

COMPSCI 535 Project Report
Modular Reversible Architecture (MRA)
by Ryan Fletcher and Riya Sharma

Architecture:

MRA is a clean-slate general-purpose minimally-speculative architecture, taking instruction-set inspiration from established architectures, with an emphasis on user control/modularity. Its distinguishing features are:

- The ISA supports two different instruction word sizes, configurable by the coder. This allows flexible management of instruction memory within unified memory, and allows the usage of different assembly coding strategies (mainly in that longer instruction words allow more immediates as arguments, while shorter instruction words are more uniform w.r.t. assembly format (which may be preferable if few immediates are required) and more space-efficient (which may be preferable for especially large codebases). In that respect, this ISA is intended as a small-scale proof-of-concept for devices that may want to specialize to use words of variable sizes beyond what would be reasonable for us to offer by ourselves. The limitations provided by using the sizes we chose make this small-scale PoC more illustrative.
- The ISA is generally modular, incorporating as many optional arguments and other kinds of configurations as we had time to implement. A more thorough implementation of the same style could introduce a great many checks using the two predicate registers, dramatically increasing execution modularity without increasing the size of the instruction set.
- Multiple consecutive writes to the data registers can be reversed with a single instruction (UNDO) that could, if implemented in hardware, execute in a single clock cycle. This is a practice that is not used in modern architectures because of the speculative nature of common pipeline speed-up strategies. However, it can be useful in several ways, and may be ideal for an emerging field of digital computing: simulated quantum computation, which turns out to potentially be more efficiently than actual quantum computing. Quantum computing requires all operations to be reversible, so supporting that behavior at the hardware level may be more efficient than doing so at the software level.

The instruction word sizes are 32 bits and 64 bits, controlled by the user with a special instruction at the beginning of the assembly code (see “Other Instructions”). The data memory word size is 32 bits. The only supported data type in the implementation is Integer, though the ISA defines Floating Point instructions. The Integer operations are Add, Subtract, Multiply, Divide, Modulo, AND, OR, XOR, NOT, compare, and all types of bitshifts (left/right logical/arithmetic, left rotating).

There are 22 registers total in the two main banks. The first bank (the data bank) has 16 registers, indexable by the user and each 32 bits long. The first (R0) is designated as the return register. The rest are general-purpose (R1-R15). There are 6 registers in the second bank, which the user cannot access directly (the ISA defines instructions to manipulate some of these). Its registers are: a 1-bit 0-constant register for logic reference (not simulated directly, but included for hardware faithfulness), a 25-bit constant pointer to the starting address of data memory in unified storage (see below), a 25-bit program counter, a 16-bit condition codes register, and two 32-bit predicate registers which are defined to enable/disable certain functionalities of the ISA (see “Other Instructions”).

The execution model is a basic 5-stage single-pipeline architecture with one instruction per word, Harvard memory architecture, with consecutive instructions stored consecutively. The program counter tracks the current execution location in memory. The Fetch stage accesses the instruction cache nearest the CPU (or the nearest RAM if there is no instruction cache) at each clock cycle and passes the next instruction up the pipeline when it’s finished fetching. The Decode stage takes the instructions and associates various “auxiliary terms” with them, depending on their structure. This is mostly source and destination registers (which are tracked as available or not through scoreboarding). Other auxiliary terms include whether or not the instruction is finished executing (in whichever stage it executes) and which register files the instructions work on. The execute stage executes most of the instructions, including all the ALU instructions. It also automatically checks certain predications based on the condition code and predicate registers. These can be used for increased control of which instructions actually execute by using implementations of the GET/SET {P,C} instructions (not implemented by us due to lack of time). The Memory Access stage executes load and store instructions. The Memory Writeback stage writes results of executions to destination registers, and it executes branch instructions and UNDO instructions. It also records no-ops and stalls that pass through it.

The ISA supports operation with as little as a single unified-memory RAM device, or with several caches of both types, or anything in-between. Memory modules automatically link themselves according to the order and partition in which they’re created. The nearest cache of each type will always be accessible to the “CPU” and the farthest cache of each type will always pass requests to the nearest unified memory module. Memory modules can be configured with any non-negative clock-cycle delay value, which is used for both kinds of accesses. Line size should always be 8, but technically the memory can be configured differently, to detriment of operational stability. A memory module can be configured to have any positive number of lines. Only the nearest instructions cache should ever be set as using 64-bit words. Instruction fetches in 64-bit word mode are translated automatically when passed to 32-bit word mode modules. RAM modules are set to a write-through allocate policy by default, and cache modules are set to a write-back policy by default. These policies can be manually changed at any time, with cache flushes occurring as necessary.

The virtual address range is 0 through $2^{25}-1$, indexed every 4 bytes. That means the virtual address space will have a maximum of ~134MB of combined drive memory. This

The assembler supports base-10 immediate signed integers written as “#value”, base-16 immediate unsigned integers written as “xvalue”, and unsigned binary numbers written as “value”. For use after implementing defined FP operations, decimal immediate signed floating-points written as “#whole.fractional” should also be allowed. The assembler also evaluates multi-term addition/subtraction expressions that both involve constants (formatted “term1±term2...” where each term includes its appropriate prefix). This enables easy relative-indexing with instruction location labels (see below).

Carets around arguments are not written in the assembly. i.e. a correct example instruction is “LOAD #65000, R4” but not “LOAD <#65,000>, <R4>”. Optional arguments are denoted by curly brackets around the argument(s) (i.e. {<arg>}). Arbitrary bit arrangements in instruction formats are denoted in set notation (i.e. {b_x,b_y}).

Any line can be commented by placing “@comment” at the end of the line (any amount of whitespace before and after the ‘@’ and comment is allowed). An otherwise-blank line can be labeled with “=label” (with ignored whitespace before the ‘=’, though whitespace after the ‘=’ will not ignored), and “%label” can be used in place of an immediate instruction memory address anywhere in the assembly. (Note: Future implementations should automatically remove the no-ops from labels and other instruction-less lines and adjust line immediates accordingly.)

e.g.:

1 JUMP #400	1 JUMP %foo	
2 ...	2 ...	
... 	
400	400 =foo	
401 LOAD #65000, R4	401 LOAD #65000, R4	
402 COPY #400, R5	402 COPY %foo, R5	@ %foo+1 skips the
403 JUMP #400	403 JUMP R5	@ no-op in line 400

Special Feature:

UNDO instruction and ‘step-back’ in debugger to take full advantage of reversible instructions.

Some computations share many (or all) of their arguments, so it’s convenient to be able to keep those arguments readily available even after one or more of those computations have been performed. However, with limited data register space, limited call stack space, and the inefficiency of frequent calls, some assembly-level computation simply must overwrite arguments in registers in order to store intermediate or final results. Using the UNDO instruction, those final results can be kept, and the arguments can be restored in-place. UNDO only reverses the state of the data registers, so it leaves things like the PC and CC registers alone, and does not affect memory. As stated above, this allows storage of intermediate results as well.

The previous state frames are stored in the reversal stack, which is structured almost identically to the call stack. However, it’s a circular stack (i.e. too-old data gets dropped out the bottom), R0 is the first data register recorded (rather than being ignored), and each frame starts with a 16-bit mask to indicate which data registers were actually overwritten in that frame.

The simulator executes UNDO slightly unfaithfully to hardware (for quicker implementation). It overwrites all 16 indexable registers from the appropriate index of the reversal stack (which is correct), but then it must restore the changes from “skipped” writes. The simulator does this by iterating backwards through each of the skipped frames, checking to see all the registers that actually changed in that frame, and setting the appropriate register with the appropriate value if it did. That would take linear time up to the size of the reversal stack if implemented in hardware. However, identical behavior could be produced in-between clock cycles by keeping track of the reversal stack index of the most recent actual change to each indexable register (using 16 special registers in the file), and then instead of iterating through the skipped reversals, simply checking the most recent change to each data register to see if it happened during one of the skipped frames and setting the appropriate data register if so. That requires constant time of one line write to the entire data register file and thirty-two (parallelizable) index comparisons (plus another masked line write) for the most recent changes, so UNDO can be implemented in hardware as a non-stalling instruction.

The step-back function in the debugger causes the writeback pipeline stage to act as though it were just cycled while holding an UNDO instruction with the corresponding arguments.

When the UNDO instruction is encountered, the PC register will still be incremented after executing the UNDO.

Simulator:

The simulator's controls will be defined below in the **Usage Manual** section.

The simulator is created in a single class, using java swing for the GUI. It's instantiated by `Main.main()`, and is passed register files and a pipeline to use and manage. The simulator creates and manages memory modules within itself. Once the GUI is booted up and all memory module preset requirements are met, the simulator waits for user input.

Upon the creation of the first RAM memory module, the simulator will automatically read raw bytes from `files/bin/{1,3}.txt` into that module. The user can then create whatever other memory modules they wish to use.

Once the memory is set up, the user can inspect the starting state of the system however they please. The simulator tracks and displays all registers, all stacks, the pipeline, and all memory states. The user can use the memory manipulation panel to manually alter memory modules if they wish.

The simulator's main function is, obviously, running instructions. The main way to do this is for the user to use the "Cycle" button. The user can enter a number into the button's associated text field in order to step that many cycles into execution, or they can empty the text field and press the button, in which case the program will run until an error or HALT is encountered. (The simulator prints the status of the indexable registers to the console every 100,000 cycles for debug convenience.) At every cycle, the simulator first pre-cycles the pipeline (equivalent to the steps taken by the pipeline in the first half of the clock cycle swing), then cycles the memory, then standard-cycles the pipeline (equivalent to the second half of the clock cycle swing). This pattern ensures proper synchronization and cycle counting of pipeline interactions with memory. The clock cycle controller has special checks to ensure data memory is not fetched and partially executed as instructions at the end of a program unless intended.

