mp_ooo

Part 3

Topics

- How's it going
  - Any documentation wishes?
  - Anything that should have been w/ code prior to distribution?
- DRAM no longer on exam
- Patch releasing today
  - Test suite
  - Comp code
    - Memory model
    - Superscalar RVFI
  - Verilator
  - See Campuwire post for details

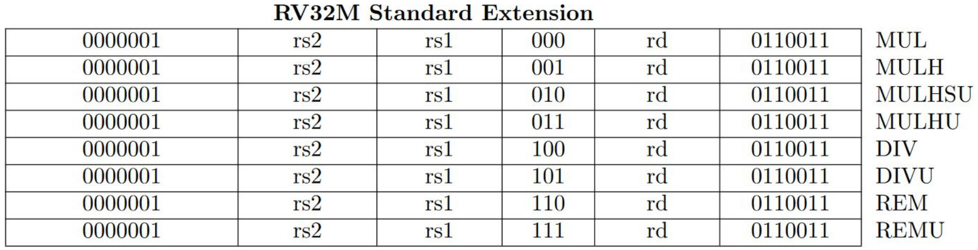
Precursor

Checkpoint 3 Overview

- Add support for control instructions
- Add support for memory instructions
- Move to more realistic memory hierarchy
  - Memory -> Burst memory (competition memory)
  - Integrate instruction and data caches
- Run Coremark

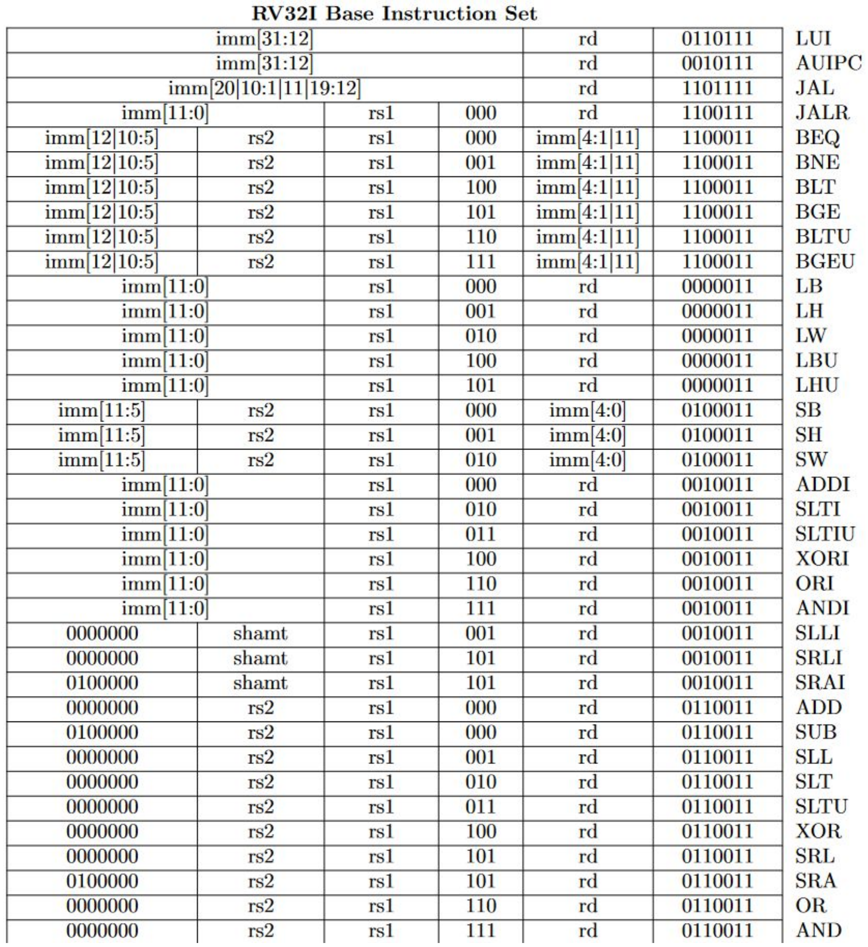
Where You're At

- That's a lot of instructions to do . . .



RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU



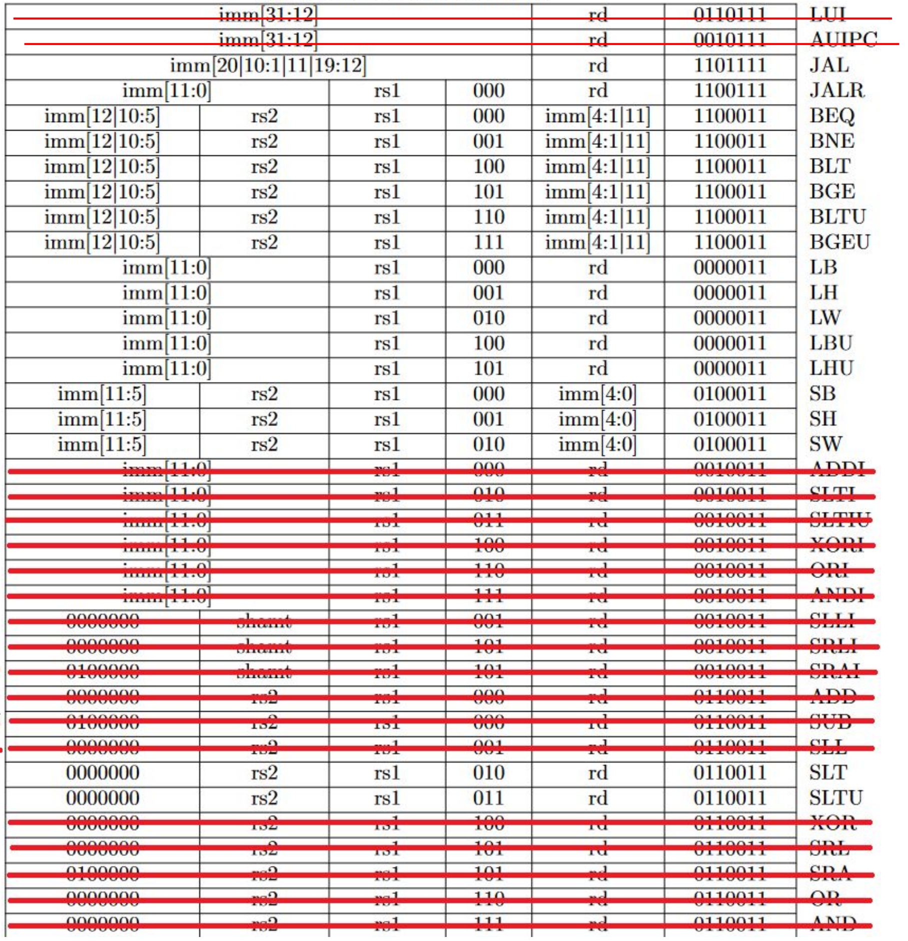
RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Where You're At

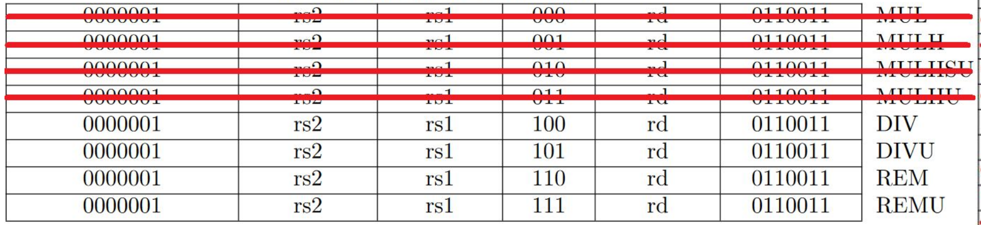
- That's a lot of instructions to do . . .
    - But you have done a lot!

RV32I Base Instruction Set



~~imm[31:12]~~				~~rd~~	~~0110111~~	~~LUI~~
~~imm[31:12]~~				~~rd~~	~~0010111~~	~~AUIPC~~
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
~~imm[11:0]~~		~~rs1~~	~~000~~	~~rd~~	~~0010011~~	~~ADDI~~
~~imm[11:0]~~		~~rs1~~	~~010~~	~~rd~~	~~0010011~~	~~SLTI~~
~~imm[11:0]~~		~~rs1~~	~~011~~	~~rd~~	~~0010011~~	~~SLTIU~~
~~imm[11:0]~~		~~rs1~~	~~100~~	~~rd~~	~~0010011~~	~~XORI~~
~~imm[11:0]~~		~~rs1~~	~~110~~	~~rd~~	~~0010011~~	~~ORI~~
~~imm[11:0]~~		~~rs1~~	~~111~~	~~rd~~	~~0010011~~	~~ANDI~~
~~0000000~~	~~shamt~~	~~rs1~~	~~001~~	~~rd~~	~~0010011~~	~~SLLI~~
~~0000000~~	~~shamt~~	~~rs1~~	~~101~~	~~rd~~	~~0010011~~	~~SRLI~~
~~0100000~~	~~shamt~~	~~rs1~~	~~101~~	~~rd~~	~~0010011~~	~~SRAI~~
~~0000000~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~ADD~~
~~0100000~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~SUB~~
~~0000000~~	~~rs2~~	~~rs1~~	~~001~~	~~rd~~	~~0110011~~	~~SLL~~
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
~~0000000~~	~~rs2~~	~~rs1~~	~~100~~	~~rd~~	~~0110011~~	~~XOR~~
~~0000000~~	~~rs2~~	~~rs1~~	~~101~~	~~rd~~	~~0110011~~	~~SRL~~
~~0100000~~	~~rs2~~	~~rs1~~	~~101~~	~~rd~~	~~0110011~~	~~SRA~~
~~0000000~~	~~rs2~~	~~rs1~~	~~110~~	~~rd~~	~~0110011~~	~~OR~~
~~0000000~~	~~rs2~~	~~rs1~~	~~111~~	~~rd~~	~~0110011~~	~~AND~~

