Efficient methods for out-of-order load/store execution for high-performance soft processors (*FPT 2013*)
  - Microarchitecture and circuits for a 200 MHz Out-of-order soft processor memory system (*TRETS 2016*)
- **Out-of-order instruction scheduling**
  - High performance instruction scheduling circuits for out-of-order soft processors (*FCCM 2016)*
  - High performance instruction scheduling circuits for superscalar out-of-order soft processors (*TRETS 2017)*