The user can also reverse data register writes with the "UNDO" button and its associated text fields. They can be used to directly and immediately simulate an UNDO instruction passing through the write-back stage of the pipeline.

The results of the program are not output by the simulator directly, but can be read in memory through the memory display. The contents of memory modules can be saved to files in raw form in order to save computation results. Program metrics (stalls, no-ops, clock cycles taken) are displayed at all times at the top left of the simulator GUI.

Software Engineering:

The project was developed in sprints according to the project milestone outlined in our proposal and the presentation deadlines for the course. We typically divvied up subtasks between us and tracked assignments using a taskboard on Jira. We used an agile methodology and created logs of tasks for each sprint. We met at least once each week, and update our progress and roadblocks throughout sprints to ensure efficient communication throughout the project. We did pair programming for some portions of the tasks, and delegated the rest as presented in the **Contributions** section below.

We used GitHub to update and revise code. We would each work in one branch (making new ones as appropriate) during each sprint, and then we would update our respective branches before merging them at the end of the sprint/when the task was finished. (Occasionally we'd need to manually add code some other way when/if there was a glitch or hiccup with our local IDEs.) Documentation (besides the **Usage Manual** below) consisted of partial docstrings where appropriate.

https://github.com/NephilimGreen/Final_535_Project.git

The software was implemented in Java. The infrastructure is set up such that it is comparatively easy to implement new instructions with highly configurable formats and behaviors. (i.e. Defining new instruction mnemonics/categories, parsing assembly, decoding binaries, and obviously executing instructions in the pipeline are all distinct steps that can be mostly copy/pasted from instruction to instruction, or can be heavily modified without breaking other functionality.) This allowed for better prototyping and experimentation.

Debugging was performed in an informal unit-testing manner. As complex as each component was, each component's functionality was typically straightforward and somewhat narrow, so a session of operational debugging was used to make sure each component was ready for presentation in the appropriate order. This operational debugging would consist of manually using the simulator (the first GUI version of which was constructed *very* early on) to reach each branch of the new code using a few different versions of data. This was obviously tedious, but it simulated actual operational stresses on the system and simultaneously helped us make live improvements to the simulator to make it more user-friendly and flexible.

This is discussed more in the **What We Learned** section, but the specific order of the presentation deadlines for the course was not in line with our particular intuitive construction of a project like this, so we did a few things backwards, such as making the memory system "API" before figuring out how the pipeline would actually make memory calls. As a result, we ended up making partial overhauls of a couple tasks after their initial completions in order to keep them fully functional and flexible enough for future integrations (such as the aforementioned memory system and the clock cycling procedure).

Contributions:

- Revisions to project proposal
 - Contributors:
 - Ryan
 - ISA Spec
 - Riya
 - Other portions
- Implementation of memory system
 - Related files:
 - memory/*
 - Contributors:
 - Ryan
 - Core MemoryModule class
 - Supplementary classes
 - Revisions and integration of core RegisterFile class
 - Riya
 - Framework of RegisterFile class
- Simulator
 - Related files:
 - main/Simulator.java
 - Contributors:
 - Ryan
- Pipeline, cache integration
 - Related files:
 - pipeline/*
 - Contributors:
 - Ryan
- Assembler
 - Related files:
 - main/Assembler.java
 - Contributors:
 - Ryan
- Add minimal instructions to assembler
 - Related files:
 - main/Assembler.java
 - instructions/*
 - Contributors:
 - Ryan
- ISA completed
 - Related files:

- main/Assembler.java
 - instructions/*
- Contributors:
 - Ryan
 - Finished code for all instructions
 - Riya
 - Starter code for some typecodes
- ISA integrated w/simulator, debug UI
 - Related files:
 - main/Assembler.java
 - instructions/*
 - Contributors:
 - Ryan
- Full ISA, simulator, debug/UI, with timing and benchmarks
 - Related files:
 - files/holding...
 - .../MatrixBenchmarks/*
 - .../ExchangeSortBenchmarks/*
 - .../UndoBenchmarks/*
 - Contributors:
 - Ryan
 - Matrix benchmark creation
 - UNDO showcase benchmark creation
 - All benchmark testing
 - Riya
 - Exchange sort benchmark creation and testing
- Final report
 - Contributors:
 - Ryan:
 - ISA Spec updates
 - Restructuring after draft feedback
 - Architecture section
 - Benchmarks section
 - Simulator section
 - Usage Manual section
 - Software Engineering section
 - What We Learned section
 - Riya:
 - Software Engineering section
 - What We Learned section

Benchmarks:

Matrix Multiply:

The algorithm is mostly unoptimized, except it starts by making a transpose of the second matrix so the accesses of the second matrix are more compatible with the row-major format of the memory storage of the matrices.

30×30 * 30×30:

- | | |
|--|---|
| <ul style="list-style-type: none">● No Pipeline, 100-cycle RAM:<ul style="list-style-type: none">○ 17,928,697 stalls○ 28,837 no-ops○ 48,367,417 clock cycles (~13 minutes) | <ul style="list-style-type: none">● Pipeline, 100-cycle RAM:<ul style="list-style-type: none">○ 47,748,036 stalls○ 28,837 no-ops○ 48,257,604 clock cycles (~13 minutes) |
| <ul style="list-style-type: none">● No Pipeline, 100-cycle RAM, 2-cycle 16-line data cache, 1-cycle 4-line instruction cache:<ul style="list-style-type: none">○ 2,212,033 stalls○ 28,837 no-ops○ 8,750,553 clock cycles | <ul style="list-style-type: none">● Pipeline, 100-cycle RAM, 2-cycle 16-line data cache, 1-cycle 4-line instruction cache:<ul style="list-style-type: none">○ 7,671,773 stalls○ 28,837 no-ops○ 8,181,341 clock cycles |
| <ul style="list-style-type: none">● No Pipeline, 100-cycle RAM, 10-cycle 64-line data cache, 2-cycle 32-line data cache, 1-cycle 8-line instruction cache:<ul style="list-style-type: none">○ 849,863 stalls○ 28,837 no-ops○ 2,841,119 clock cycles | <ul style="list-style-type: none">● Pipeline, 100-cycle RAM, 10-cycle 64-line data cache, 2-cycle 32-line data cache, 1-cycle 8-line instruction cache:<ul style="list-style-type: none">○ 1,439,797 stalls○ 28,837 no-ops○ 1,949,365 clock cycles |

As you can see, the pipelining makes a noticeable but small difference in the cycle count, but caching makes a huge difference. This is because most of the time is spent pulling data from memory.

For context, a 100×100×100 operation with pipelining, three data caches (20 cycles, 1024 lines; 10 cycles, 256 lines; 2 cycles, 128 lines), & a 32-line instruction cache gives the following metrics. This is about the reasonable (realistic) limit of our matrix multiplication capabilities (it takes ~15 minutes to run on a decent desktop with three data caches; it would take hours per run to compare on the weaker configurations).

- 32,417,476 stalls
- 1,020,107 no-ops
- **50,677,994** clock cycles

UNDO Showcase:

This algorithm is a modified matrix multiply that uses the UNDO instruction to more quickly return to a registerfile state with old data, cutting out many of the LOAD instructions. This should, with enough data or the right amount of space, significantly decrease the number of required clock cycles.

Standard Matrix Multiply:

- $10 \times 5 \times 10$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **48,666** clock cycles
- $30 \times 30 \times 30$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **2,841,119** clock cycles

-
- $50 \times 50 \times 50$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **7,787,563** clock cycles
 - $50 \times 50 \times 50$
 - 100-cycle RAM,
 - 20-cycle 1024-line data cache,
 - 10-cycle 256-line data cache,
 - 2-cycle 128-line data cache,
 - 1-cycle 32-line instruction cache:
 - **5,711,740** clock cycles

Half-UNDO Matrix Multiply:

- $10 \times 5 \times 10$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **106,872** clock cycles
- $30 \times 30 \times 30$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **2,493,361** clock cycles

-
- $50 \times 50 \times 50$
 - 100-cycle RAM,
 - 10-cycle 64-line data cache,
 - 2-cycle 32-line data cache,
 - 1-cycle 8-line instruction cache:
 - **9,450,344** clock cycles
 - $50 \times 50 \times 50$
 - 100-cycle RAM,
 - 20-cycle 1024-line data cache,
 - 10-cycle 256-line data cache,
 - 2-cycle 128-line data cache,
 - 1-cycle 32-line instruction cache:
 - **5,577,514** clock cycles

The minimum data and cache size requirements are visible here. When there are few enough LOAD instructions, or enough cache space, for data loading to be dominated by the fetch waiting time, the overhead from using UNDO properly negatively impacts performance. But with enough data, that reverses, and UNDO becomes advantageous.

Exchange Sort:

The algorithm was run on maximally-unsorted data (i.e. pre-sorted in descending order).

200 Items in Array:

- | | |
|--|--|
| ● No Pipeline, 100-cycle RAM: <ul style="list-style-type: none">○ 25,385,187 stalls○ 80,212 no-ops○ 66,291,207 clock cycles (~14 minutes) | ● Pipeline, 100-cycle RAM: <ul style="list-style-type: none">○ 65,410,392 stalls○ 80,212 no-ops○ 66,131,204 clock cycles (~14 minutes) |
| ● No Pipeline, 100-cycle RAM,
2-cycle 8-line data cache,
1-cycle 2-line instruction cache: <ul style="list-style-type: none">○ 4,850,599 stalls○ 80,210 no-ops○ 22,755,633 clock cycles | ● Pipeline, 100-cycle RAM,
2-cycle 8-line data cache,
1-cycle 2-line instruction cache: <ul style="list-style-type: none">○ 22,034,823 stalls○ 80,212 no-ops○ 20,087,707 clock cycles |
| ● No Pipeline, 100-cycle RAM,
10-cycle 16-line data cache,
2-cycle 8-line data cache,
1-cycle 4-line instruction cache: <ul style="list-style-type: none">○ 9,560,830 stalls○ 80,210 no-ops○ 10,281,640 clock cycles | ● Pipeline, 100-cycle RAM,
10-cycle 16-line data cache,
2-cycle 8-line data cache,
1-cycle 4-line instruction cache: <ul style="list-style-type: none">○ 1,274,245 stalls○ 80,212 no-ops○ 3,744,257 clock cycles |

As you can see, the pipelining makes a noticeable but small difference in the cycle count, but caching makes a huge difference. This is because most of the time is spent pulling data from memory. However, once the data caches begin to approach the size of the data, significantly less time is spent fetching from RAM, so the pipelining provides a larger advantage.