RV32M Standard Extension

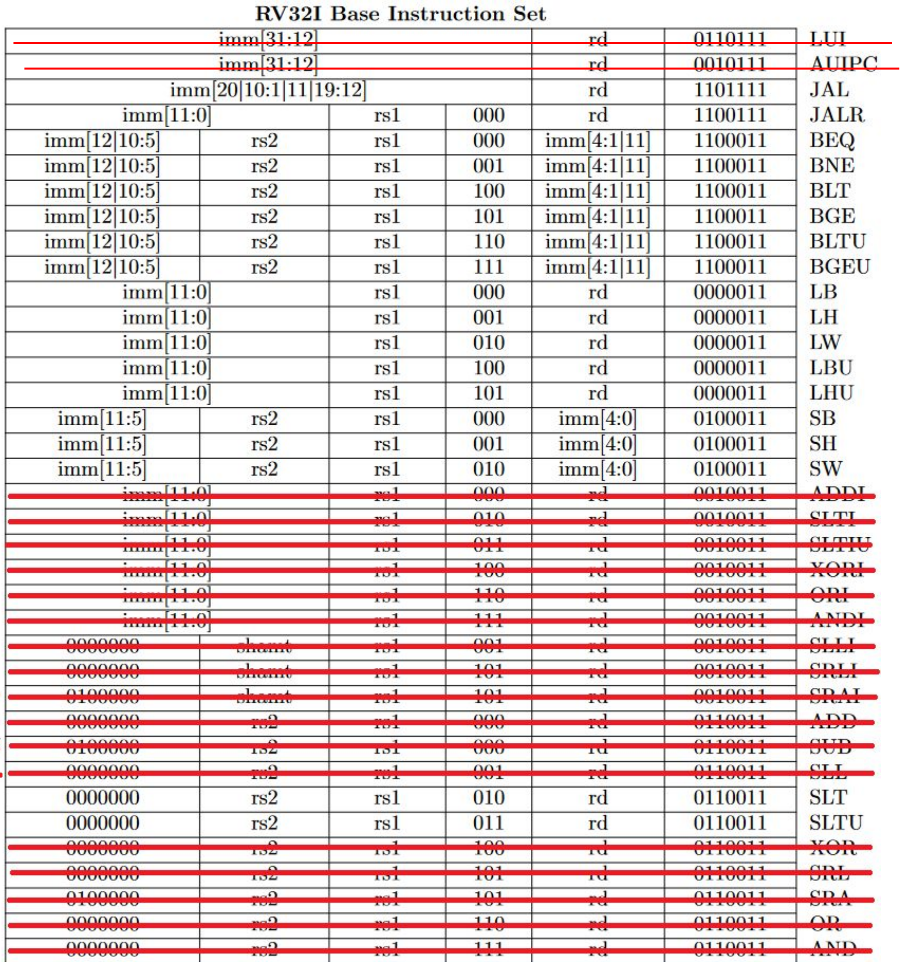


~~0000001~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~MUL~~
~~0000001~~	~~rs2~~	~~rs1~~	~~001~~	~~rd~~	~~0110011~~	~~MULH~~
~~0000001~~	~~rs2~~	~~rs1~~	~~010~~	~~rd~~	~~0110011~~	~~MULHSU~~
~~0000001~~	~~rs2~~	~~rs1~~	~~011~~	~~rd~~	~~0110011~~	~~MULHU~~
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Where You’re At

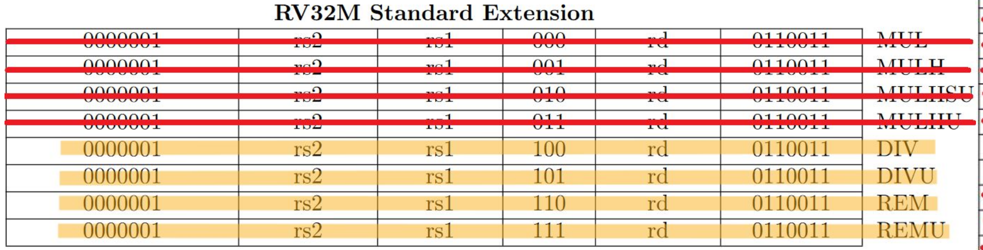
- That’s a lot of instructions to do . . .
  - But you have done a lot!
  - Other M instructions not required