For context, sorting a 1000-item maximally-unsorted list with pipelining, two data caches (10 cycles, 64 lines; 2 cycles, 32 lines), & an 8-line instruction cache gives the following metrics. This is about the reasonable limit of our matrix multiplication capabilities (it takes ~17 minutes to run on a decent desktop with three data caches; it would take hours per run to compare on the weaker configurations).

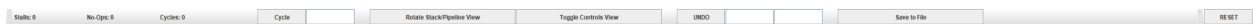
- 34,866,175 stalls
- 2,001,010 no-ops
- **52,870,185** clock cycles

Usage Manual:

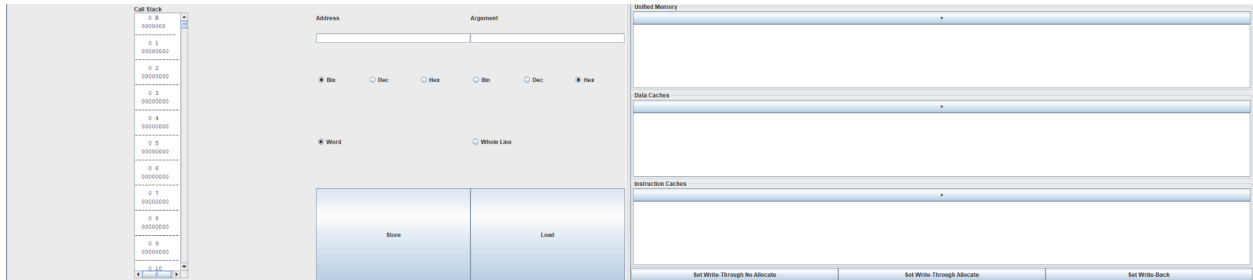
The assembler and the simulator are run by two different files. Or you can set `src/main/Main.RUN_ASSEMBLER` to true to automatically assemble your code into a new pair of binaries before running the simulator.

To assemble code into binary, first create two text files in `src/files/assembly`: “`instructionAssembly.txt`” and “`dataAssembly.txt`”. Then run `src/main/Assembler.main()`. That will create two files in `src/files/bin`: “`1.txt`” and “`3.txt`”.

Then to run the simulator, first change `src/main/Main.STARTING_MEMORIES` to be one of the memory presets found in `src/main/GLOBALS.java`. If you want full control of the memory structure, modify and/or use one of the presets that only has a RAM unit or has no units at all (shipped as the default). Then run `src/main/Main.main()`. This will launch the simulator. You will see a window with a top toolbar,



a top body section,



a middle body section,

R0		R1		R2		R3		R4		R5		R6		R7
0		0		0		0		0		0		0		0
<hr/>														
R8		R9		R10		R11		R12		R13		R14		R15
0		0		0		0		0		0		0		0
<hr/>														
CO		CM		PC		CC		PRED 1		PRED 2				
0		80		8		0		0		0				

and a lower body section.

DIRTY	VALID	LINE ADDRESS	000	001	010	011	100	101	110	111
0	1	0	-1072691456	80	0	0	0	0	0	0
0	1	1	-469762049	1	-469762049	-1	33554432	1	-1845493760	18
0	1	10	-469762049	-1	-469762049	-1	-1845493760	3	-469762049	-1
0	1	11	-1845493760	4	-469762049	-1	1073741824	9040	1073741824	9312
0	1	100	0	87	0	104	1879048192	135	704643072	44
0	1	101	134217728	268435461	134217728	234881030	-469762049	-1	1090520064	320
0	1	110	1879048192	65	704643072	26	1090519808	304	1879048192	49
0	1	111	704643072	22	-469762049	-1	-469762049	-1	-469762049	-1
0	1	1000	-469762049	-1	-704643072	0	-704643072	0	-704643072	0
0	1	1001	-704643072	0	-704643072	0	-704643072	0	-704643072	0
0	1	1010	200	200	199	198	197	196	195	194
0	1	1011	193	192	191	190	189	188	187	186
0	1	1100	185	184	183	182	181	180	179	178
0	1	1101	177	176	175	174	173	172	171	170
0	1	1110	169	168	167	166	165	164	163	162
0	1	1111	161	160	159	158	157	156	155	154
0	1	10000	153	152	151	150	149	148	147	146
0	1	10001	145	144	143	142	141	140	139	138
0	1	10010	137	136	135	134	133	132	131	130
0	1	10011	129	128	127	126	125	124	123	122
0	1	10100	121	120	119	118	117	116	115	114
0	1	10101	113	112	111	110	109	108	107	106
0	1	10110	105	104	103	102	101	100	99	98
0	1	10111	97	96	95	94	93	92	91	90
0	1	11000	89	88	87	86	85	84	83	82
0	1	11001	81	80	79	78	77	76	75	74
0	1	11010	73	72	71	70	69	68	67	66
0	1	11011	65	64	63	62	61	60	59	58
0	1	11100	57	56	55	54	53	52	51	50

Toolbar:

Stalls: 0	No-Ops: 0	Cycles: 0
-----------	-----------	-----------

- “Stalls: <num>”
 - The number of stalls that have passed through the writeback pipeline stage.
- “No-Ops: <num>”
 - The number of no-ops that have passed through the writeback stage.
- “Cycles: <num>”
 - The number of clock cycles that have occurred.

Cycle	
-------	--

- - Click the cycle button with nothing in the text field to cycle until a HALT is encountered.
 - Enter a positive decimal integer in the text field before clicking the button to cycle that many times or until a HALT is encountered.
 - If the pipeline is disabled, the number in the text field is instead how many instructions to execute (incl. stalls from multi-cycle instructions). The clock cycles will still be counted correctly.

Rotate Stack/Pipeline View	Toggle Controls View
----------------------------	----------------------

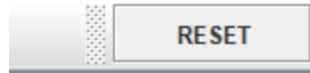
- [Rotate Stack/Pipeline View]
 - Click this button to switch between viewing the Call Stack, the Reversal Stack, or the Pipeline in the top body section.
- [Toggle Controls View]
 - Click this button to switch between viewing the memory manipulation controls (default) or the memory creation controls in the top body section.

UNDO		
------	--	--

- - Enter <quantity> into the first text field and (optionally) <skip> into the second text field and then click the UNDO button to execute an UNDO instruction instantly.

Save to File

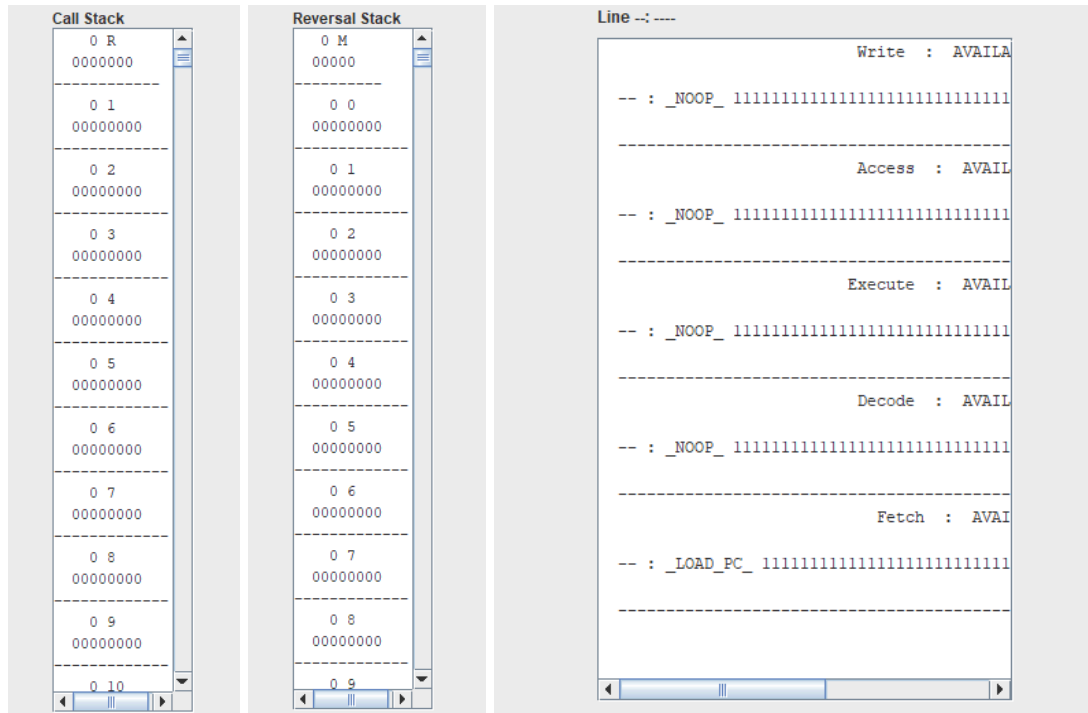
- - Click this button to save the contents of the **currently-selected** memory device to a file in raw form (can be loaded into a memory device in later simulations or can be read with anything that can process raw bytes).