    But are advanced feature points



RV32I Base Instruction Set

~~imm[31:12]~~				~~rd~~	~~0110111~~	~~LUI~~
~~imm[31:12]~~				~~rd~~	~~0010111~~	~~AUIPC~~
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
~~imm[11:0]~~		~~rs1~~	~~000~~	~~rd~~	~~0010011~~	~~ADDI~~
~~imm[11:0]~~		~~rs1~~	~~010~~	~~rd~~	~~0010011~~	~~SLTI~~
~~imm[11:0]~~		~~rs1~~	~~011~~	~~rd~~	~~0010011~~	~~SLTIU~~
~~imm[11:0]~~		~~rs1~~	~~100~~	~~rd~~	~~0010011~~	~~XORI~~
~~imm[11:0]~~		~~rs1~~	~~110~~	~~rd~~	~~0010011~~	~~ORI~~
~~imm[11:0]~~		~~rs1~~	~~111~~	~~rd~~	~~0010011~~	~~ANDI~~
~~0000000~~	~~shamt~~	~~rs1~~	~~001~~	~~rd~~	~~0010011~~	~~SLLI~~
~~0000000~~	~~shamt~~	~~rs1~~	~~101~~	~~rd~~	~~0010011~~	~~SRLI~~
~~0100000~~	~~shamt~~	~~rs1~~	~~101~~	~~rd~~	~~0010011~~	~~SRAI~~
~~0000000~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~ADD~~
~~0100000~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~SUB~~
~~0000000~~	~~rs2~~	~~rs1~~	~~001~~	~~rd~~	~~0110011~~	~~SLL~~
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
~~0000000~~	~~rs2~~	~~rs1~~	~~100~~	~~rd~~	~~0110011~~	~~XOR~~
~~0000000~~	~~rs2~~	~~rs1~~	~~101~~	~~rd~~	~~0110011~~	~~SRL~~
~~0100000~~	~~rs2~~	~~rs1~~	~~101~~	~~rd~~	~~0110011~~	~~SRA~~
~~0000000~~	~~rs2~~	~~rs1~~	~~110~~	~~rd~~	~~0110011~~	~~OR~~
~~0000000~~	~~rs2~~	~~rs1~~	~~111~~	~~rd~~	~~0110011~~	~~AND~~

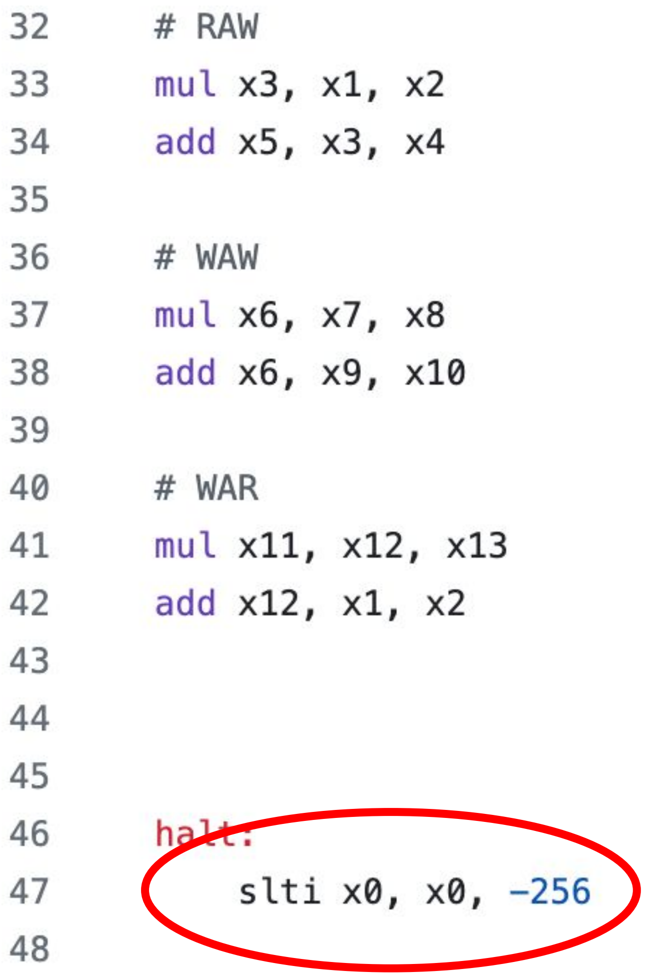


RV32M Standard Extension

~~0000001~~	~~rs2~~	~~rs1~~	~~000~~	~~rd~~	~~0110011~~	~~MUL~~
~~0000001~~	~~rs2~~	~~rs1~~	~~001~~	~~rd~~	~~0110011~~	~~MULH~~
~~0000001~~	~~rs2~~	~~rs1~~	~~010~~	~~rd~~	~~0110011~~	~~MULHSU~~
~~0000001~~	~~rs2~~	~~rs1~~	~~011~~	~~rd~~	~~0110011~~	~~MULHU~~
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

The "Hidden" Instruction

- CP2 requires arithmetic inst
- CP3 requires control & memory inst
  - There is an instruction that maybe wasn't explicitly highlighted in these requirements
    - SLT/SLTU -> Uses the CMP not ALU!
- CP2 test code doesn't actually check SLT
- Ensure you have a CMP

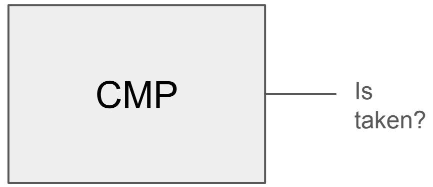


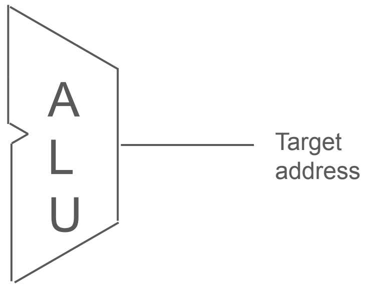
```
# RAW
mul x3, x1, x2
add x5, x3, x4

# WAW
mul x6, x7, x8
add x6, x9, x10

# WAR
mul x11, x12, x13
add x12, x1, x2



halt:
    slti x0, x0, -256

```

Control

Branch Calculation Hardware

- What did we need to compute branches in mp_pipeline?
  - Evaluate if branch condition is met
  - Compute branch target address
- Needs two pieces of hardware
  - ALU
  - CMP
- Is this weird for OoO?
  - Have to synchronize between two different FUs
- Have a branch unit
  - CMP and simple adder
    - And maybe use this too for SLT(U)?
- Best allocation depends on program





Flushing on Mispredictions

- Flushing is made of two parts:
    - Restoring the architectural state of the processor
    - Updating the PC to start fetching new instructions

Restoring Architectural State (Tomasulo)

When committing the mispredicted control instruction:

- Flush the ROB → Mark all entries as invalid.
- Flush the instruction queue and any other pipeline registers in the front end
- Flush all the reservation stations and functional units

- Mark all data entries in the register file as valid

Restoring Architectural State (ERR)

When committing the mispredicted control instruction:

- Flush the ROB → Mark all entries as invalid.
- Flush the instruction queue and any other pipeline registers
- Flush all the reservation stations and functional units

- Load the RAT with all the mappings in the RRAT/RRF
- Mark the Free list as full (head == tail and top bits of the pointers are the inverse of each other).

Updating the PC

- PC + b/j_imm or rs1 + i_imm needs to be calculated, stored somewhere, then used to update the pc when the control instruction commits.

Recommendation:

Make a queue - Control Buffer

It contains the target address for all control instructions

On commit, dequeue. (

On flush, flush the queue and update the PC with the head entry target address.

Memory Unit

Memory Stall

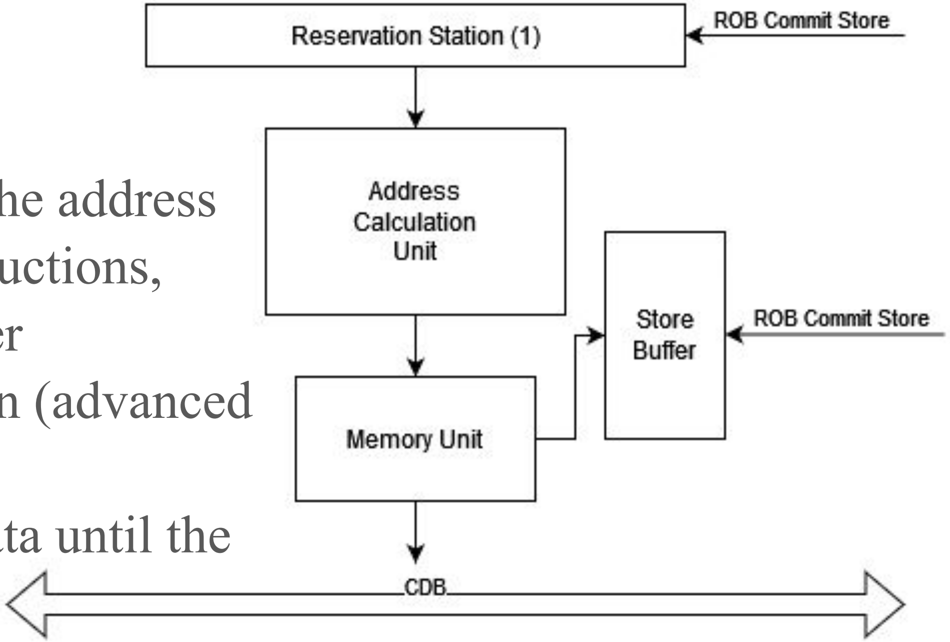
- Arithmetic operations need dependency checks to execute OoO
  - Do memory operations need the same?
- Memory instructions also have data hazards
  - Stores - **W**rite to an address
  - Load - **R**ead from an address
    - Still have RAW, WAW, WAR!
- Can we resolve these hazards in the same way as registers?
  - No

Memory Execution

- Stores only write to memory on commit from ROB
  - They cannot be speculative!
- Loads can execute whenever, as long as there aren’t any hazards
  - How can we ensure there are no hazards?

(Simple) Memory Unit

- Address Calculation: Calculates the address
- Memory unit: Executes load instructions, populates the (1 entry) store buffer
  - Does memory disambiguation (advanced feature)
- Store buffer - Stores address, wdata until the ROB is committing the store
- All memory instructions executed in order wrt to each other, loads OoO relative to other instructions.