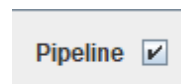
- Completely reset the simulator. Will forget the entire temporary state and restart the window.

Top body section:

- Left view



- The call stack, data register write reversal stack, or pipeline. (Switched with [Rotate Stack/Pipeline View] in the toolbar.
- Each register is labeled with the index of its frame followed by its index/name *in* the frame.
- Each pipeline stage is labeled with its name, whether it's blocked or available, and info about its held instruction. Select the Argument Bin or Dec radio buttons in the Memory Manipulation view to see the source line number and instruction header in place of the first six bits of each instruction.



- Check/Uncheck **Pipeline** to control whether or not the pipeline is activated at any given time.

- Middle view
 - Switched with [Toggle Controls View] in the toolbar.
 - Memory manipulation view

Address	Argument
<input type="text"/>	<input type="text"/>
<input checked="" type="radio"/> Bin <input type="radio"/> Dec <input type="radio"/> Hex	<input type="radio"/> Bin <input checked="" type="radio"/> Dec <input type="radio"/> Hex
<input checked="" type="radio"/> Word <input type="radio"/> Whole Line	
Store	Load

Address

- Populate the text field with the memory address for [Store]/[Load].

☒ Bin
 ☐ Dec
 ☐ Hex

- Select the numerical base for addresses to be displayed in all memory views.

Argument

- Populate the text field with the source register or immediate value for [Store] or with the destination register for [Load].

☒ Bin
 ☐ Dec
 ☐ Hex

- Select the base for all values to displayed in all memory views.

☒ Word
 ☐ Whole Line

- Select value size for [Store]/[Load].

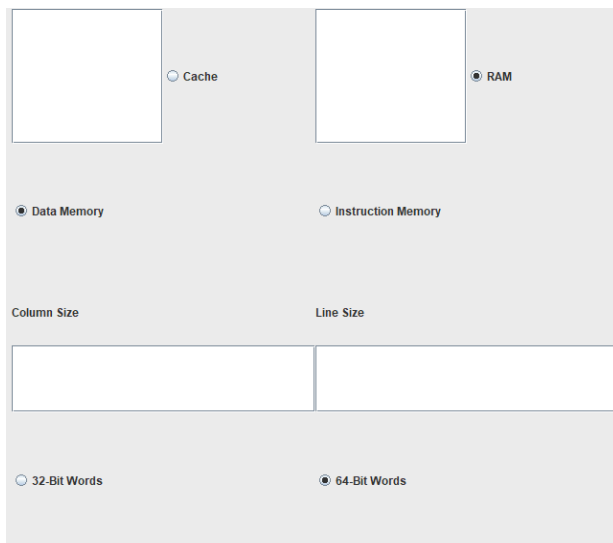
Store

- Click this button to send a store request to the **currently-selected** memory device according to the arguments in Address and Argument text fields.

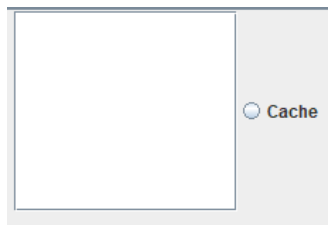
Load

- Click this button to send a load request from the CPU to the **currently-selected** memory device according to the arguments in Address and Argument text fields.

○ Memory creation view:

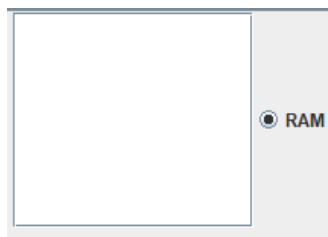


The form is divided into two main sections. The top section contains two columns. The left column has a radio button labeled 'Cache' and a radio button labeled 'Data Memory'. The right column has a radio button labeled 'RAM' and a radio button labeled 'Instruction Memory'. Below these are two text input fields labeled 'Column Size' and 'Line Size'. The bottom section contains two radio buttons: '32-Bit Words' and '64-Bit Words'.



The form consists of a large text input field on the left and a radio button labeled 'Cache' on the right.

- Select this radio button and populate the text field with the access delay (in cycles) of the cache device you will create.



The form consists of a large text input field on the left and a radio button labeled 'RAM' on the right.

- Select this radio button and populate the text field with the access delay (in cycles) of the RAM device you will create.

☒ Data Memory
 ☐ Instruction Memory

- Select the kind of memory you will create.

Column Size

- Populate the text field with the number of lines in the memory device you will create.

Line Size

- Populate the text field with the number of words per line in the memory device you will create.

☒ 32-Bit Words
 ☐ 64-Bit Words

- Select the word size of the memory device you will create.
 - Do not select 64 except for Instruction-type Caches!

- Right view




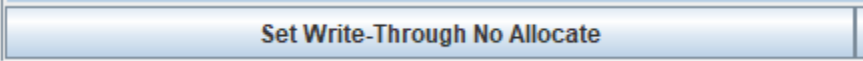
Unified Memory				
ID: 22	Mode: 32 - THROUGH_ALLOCATE	Lines: 1024	Line Size: 8	AVAILABLE

Data Caches				
ID: 23	Mode: 32 - BACK	Lines: 16	Line Size: 8	AVAILABLE
ID: 24	Mode: 32 - BACK	Lines: 8	Line Size: 8	AVAILABLE

Instruction Caches				
ID: 25	Mode: 64 - BACK	Lines: 4	Line Size: 8	AVAILABLE

Set Write-Through No Allocate	Set Write-Through Allocate	Set Write-Back
-------------------------------	----------------------------	----------------

- Click on any device in any list to see its contents in the bottom body section of the window.
- Unified memory
 - All RAM devices in the architecture, listed from furthest-from-CPU to closest-to-CPU.

- The topmost RAM device will be treated as the sum total of all memory storage available to the computer. It will automatically load the binary files created by the assembler.
- 
 - Click this button to create a new unified-memory RAM device according to the arguments populated in the Memory Creation view. It will be closer to the CPU than the other RAM devices, and will defer to the next one up the list.
- Data Caches
 - All Data-type Cache devices in the architecture, listed from furthest-from-CPU to closest-to-CPU.
 - The topmost Data-type Cache device will defer to the bottommost **RAM** device.
 - 
 - Click this button to create a new Data-type Cache device according to the arguments populated in the Memory Creation view. It will be closer to the CPU than the others, and will defer to the next one up the list.
- Instruction Caches
 - All Instruction-type Cache devices in the architecture, listed from furthest-from-CPU to closest-to-CPU.
 - The topmost Data-type Cache device will defer to the bottommost **RAM** device.
 - 
 - Click this button to create a new Instruction-type Cache device according to the arguments populated in the Memory Creation view. It will be closer to the CPU than the others, and will defer to the next one up the list.
- 
 - Click this button to change the write policy of the **currently-selected** memory device to Write-Through No Allocate. This may cause a cache flush which will need to be cycled through.

○

Set Write-Through Allocate

- Click this button to change the write policy of the **currently-selected** memory device to Write-Through Allocate. This may cause a cache flush which will need to be cycled through.

○

Set Write-Back

- Click this button to change the write policy of the **currently-selected** memory device to Write-Back. This may cause a cache flush which will need to be cycled through.

Middle body section:

R0		R1		R2		R3		R4		R5		R6		R7
0		0		0		0		0		0		0		0
<hr/>														
R8		R9		R10		R11		R12		R13		R14		R15
0		0		0		0		0		0		0		0
CO		CM		PC		CC		PRED 1		PRED 2				
0		80		8		0		0		0				

- A scrollable text display of the data registers and internal registers, labeled with their names. One star on each side of a register's name means it has two writes pending. Two stars on each side means it has two writes pending.

Bottom body section:

DIRTY	VALID	LINE ADDRESS	000	001	010	011	100	101	110	111
0	1	0	-1072691456	80	0	0	0	0	0	0
0	1	1	-469762049	-1	-469762049	-1	33554432	1	-1845493760	18
0	1	10	-469762049	-1	-469762049	-1	-1845493760	3	-469762049	-1
0	1	11	-1845493760	4	-469762049	-1	1073741824	9040	1073741824	9312
0	1	100	0	87	0	104	1879048192	135	704643072	44
0	1	101	134217728	268435461	134217728	234881030	-469762049	-1	1090520064	320
0	1	110	1879048192	65	704643072	26	1090519808	304	1879048192	49
0	1	111	704643072	22	-469762049	-1	-469762049	-1	-469762049	-1
0	1	1000	-469762049	-1	-704643072	0	-704643072	0	-704643072	0
0	1	1001	-704643072	0	-704643072	0	-704643072	0	-704643072	0
0	1	1010	200	200	199	198	197	196	195	194
0	1	1011	193	192	191	190	189	188	187	186
0	1	1100	185	184	183	182	181	180	179	178
0	1	1101	177	176	175	174	173	172	171	170
0	1	1110	169	168	167	166	165	164	163	162
0	1	1111	161	160	159	158	157	156	155	154
0	1	10000	153	152	151	150	149	148	147	146
0	1	10001	145	144	143	142	141	140	139	138
0	1	10010	137	136	135	134	133	132	131	130
0	1	10011	129	128	127	126	125	124	123	122
0	1	10100	121	120	119	118	117	116	115	114
0	1	10101	113	112	111	110	109	108	107	106
0	1	10110	105	104	103	102	101	100	99	98
0	1	10111	97	96	95	94	93	92	91	90
0	1	11000	89	88	87	86	85	84	83	82
0	1	11001	81	80	79	78	77	76	75	74
0	1	11010	73	72	71	70	69	68	67	66
0	1	11011	65	64	63	62	61	60	59	58
0	1	11100	57	56	55	54	53	52	51	50

- A scrollable grid text display of the currently-selected memory device's contents.
 - Includes dirty and valid bits, along with split addresses (left column for first $n - \log_2(\text{lineSize})$ bits, top row for last $\log_2(\text{lineSize})$ bits).

ISA Specification:

Instructions highlighted in red are not implemented.

Load/Store Operations (typecode 000)

- **LOAD** <src adr>, <dest reg> Load Single Word
32 : [0 0 0] [0 0 0] [ignore×18] [<src adr>×4 <dest reg>×4]
64 : [0 0 0] [0 0 0] [{0,1}] [ignore×28] [<src adr>×25 <dest reg>×4]
 - <src adr>:
 - Register w/data memory address (flag 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag 1)
Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.
 - <dest reg>:
 - RR-R15

Loads a 4-byte word from data memory into the destination register.

Examples: (all behaviorally equivalent)

LOAD #28330, R3

LOAD 110111010101010, R3

LOAD 0000000001101110101010, R3

LOAD 111010110000000001101110101010, R3

LOAD #721448618, R3

COPY #28330, R3 followed by LOAD R3, R3

- **LOADL** <src adr>, <dest reg start> Load Line
32 : [0 0 0] [0 0 1] [ignore×18] [<src adr>×4 <dest reg>×4]
64 : [0 0 0] [0 0 1] [{0,1}] [ignore×28] [<src adr>×25 <dest reg>×4]
 - <src adr>:
 - Register w/data memory target address (flag 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag 1)
Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.
 - <dest reg start>:
 - RR-R15

Loads line of consecutive 4-byte words from data memory containing the target address into the eight consecutive registers starting at the destination register. Every line starts at an address that is a multiple of 8 (and data memory starts with the first word of its first line), so if target address is not a multiple of 8, it will not be in the first written register. Additionally, if <dest reg start> is after R8, the extra words at the end of the line will not be written anywhere. (Note: This instruction takes the same amount of time to execute as LOAD because cache always loads full 8-word lines from data memory.)

- STR <src>, <dest adr> Store Single Value
 32 : [0 0 0] [0 1 0] [ignore×18] [<src>×4 <dest adr>×4]
 64 : [0 0 0] [0 1 0] [a1 b0] [ignore×20] [<src>×32 <dest adr>×4]
 [0 0 0] [0 1 0] [a0 b{0,1}] [ignore×27] [<src>×4 <dest adr>×25]
 - <src>:
 - Register w/value to be stored (flag_a 0)
 - RR-R15
 - Immediate (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 - <dest adr>:
 - Register w/data memory address (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.

Copies the value from a single register, or writes any immediate value, into data memory.

- STRL <src reg start>, <dest adr> Store Line
 32 : [0 0 0] [0 1 1] [ignore×18] [<src reg start>×4 <dest adr>×4]
 64 : [0 0 0] [0 1 1] [{0,1}] [ignore×28] [<src reg start>×4 <dest adr>×25]
 - <src reg start>:
 - RR-R15
 - <dest adr start>:
 - Register w/data memory address (flag 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag 1)

Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.

Copies the values from the 8 consecutive registers starting at the given source into the 8 consecutive words in the line of data memory containing the target address. Every line starts at an address that is a multiple of 8 (and data memory starts with the first word of its first line), so if target address is not a multiple of 8, it will not be in the first written word. Additionally, if <src reg start> is after R8, the missing words at the end of the line will be replaced with 0s, and **will still overwrite data**. (Note: This instruction takes the same amount of time to execute as STR because cache always writes full 8-word lines into data memory.)

Control Instructions/Modes (typecode 001)

- BR0 <dest adrs> Branch if Zero
32/64 : [0 0 1] [0 0 0] [{0,1}] [ignore×0/32] [<dest adrs>×25]
 - <dest adrs>:
 - Register w/instruction memory address (flag 0)
 - RR-R15
 - Immediate integer (flag 1)
Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.

Sets PC to given instruction address if 0-comparison flag bit in CC register is 1.

● BR0R <sign><shift> Branch if Zero (PC Relative)

- 32/64 : [0 0 1] [0 0 1] [a{0,1} b<sign>] [ignore×0/32] [<shift>×24]
- <sign>:
 - +/- in assembly = 0/1 in flag_b
 - <shift>:
 - Register w/shift value (flag_a 0)
 - RR-R15
 - **Non-negative** immediate integer (flag_a 1)
Only rightmost **24** bits are counted. Those **24** bits are treated as an unsigned integer. (24 bits allows traversal of half the memory space in either direction.)

Shifts PC by given amount if 0-comparison flag bit in CC register is 1.

- BRN <dest adrs> Branch if Negative
32/64 : [0 0 1] [0 1 0] [{0,1}] [ignore×0/32] [<dest adrs>×25]
 - <dest adrs>:
 - Register w/instruction memory address (flag 0)
 - RR-R15
 - Immediate integer (flag 1)
Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.

Sets PC to given instruction address if negative-comparison flag bit in CC register is 1.

- **BRNR <sign><shift>** **Branch if Negative (PC Relative)**

32/64 : [0 0 1] [0 1 1] [a{0,1} b<sign>] [ignore×0/32] [<shift>×24]

- <sign>:
 - +/- in assembly = 0/1 in flag_b
- <shift>:
 - Register w/shift value (flag_a 0)
 - RR-R15
 - **Non-negative** immediate integer (flag_a 1)

Only rightmost **24** bits are counted. Those **24** bits are treated as an unsigned integer. (24 bits allows traversal of half the memory space in either direction.)

Shifts PC by given amount if negative-comparison flag bit in CC register is 1.

- **JUMP <dest adrs>** **Jump**

32/64 : [0 0 1] [1 0 0] [{0,1}] [ignore×0/32] [<dest adrs>×25]

- <dest adrs>:
 - Register w/instruction memory address (flag 0)
 - RR-R15
 - Immediate integer (flag 1)

Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.

Sets PC to given instruction address.

- **JUMPR <sign><shift>** **Jump (PC Relative)**

32/64 : [0 0 1] [1 0 1] [a{0,1} b<sign>] [ignore×0/32] [<shift>×24]

- <sign>:
 - +/- in assembly = 0/1 in flag_b
- <shift>:
 - Register w/shift value (flag_a 0)
 - RR-R15
 - **Non-negative** immediate integer (flag_a 1)

Only rightmost **24** bits are counted. Those **24** bits are treated as an unsigned integer. (24 bits allows traversal of half the memory space in either direction.)

Shifts PC by given amount.

- CALL <dest adr>{, <sign><return shift>} Call

32 : [0 0 1] [1 1 0] [b{0,1} c<sign>] [ignore×17] [<dest adr>×4 <return shift>×4]

64 : [0 0 1] [1 1 0] [a{0,1} b{0,1} c<sign>] [ignore×6] [<dest adr>×25 <return shift>×24]

 - <dest adr>:
 - Register w/instruction memory address (flag_a 0)
 - RR-R15
 - Immediate (64-bit mode only) (flag_a 1)

Only rightmost 25 bits are counted. Those 25 bits are treated as an unsigned integer.
 - <sign>:
 - +/- in assembly = 0/1 in flag_c
 - <return shift>:
 - Immediate (64-bit mode only) (flag_b 0)

Only rightmost **24** bits are counted. Those **24** bits are counted as an unsigned integer.

In 32-bit word mode, if <return shift> defaults, it's parsed as an immediate.
 - Register w/shift value (flag_b 1)
 - RR-R15
 - Default 1 (Return shift is relative address of first instruction that will be executed after return. Can use 0 arg to re-execute CALL.)
 - Default flag_b=0, flag_c=0

Pushes the following line to the call stack: PC+(<sign><return shift>) (25 bits), R1 through R15 in order (15×32 bits). Jumps to <dest adr>.
- RETURN Return

32/64 : [0 0 1] [1 1 1] [ignore×26/58]

Pops the call stack. Sets the PC register equal to the first 25 bits of the retrieved state frame, then writes the remaining 15 32-bit words in the frame into R1-R15, in order.

Integer Arithmetic Operations (typecode 010)

- ADD <operand 1>, <operand 2>{, <dest reg>{, <set carry>}} Integer Add
- 32 : [0 1 0][0 0 0][c{0,1}][ignore×9][<operand 1>×4 <operand 2>×4 <dest reg>×4 <set carry>×4]
- 64 : [0 1 0][0 0 0][a0 b0 c{0,1}][ignore×39][<operand 1>×4 <operand 2>×4 <dest reg>×4 <set carry>×4]
 [0 1 0][0 0 0][a1 b0 c{0,1}][ignore×11][<operand 1>×32 <operand 2>×4 <dest reg>×4 <set carry>×4]
 [0 1 0][0 0 0][a0 b1 c{0,1}][ignore×11][<operand 1>×4 <operand 2>×32 <dest reg>×4 <set carry>×4]
- <operand 1>:
 - Register w/integer operand 1 value (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 Treated as signed integer.
 - <operand 2>:
 - Register w/integer operand 2 value (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 Treated as signed integer.
 - <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1
 - <set carry>:
 - Immediate integer (flag_c 0)
 <set carry>×4 in instruction word is set to 0000 if <set carry> arg = 0 or #0, set to 0001 otherwise.
 - Register with integer 0 or non-0 value (flag_c 1)
 - RR-R15
 - Default 0

Performs integer addition between the operand 1 and the operand 2 and stores the result in the destination register. Writes to the carry bit in the condition codes register if <set carry> is non-0.