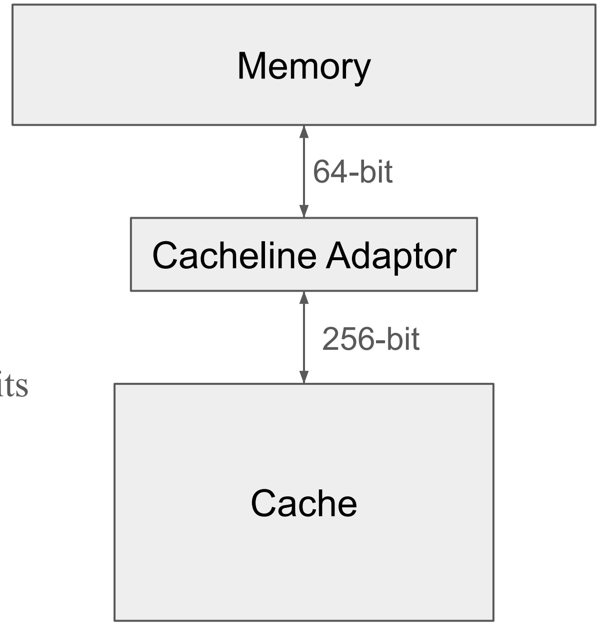
Memory Hierarchy

Competition Memory / Burst Memory

- CP1 & CP2 memory is ideal
    - Response data w/ perfect size of cacheline
    - Responds in one transaction
    - Has two independent ports
    - Responds instantly
- Real memory is not so luxurious
    - Response data is a static 64 bits
    - Responds in bursts
    - Has only one port (let’s ignore channels . . .)
    - Response time is variable
        - More importantly slow . . .

Competition Memory / Burst Memory - Annoyances from Reality

- Memory delay is based on time NOT clock cycles
  - Memory doesn’t speed up as your processor does
  - Emphasizes strong caching, prefetching
    - These are advanced features points for a reason. . .
- Respond w/ 4 burst messages
  - Need to collect the 4 bursts before handing to cache*
    - Cacheline adaptor
  - Modify generate_memory_file.sh -> ADDRESSABILITY=32
    - A single address now has 32 bytes from 4 bursts of 64 bits
    - (4 bursts * 64 bits/burst = 256 bits/8 = 32 bytes)



*This is assuming a cacheline size of 256 bits from mp_cache. You are permitted to change this.

Arbiter

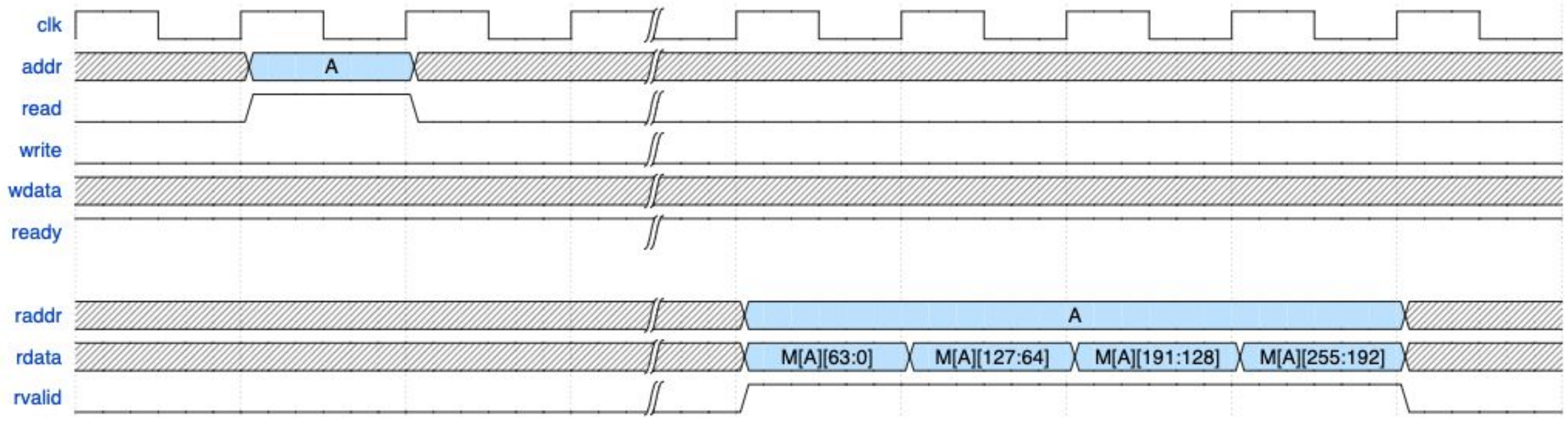
- 2 caches, 1 memory port
    - Who gets to use it?
        - Answered a similar question with the FUs and CDB in CP2
- Manage who get to use memory port
    - Simple state machine that shouldn’t need more than 3 states