- SUB <operand 1>, <operand 2>{, <dest reg>, <compare>} Integer Subtract
- 32 : [0 1 0][0 0 1][c{0,1}][ignore×9][<operand 1>×4 <operand 2>×4 <dest reg>×4 <compare>×4]
- 64 : [0 1 0][0 0 1][a0 b0 c{0,1}][ignore×39][<operand 1>×4 <operand 2>×4 <dest reg>×4 <compare>×4]
[0 1 0][0 0 1][a1 b0 c{0,1}][ignore×11][<operand 1>×32 <operand 2>×4 <dest reg>×4 <compare>×4]
[0 1 0][0 0 1][a0 b1 c{0,1}][ignore×11][<operand 1>×4 <operand 2>×32 <dest reg>×4 <compare>×4]
- <operand 1>:
 - Register w/integer operand 1 value (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
Treated as signed integer.
 - <operand 2>:
 - Register w/integer operand 2 value (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
Treated as signed integer.
 - <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1
 - <compare>:
 - Immediate integer (flag_c 0)
<compare>×4 in instruction word is set to 0000 if <compare> arg = 0 or #0, set to 0001 otherwise.
 - Register with integer 0 or non-0 value (flag_c 1)
 - RR-R15
 - Default 0

Subtracts the operand 2 from the the operand 1 and stores the result in the destination integer register. If compare value is non-0, sets the 0, negative, and positive bits in the CC register according to the result. Compare value is taken from <compare> in instruction word as immediate 0 or non-0 if flag_c is 0. If flag_c is 1, compare value is read from register with address <compare>.

- MUL <operand 1>, <operand 2>{, <dest reg>{, <major>}} Integer Multiply

32 : [0 1 0] [0 1 0] [c{0,1}] [ignore×9] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <major>×4]

64 : [0 1 0] [0 1 0] [a0 b0 c{0,1}] [ignore×39] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <major>×4]
 [0 1 0] [0 1 0] [a1 b0 c{0,1}] [ignore×11] [<operand 1>×32 <operand 2>×4 <dest reg>×4 <major>×4]
 [0 1 0] [0 1 0] [a0 b1 c{0,1}] [ignore×11] [<operand 1>×4 <operand 2>×32 <dest reg>×4 <major>×4]

- <operand 1>:

- Register w/integer operand 1 value (flag_a 0)

- RR-R15

- Immediate integer (64-bit mode only) (flag_a 1)

- Only if flag_b=0

Treated as signed integer.

- <operand 2>:

- Register w/integer operand 2 value (flag_b 0)

- RR-R15

- Immediate integer (64-bit mode only) (flag_b 1)

- Only if flag_a=0

Treated as signed integer.

- <dest reg>: (for rightmost 32 bits of result)

- RR-R15

- Default <operand 1> if flag_a=0

- Default <operand 2> if flag_a=1

- <major>: (flag_c 1)

- Destination register for leftmost 32 bits of result.

- RR-R15

- Not <dest reg>

Default flag_c 0

Multiplies the operand 1 by the operand 2 and stores the result in the destination register(s).

- DIV <operand 1>, <operand 2>{, <dest reg>{, <remainder>}}

Integer Divide

32 : [0 1 0] [0 1 1] [c{0,1}] [ignore×9] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <remainder>×4]

64 : [0 1 0] [0 1 1] [a0 b0 c{0,1}] [ignore×39] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <remainder>×4]
 [0 1 0] [0 1 1] [a1 b0 c{0,1}] [ignore×11] [<operand 1>×32 <operand 2>×4 <dest reg>×4 <remainder>×4]
 [0 1 0] [0 1 1] [a0 b1 c{0,1}] [ignore×11] [<operand 1>×4 <operand 2>×32 <dest reg>×4 <remainder>×4]

- <operand 1>:

- Register w/integer operand 1 value (flag_a 0)

- RR-R15

- Immediate integer (64-bit mode only) (flag_a 1)

- Only if flag_b=0

Treated as signed integer.

- <operand 2>:

- Register w/integer operand 2 value (flag_b 0)

- RR-R15

- Immediate integer (64-bit mode only) (flag_b 1)

- Only if flag_a=0

Treated as signed integer.

- <dest reg>:

- RR-R15

- Default <operand 1> if flag_a=0

- Default <operand 2> if flag_a=1

- <remainder>:

(flag_c 1)

- Destination register for remainder of result

- RR-R15

- Not <dest reg>

Default flag_c 0

- MOD <operand 1>, <operand 2>{, <dest reg>{, <divisor>}}
- Modulo
- 32 : [0 1 0] [1 0 0] [c{0,1}] [ignore×9] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <divisor>×4]
- 64 : [0 1 0] [1 0 0] [a0 b0 c{0,1}] [ignore×39] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <divisor>×4]
 [0 1 0] [1 0 0] [a1 b0 c{0,1}] [ignore×11] [<operand 1>×32 <operand 2>×4 <dest reg>×4 <divisor>×4]
 [0 1 0] [1 0 0] [a0 b1 c{0,1}] [ignore×11] [<operand 1>×4 <operand 2>×32 <dest reg>×4 <divisor>×4]
- <operand 1>:
 - Register w/integer operand 1 value (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 Treated as signed integer.
 - <operand 2>:
 - Register w/integer operand 2 value (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 Treated as signed integer.
 - <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1
 - <divisor>: (flag_c 1)
 - Destination register for divisor to get modulus
 - RR-R15
 - Not <dest reg>

Default flag_c 0

Integer divides the operand 1 by the operand 2 and stores the modulus in the destination register(s). If the <divisor> address is given, in it is stored the value by which the operand 1 can be divided such that the remainder is this operation's retrieved modulus.

Integer Logic Operations: (typecode 011)

- AND <x>, <y>{, <dest reg>} Logical bitwise AND
32 : [0 1 1] [0 0 0] [ignore×14] [<x>×4 <y>×4 <dest reg>×4]
64 : [0 1 1] [0 0 0] [a0 b0] [ignore×44] [<x>×4 <y>×4 <dest reg>×4]
[0 1 1] [0 0 0] [a1 b0] [ignore×16] [<x>×32 <y>×4 <dest reg>×4]
[0 1 1] [0 0 0] [a0 b1] [ignore×16] [<x>×4 <y>×32 <dest reg>×4]
 - <x>:
 - Register with x bits (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 - <y>:
 - Register with y bits (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 - <dest reg>:
 - RR-R15
 - Default <x> if flag_a=0
 - Default <y> if flag_a=1

Performs logical AND with x and y and stores the result in the destination register.

- OR <x>, <y>{, <dest reg>} Logical bitwise OR
 - 32 : [0 1 1] [0 0 1] [ignore×14] [<x>×4 <y>×4 <dest reg>×4]
 - 64 : [0 1 1] [0 0 1] [a0 b0] [ignore×44] [<x>×4 <y>×4 <dest reg>×4]
[0 1 1] [0 0 1] [a1 b0] [ignore×16] [<x>×32 <y>×4 <dest reg>×4]
[0 1 1] [0 0 1] [a0 b1] [ignore×16] [<x>×4 <y>×32 <dest reg>×4]
 - <x>:
 - Register with x bits (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 - <y>:
 - Register with y bits (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 - <dest reg>:
 - RR-R15
 - Default <x> if flag_a=0
 - Default <y> if flag_a=1

Performs logical OR with x and y and stores the result in the destination register.

- XOR <x>, <y>{, <dest reg>} Logical bitwise OR
 - 32 : [0 1 1] [0 1 0] [ignore×14] [<x>×4 <y>×4 <dest reg>×4]
 - 64 : [0 1 1] [0 1 0] [a0 b0] [ignore×44] [<x>×4 <y>×4 <dest reg>×4]
[0 1 1] [0 1 0] [a1 b0] [ignore×16] [<x>×32 <y>×4 <dest reg>×4]
[0 1 1] [0 1 0] [a0 b1] [ignore×16] [<x>×4 <y>×32 <dest reg>×4]
 - <x>:
 - Register with x bits (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 - <y>:
 - Register with y bits (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 - <dest reg>:
 - RR-R15
 - Default <x> if flag_a=0
 - Default <y> if flag_a=1

Performs logical XOR with x and y and stores the result in the destination register.

- NOT <bits reg>{, <dest reg>} Logical bitwise NOT

32/64 : [0 1 1] [0 1 1] [ignore×18/50] [<bits reg>×4 <dest reg>×4]

- <bits reg>:
 - RR-R15
- <dest reg>:
 - RR-R15
 - Default <bits reg>

Performs logical NOT on the bits in the given integer register and stores the result in the destination integer register.

- CMP <operand 1>, <operand 2> Compare

32 : [0 1 1] [1 0 0] [ignore×18] [<operand 1>×4 <operand 2>×4]

64 : [0 1 1] [1 0 0] [a0 b0] [ignore×48] [<operand 1>×4 <operand 2>×4]
 [0 1 1] [1 0 0] [a1 b0] [ignore×20] [<operand 1>×32 <operand 2>×4]
 [0 1 1] [1 0 0] [a0 b1] [ignore×20] [<operand 1>×4 <operand 2>×32]

- <operand 1>:
 - Register w/operand 1 value (flag_a 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
- <operand 2>:
 - Register w/operand 2 value (flag_b 0)
 - RR-R15
 - Immediate integer (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

Subtracts the operand 2 from the operand 1 and sets the 0 bit and negative/positive bit in the CC register according to the result. Does not store the result.

Integer ALU Control (typecode 100)

- SLL <src reg>, <amount>{, <dest reg>} Shift Left Logical
32 : [1 0 0] [0 0 0] [ignore×14] [<src reg>×4 <amount>×4 <dest reg>×4]
64 : [1 0 0] [0 0 0] [{0,1}] [ignore×17] [<src reg>×4 <amount>×32 <dest reg>×4]
 - <src reg>:
 - RR-R15
 - <amount>:
 - Register w/integer shift amount (flag 0)
 - Immediate integer (64-bit mode only) (flag 1)
 - Treated as unsigned integer.
 - <dest reg>:
 - RR-R15
 - Default <src reg>

Performs logical left shift of the bits in the source register and stores the result in the destination register.

- SLR <src reg>, <amount>{, <dest reg>} Shift Left Rotating
32 : [1 0 0] [0 0 1] [ignore×14] [<src reg>×4 <amount>×4 <dest reg>×4]
64 : [1 0 0] [0 0 1] [{0,1}] [ignore×17] [<src reg>×4 <amount>×32 <dest reg>×4]
 - <src reg>:
 - RR-R15
 - <amount>:
 - Register w/integer shift amount (flag 0)
 - Immediate integer (64-bit mode only) (flag 1)
 - Treated as unsigned integer.
 - <dest reg>:
 - RR-R15
 - Default <src reg>

Performs rotating left shift of the bits in the source register and stores the result in the destination register.

- SRL <src reg>, <amount>{, <dest reg>} Shift Right Logical
 - 32 : [1 0 0] [0 1 0] [ignore×14] [<src reg>×4 <amount>×4 <dest reg>×4]
 - 64 : [1 0 0] [0 1 0] [{0,1}] [ignore×17] [<src reg>×4 <amount>×32 <dest reg>×4]
 - <src reg>:
 - RR-R15
 - <amount>:
 - Register w/integer shift amount (flag 0)
 - Immediate integer (64-bit mode only) (flag 1)
 - Treated as unsigned integer.
 - <dest reg>:
 - RR-R15
 - Default <src reg>

Performs logical right shift of the bits in the source register and stores the result in the destination register.

- SRA <src reg>, <amount>{, <dest reg>} Shift Right Arithmetic
 - 32 : [1 0 0] [0 1 1] [ignore×14] [<src reg>×4 <amount>×4 <dest reg>×4]
 - 64 : [1 0 0] [0 1 1] [{0,1}] [ignore×17] [<src reg>×4 <amount>×32 <dest reg>×4]
 - <src reg>:
 - RR-R15
 - <amount>:
 - Register w/integer shift amount (flag 0)
 - Immediate integer (64-bit mode only) (flag 1)
 - Treated as unsigned integer.
 - <dest reg>:
 - RR-R15
 - Default <src reg>

Performs arithmetic right shift of the bits in the source register and stores the result in the destination register.

- COPY <src>, <dest reg> Copy into Register
 - 32 : [1 0 0] [1 0 0] [{0,1}] [<src>×21 <dest reg>×4]
 - 64 : [1 0 0] [1 0 0] [{0,1}] [ignore×21] [<src>×32 <dest reg>×4]
 - <src>:
 - Register (flag 0)
 - RR-R15
 - Any immediate (flag 1)
 - In 32-bit word mode, only the rightmost 21 bits are counted
 - <dest reg>:
 - RR-R15

Copies the source into the destination register and overwrites the old value.

- SWAP <src reg>, <dest reg> Swap Registers

32 : [1 0 0] [1 0 1] [ignore×18] [<reg 1>×4 <reg 2>×4]

64 : [1 0 0] [1 0 1] [ignore×50] [<reg 1>×4 <reg 2>×4]

- <src reg>:
 - RR-R15
- <dest reg>:
 - RR-R15
 - Not <src reg>

Swaps the values in the source and destination registers.

Floating point operations: (typecode 101)

- Overflow, underflow, NaN, /0 exceptions

● **ADDF <operand 1>, <operand 2>{, <dest reg>}** **Floating Point Add**

- 32 : [1 0 1] [0 0 0] [ignore×14] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
- 64 : [1 0 1] [0 0 0] [a0 b0] [ignore×44] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
[1 0 1] [0 0 0] [a1 b0] [ignore×16] [<operand 1>×32 <operand 2>×4 <dest reg>×4]
[1 0 1] [0 0 0] [a0 b1] [ignore×16] [<operand 1>×4 <operand 2>×32 <dest reg>×4]
- <operand 1>:
 - Register w/floating point operand 1 value (flag_a 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
 - <operand 2>:
 - Register w/floating point operand 2 value (flag_b 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
 - <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1

Performs floating point addition between the operand 1 and the operand 2 and stores the result in the destination register.

● SUBF <operand 1>, <operand 2>{, <dest reg>, <compare>} Floating Point Subtract

32 : [1 0 1] [0 0 1] [c{0,1}] [ignore×9] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <compare>×4]

64 : [1 0 1] [0 0 1] [a0 b0 c{0,1}] [ignore×39] [<operand 1>×4 <operand 2>×4 <dest reg>×4 <compare>×4]
 [1 0 1] [0 0 1] [a1 b0 c{0,1}] [ignore×11] [<operand 1>×32 <operand 2>×4 <dest reg>×4 <compare>×4]
 [1 0 1] [0 0 1] [a0 b1 c{0,1}] [ignore×11] [<operand 1>×4 <operand 2>×32 <dest reg>×4 <compare>×4]

○ <operand 1>:

- Register w/floating point operand 1 value (flag_a 0)
 - RR-R15
- Immediate floating point (64-bit mode only) (flag_a 1)
 - Only if flag_b=0

○ <operand 2>:

- Register w/floating point operand 2 value (flag_b 0)
 - RR-R15
- Immediate floating point (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

○ <dest reg>:

- RR-R15
- Default <operand 1> if flag_a=0
- Default <operand 2> if flag_a=1

○ <compare>:

- Immediate integer (flag_c 0)

<compare> in instruction word is set to 0000 if <compare> arg = 0 or #0, set to 0001 otherwise.
- Register with integer 0 or non-0 value (flag_c 1)
 - RR-R15
- Default 0

Subtracts the floating point operand 2 from the the floating point operand 1 and stores the result in the destination integer register. If compare value is non-0, sets the 0 bit and negative bit in the CC register according to the result. Compare value is taken from <compare> in instruction word as immediate 0 or non-0 if flag_c is 0. If flag_c is 1, compare value is read from register with address <compare>.

● **MULF <operand 1>, <operand 2>{, <dest reg>} Floating Point Multiply**

32 : [1 0 1] [0 1 0] [ignore×14] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
 64 : [1 0 1] [0 1 0] [a0 b0] [ignore×44] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
 [1 0 1] [0 1 0] [a1 b0] [ignore×16] [<operand 1>×32 <operand 2>×4 <dest reg>×4]
 [1 0 1] [0 1 0] [a0 b1] [ignore×16] [<operand 1>×4 <operand 2>×32 <dest reg>×4]

- <operand 1>:
 - Register w/floating point operand 1 value (flag_a 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
- <operand 2>:
 - Register w/floating point operand 2 value (flag_b 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
- <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1

Multiplies FP operand 1 by FP operand 2 and stores the result in the destination register.

● **DIVF <operand 1>, <operand 2>{, <dest reg>} Floating Point Divide**

32 : [1 0 1] [0 1 1] [ignore×14] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
 64 : [1 0 1] [0 1 1] [a0 b0] [ignore×44] [<operand 1>×4 <operand 2>×4 <dest reg>×4]
 [1 0 1] [0 1 1] [a1 b0] [ignore×16] [<operand 1>×32 <operand 2>×4 <dest reg>×4]
 [1 0 1] [0 1 1] [a0 b1] [ignore×16] [<operand 1>×4 <operand 2>×32 <dest reg>×4]

- <operand 1>:
 - Register w/floating point operand 1 value (flag_a 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
- <operand 2>:
 - Register w/floating point operand 2 value (flag_b 0)
 - RR-R15
 - Immediate floating point (64-bit mode only) (flag_b 1)
 - Only if flag_a=0
- <dest reg>:
 - RR-R15
 - Default <operand 1> if flag_a=0
 - Default <operand 2> if flag_a=1

Divides the operand 1 by the operand 2 and stores the result in the destination register(s).

● **CMPF <operand 1>, <operand 2>** **Floating Point Compare**

32 : [1 0 1] [1 0 0] [ignore×18] [<operand 1>×4 <operand 2>×4]
 64 : [1 0 1] [1 0 0] [a0 b0] [ignore×48] [<operand 1>×4 <operand 2>×4]
 [1 0 1] [1 0 0] [a1 b0] [ignore×20] [<operand 1>×32 <operand 2>×4]
 [1 0 1] [1 0 0] [a0 b1] [ignore×20] [<operand 1>×4 <operand 2>×32]

○ <operand 1>:

- Register w/floating point operand 1 value (flag_a 0)
 - RR-R15
- Immediate floating point (64-bit mode only) (flag_a 1)
 - Only if flag_b=0

○ <operand 2>:

- Register w/floating point operand 2 value (flag_b 0)
 - RR-R15
- Immediate floating point (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

Subtracts the floating point operand 2 from the the floating point operand 1 and sets the 0 bit and negative bit in the CC register according to the result. Does not store the result.

● **STAT <src reg>** **Floating Point Status**

32/64 : [1 0 1] [1 0 1] [ignore×22/54] [<src reg>×4]

○ <src reg>:

- RR-R15

Sets the appropriate bits in the CC register according to the status of the floating point value in the source register.

Other Instructions: (110)

- SET WORD SIZE {32,64}
SET STACK SIZE <size>
SET BUFFER SIZE <size>

↑ Those are, in typical use, the first three lines of any assembly file for our ISA.

The first 8 word of memory are handled as a special case.