Competition Memory

- Ports from perspective of memory
    - Input = CPU -> Memory
    - Output = Memory -> CPU
- Some port changes from previous memory
    - No wresp!
        - SDRAM has specific timings that are respected instead
    - rvalid - Read response (rresp)
    - raddr - The address corresponding to the presented rdata
    - ready - Indicates if memory can keep taking requests

```
modport mem (
    input       clk,
    input       rst,

    input       addr,
    input       read,
    input       write,
    input       wdata,
    output      ready,

    output      rdata,
    output      raddr,
    output      rvalid,

    output      error
);
```

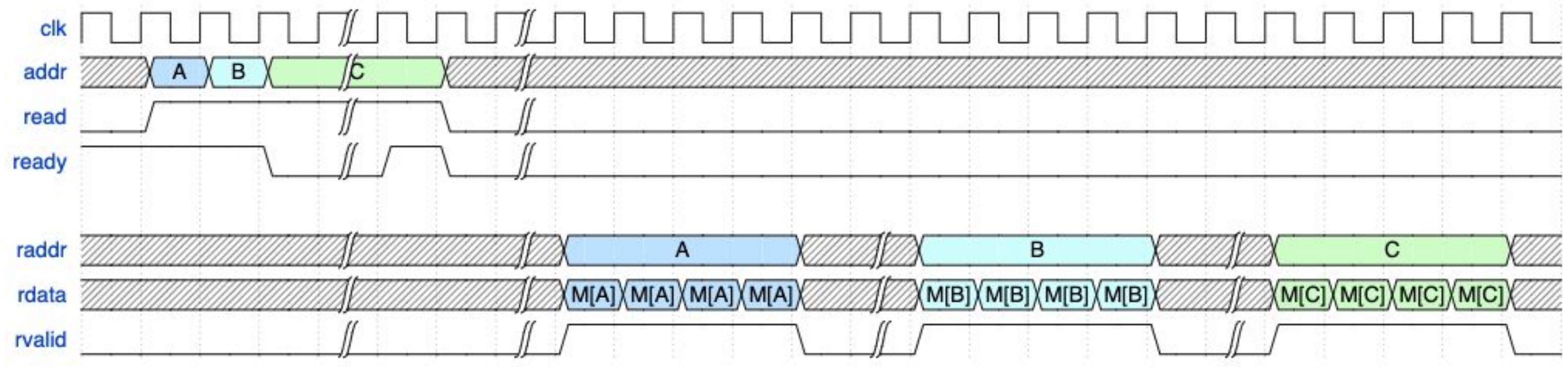
Competition Memory / Burst Memory - Reads



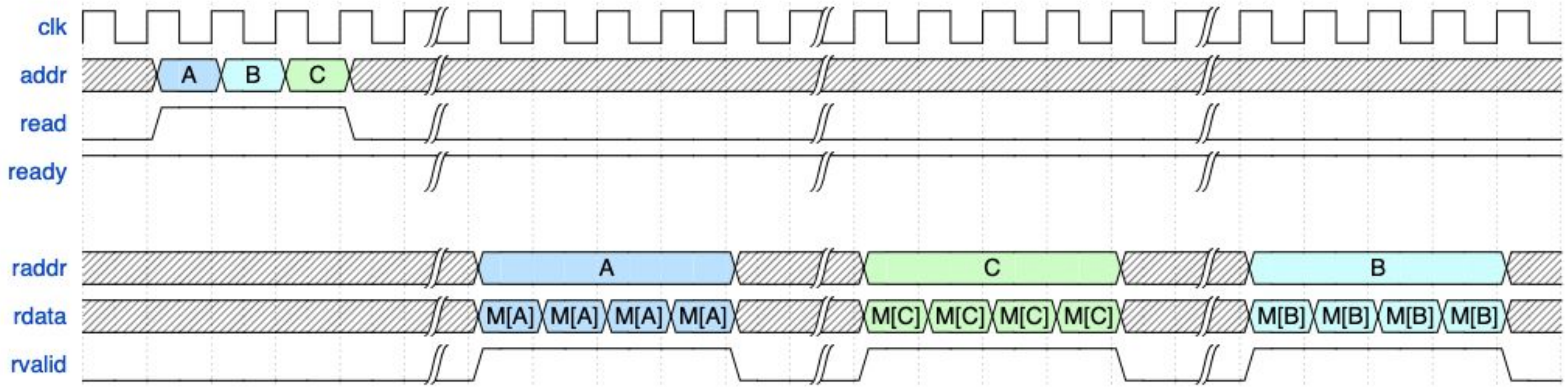
Single Read Request

Competition Memory / Burst Memory - Reads

Read Queue Full

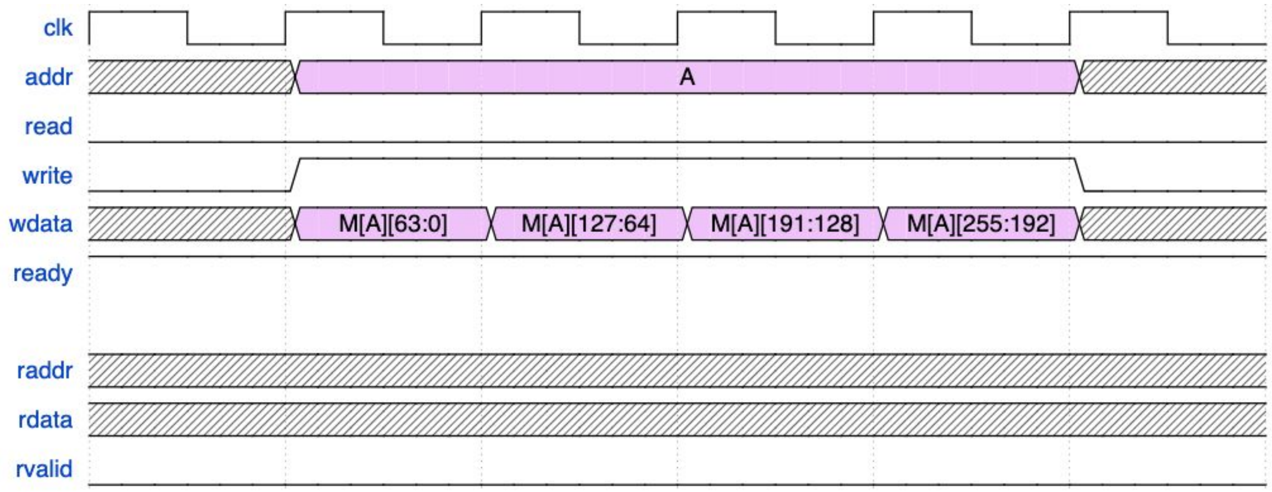


OoO Read Resp

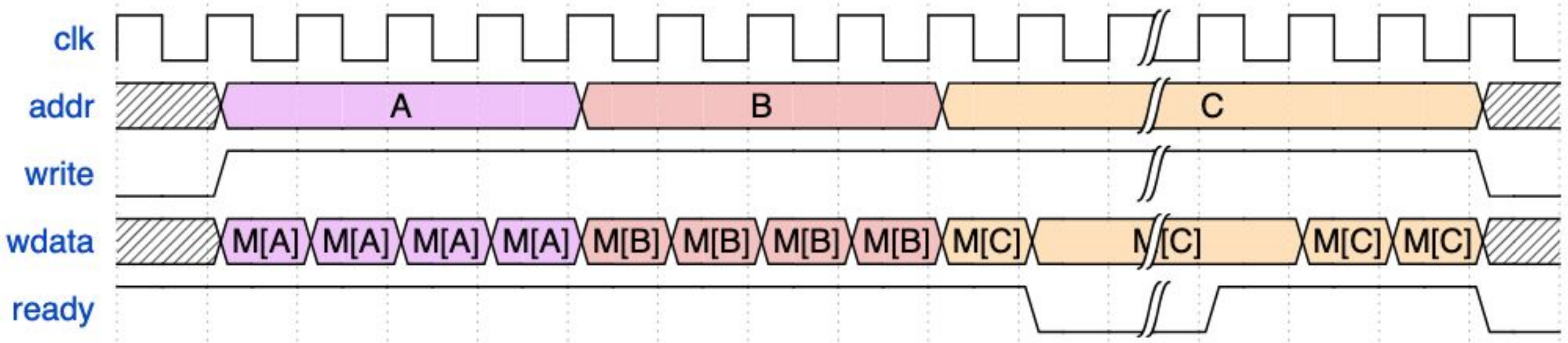


Competition Memory / Burst Memory - Writes

Single Write



Write Single Queue Full



Alternate Verification Tooling

What is Verilator?

- Like VCS, converts your hardware to C++. But it's open sourc
  - Can run on your local machines as a result, and is cross-platform
- Drawbacks: there's a lot.
  - Cycle-level simulation
  - *Only supports dual-state simulation*
  - Also pickier about the quality of your HDL (stricter linting)
  - Incomplete SV support compared to VCS
  - Harder to support threaded models
  - Can't access internal signals by default
  - Traces are way bigger compared to VCS (15GB for all of Coremark with struct extraction)
- Why bother using it at all?
  - ***Speed*** **- "all of CoreMark within seconds" type speed.**

Verilator Testbenches

- Overall simulation logic is no longer managed by an SV initial block
- Need to write C++ logic to interface with the DUT, manage CLK, etc.
  - Can still write performance monitoring in SV
- State management in C++ -> powerful software models in a familiar language
  - Lack of bitslices can sometimes make memory access challenging
- Many other features, so definitely look at the documentation on their website!
- We're nice, so we made you some tooling
  - Does the same thing as top_tb for the most part, just faster
  - Be careful when dumping traces
  - Not currently supported on EWS (both good and bad?)
  - Further instructions will all be in a README

Test Suite

Test Suite

- See your processor run on many real applications!
  - Overview of applications on Campuswire
- Help discover edge cases
  - Found some on staff CPUs!
- Use __divsi3() function to emulate divisions

Thanks for coming!
Questions?