- Leftmost bit of the first word is 0 by default/if first line of assembled file is SET WORD SIZE 32
 - This tells the CPU to read instruction memory as 32-bit words.
- Leftmost bit of the first word is 1 if first line of assembled file is SET WORD SIZE 64
 - This tells the CPU to read instruction memory as 64-bit words.
- Next 10 bits in the first word used as indicator bits for which power of 2 the call stack size is (eg. 0001000000 means it's size 2^7). Bits to the right of the leftmost 1 are ignored. 0000000000 is not allowed, and will default.
 - Set according to the rightmost 10 bits of the immediate integer value in SET STACK SIZE <size> when:
 - The first two lines of the assembled file are SET WORD SIZE {32,64} and SET STACK SIZE <size>
 - The first line of the assembled file is SET STACK SIZE <size>
 - In this case, SET STACK SIZE is not a valid instruction in the second line.
 - Defaults to 1000000000 (1024 state frames stored).
 - Other implementations of the ISA could be shipped with a lower default.
 - Next 10 bits in the first word used as indicator bits for which power of 2 the instruction reversal buffer size is (eg. 0001000000 means it's size 2^7). Bits to the right of the leftmost 1 are ignored. 0000000000 is not allowed, and will default.
 - Set according to the rightmost 10 bits of the immediate integer value in SET BUFFER SIZE <size> when:
 - The first three lines of the assembled file are SET WORD SIZE {32,64} and SET STACK SIZE <size> and SET BUFFER SIZE <size>
 - The first two lines of the assembled file are SET STACK SIZE <size> and SET BUFFER SIZE <size>
 - In this case SET BUFFER SIZE is not a valid instruction in the third line.
 - The first line of the assembled file is SET BUFFER SIZE <size>

- In this case SET BUFFER SIZE is not a valid instruction in the second or third lines.
 - Defaults to 1000000000 (1024 instructions stored).
 - The actual implementation of the ISA could be shipped with a lower default.
 - Next 3 bits of the first word used as indicator of which of each of the above three special instructions were included, for line number calculation purposes.
 - The last 25 bits of the second word are the address of the first word in data memory. This is set by the assembler and is used by load and store instructions.
- SET WORD SIZE is never a valid instruction after the 1st line of the assembled file.
 SET STACK SIZE is never a valid instruction after the 2nd line of the assembled file.

● **GETP <index>, <dest reg start>** **Get Predicate Bits**

32/64 : [1 1 0] [0 0 0] [{0,1}] [ignore×17/49] [<index>×4 <dest reg>×4]

- <index>:
 - Register w/integer = 0 or non-0 value (flag 0)
 - RR-R15
 - Immediate integer = 0 or non-0 (flag 1)
- <dest reg start>:
 - RR-R15

Gets bits from first (index=0) or second (index=non-0) predicate register and stores them in the destination data register.

● **SETP <index>, <src start>{, <mask>}** Set Predicate Bits

32 : [1 1 0] [0 0 1] [c{0,1} d{0,1}] [ignore×17] [<index>×4 <src>×4 <mask>×4]
 64 : [1 1 0] [0 0 1] [a0 b0 c{0,1} d{0,1}] [ignore×47] [<index>×4 <src>×4 <mask>×4]
 [1 1 0] [0 0 1] [a1 b0 c{0,1} d{0,1}] [ignore×19] [<index>×4 <src>×32 <mask>×4]
 [1 1 0] [0 0 1] [a0 b1 c1 d{0,1}] [ignore×19] [<index>×4 <src>×4 <mask>×32]

○ <index>:

- Register w/integer = 0 or non-0 value (flag_d 0)
 - RR-R15
- Immediate integer = 0 or non-0 (flag_d 1)

○ <src start>:

- Register w/bits to copy (flag_a 0)
- Immediate binary (64-bit mode only) (flag_a 1)
 - Only if flag_b=0

○ <mask>:

- Register w/bits to use as mask (flag_c 1)
- Immediate binary mask (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

Only the bits in the mask that are set to 1 have their corresponding bits in the predicate register altered by <src>.

- Default flag_b=0, flag_c=0

Sets the first (index=0) or second (index=non-0) predicate register bits according to the source, masked positively by the '1' bits in the mask if it's present. i.e. If the masks only has two 1s, in the second and eighth bits from the left, then all the other bits in the predicate register will remain the same, regardless of their value in the source or their existing value in the predicate register.

● **GETC <dest reg>** Get Condition Bits

32/64 : [1 1 0] [0 1 0] [ignore×22/54] [<dest reg>×4]

○ <dest reg>:

- RR-R15

Copies the CC register into the destination register.

● SETC <src>{, <mask>} Set Condition Bits

32 : [1 1 0] [0 1 1] [c{0,1}] [ignore×17] [<src>×4 <mask>×4]
 64 : [1 1 0] [0 1 1] [a0 b0 c{0,1}] [ignore×47] [<src>×4 <mask>×4]
 [1 1 0] [0 1 1] [a1 b0 c{0,1}] [ignore×19] [<src>×32 <mask>×4]
 [1 1 0] [0 1 1] [a0 b1 c1] [ignore×19] [<src>×4 <mask>×32]

- <src>:
 - Register w/bits to copy (flag_a 0)
 - Immediate binary (64-bit mode only) (flag_a 1)
 - Only if flag_b=0
- <mask>: (flag_c 1)
 - Register w/bits to use as mask (flag_b 0)
 - Immediate binary (64-bit mode only) (flag_b 1)
 - Only if flag_a=0

Only the bits in the mask that are set to 1 have their corresponding bits in the CC register altered by <src>.

- Default flag_b=0, flag_c=0.

Sets the CC register bits according to the source, masked positively by the '1' bits in the mask if it's present. i.e. If the masks only has two 1s, in the second and eighth bits from the left, then all the other bits in the CC register will remain the same, regardless of their value in the source or their existing value in the CC register.

- UNDO <quantity>{, <skip>} Reverse Writes to Data Registers
 32/64 : [1 1 0] [1 0 0] [a{0,1} b{0,1}] [ignore×4/36] [<quantity>×10 <skip>×10]

- <quantity>:

- Register w/number of instructions to undo (flag_a 0)

- RR-R15

Only the rightmost 10 bits are counted. Those 10 bits are counted as an unsigned integer.

- Immediate integer or binary (flag_a 1)

Only the rightmost 10 bits are counted. Those 10 bits are counted as an unsigned integer.

- <skip>:

- Immediate integer or binary (flag_b 0)

Only the rightmost 10 bits are counted. Those 10 bits are counted as an unsigned integer.

- Register w/number of instructions to not undo (flag_b 1)

- RR-R15

Only the rightmost 10 bits are counted. Those 10 bits are counted as an unsigned integer.

- Default 0

Uses the instruction reversal stack (see below) to undo the most recent <quantity> writes to the data registers, indexed per instruction, not per register. Does not undo the most recent <skip> writes. (This allows maintenance of certain results of computations along with restoration of the rest of the prior state.) Does not undo other register writes such as those to the PC or CC registers.

- HALT <condition> Halt
 32/64 : [1 1 0] [1 0 1] [{0,1}] [ignore×21/53] [<condition>×4]

- <condition>:

- Register w/halt condition bits (flag_a 0)

- Immediate integer (flag_a 1)

Only rightmost 3 bits are counted.

Sets the CC register halt condition bits according to the given exit code, then halts operation.

Exit code #0 is standard exit with no problems.

Exit code #1 is generic erroneous/emergency exit.

Other exit codes not defined.

Internal Instructions (typecode 111)

Special instructions for internal use only. Not assemblable.

What We Learned:

A lot of the objectives of this project were unfamiliar to us and we learned a great deal as we pieced together our final product.

The memory subsystem was complex, but surprisingly unchallenging in the beginning. The extension we did that allows the user to see load and store results before they're officially "finished" really highlighted how tight and neat the conventional methodology is. (i.e. Violating it without breaking it was exceptionally tricky. We had to use an overly-complex call system that explicitly linked successive-layer requests together, and the individual memory modules needed to check each layer of the request to see whether or not they should be ticking down their clocks at a given time.) We did end up needing to perform a partial overhaul of the memory system because we made it before the clock system was finished and before the pipeline was even started. Were we to do something similar again, we would make it even more of a black box and overengineer the API for futureproofing.

The pipeline was a monster to implement and really gave us a healthy respect for the architects who make and manage long-and-multiple-pipeline architectures. Particularly illuminating were the issues with synchronizing the cycling of the pipeline stages and the memory system. Originally, the pipeline was clock-cycled before the memory system, both separately. However, that caused complications with accurate simulation of clock cycle timing and incorrect fetch stage delays, so we needed an overhaul of parts of the pipeline return logic to make it safe to cycle parts of several pipeline stages, then the memory system, then the rest of the pipeline functionality. That gave us a focused perspective on the precision and impact of the nuances of the computer clock.

The problem of constructing the ISA in an interesting, challenging manner without relying on proportionally-unrealistic resources (i.e. too-long words and hardware-ignorant simulation) was especially enlightening. But implementing those instructions within the infrastructure we'd set up felt a bit cheap, since we completely bypassed the hardware challenges that would have come with that. A more interesting (but prohibitively more time-consuming) project would have included actual micro-hardware logic simulation as well.

Experimenting with the simulator benchmarks and different cache configurations was the best way I've seen to drive home just how much CPU time is wasted waiting for memory accesses unless the CPU has several very sophisticated pipelines. And the partitioned progression of the project stages showed us exactly why hardware architecture and ISA's should *not* be developed independently of each other (we had to do *so many* retrofittings, even though we made an active effort to future-proof our code).

This definitely was a learning experience for Riya, as implementation of an ISA was brand new. She learned about the functions of the pipeline, the advantages and disadvantages of how to orient our callstack, and overall all of the pieces that go into the entirety of the project. There were definitely challenging parts and Riya learned throughout the project, where Ryan played a huge role in the understanding of the entire project and its components. He was able to

integrate everything together, and delegate tasks, all while helping Riya work through questions within the project.

Overall, it was a very interesting project, it gave us a much deeper familiarity with the sort of reasoning (and some of the many, many challenges) that are part of ISA×microarchitecture implementation, and it was enjoyable enough that at least Ryan will likely re-implement it from scratch after the semester ends to see exactly how useful the implementation experience can be in future endeavors