LEG

Noah Huppert and Robert Scibelli

Chapter 1

Management Plan

Responsibilities

Team members will act as generalists, working on all parts of the instruction set implementation.

This will give team members an opportunity to learn about writing simulator code, developing user interfaces, and creating tooling.

Code Management

Git will be used to track changes to source code.

Code will be synchronized through GitHub repositories which are owned by a shared GitHub organization.

The master branch of repositories will be stable at all times. Features will be developed in separate branches and merged in via GitHub pull requests. Pull requests will be reviewed by the other team member to ensure correctness and keep everyone up to date on changes.

Work Coordination

The team will use GitHub issues to track and assign work.

Every feature or bug will have its own issue which contains a full description of the work required to complete or fix the feature or bug.

When a team member starts working on a feature or bug they will assign themselves to the associated GitHub issue to ensure duplicate work does not occur.

Testing

Every repository will use GitHub actions (a continuous integration solution) to run integration tests on every commit. Pull requests will be required to pass these tests before they can be merged into the stable master branch.

If new features are added corresponding tests must be added to ensure the correctness of the new features.

If bugs are fixed regression tests must be added to ensure the bug is fixed and does not occur again.

Chapter 2

What We Learned

Noah

When designing the simulator's systems and how they would interact with each other I learned the value of defining clear system boundaries. This made it easy for me to complete small defined tasks when I had the time and design / implement the ISA specification, memory, caches, pipeline, control unit, and GUI. While making it easier for my partner to implement the instructions and assembler to meet a specification. Which when compliant with the specification easily integrated with the other systems.

The process of designing and implementing the simulator helped me understand the nuances of the different system and how small design decisions can have a large impact on the overall outcome. For example how the decision to fetch multiple words into the cache from DRAM is required in order to achieve spacial locality, which can play a large role in performance.

Implementing the simulator gave me a chance to improve some areas of my programming toolbox. I was able to learn a new programming language, Rust, try out a new technology, Web Assembly, as well as learn how to create user interfaces with React Js.

Rob

I learned a tremendous amount after completing this project, and I feel like I now have a solid understanding of how programs gets ran on a processor. We get taught most of this material from CS 335, like pipelines, memory, and assemblers, but after completing this project I now feel that I have solidified that knowledge.

In the pipeline, for example, I remember going through the slides that covered the pipeline in 335 so that I could study for the test, and I thought I knew how it worked. But what I failed to understand is the little things that help make the pipeline work. Like how the interactions get loaded into memory, and the program counter keep track of which instruction to fetch. And when we preform a jump all we're doing is changing the program counter to the address of the instruction we want to start from.

Another example is with the assembler. I remember learning that it loops through the instruction twice, but I never understood why. Now that I have implemented an assembler, it makes total sense why we do that. Not only does it makes the work easier since all I need to do it grab the info that I need, but also we can learn the locations of the labels so that we can insert the positions of those into the jumps. After creating the assembler, this all makes sense.

I have also learned a tremendous amount in the programming language rust as well. Prior to this project I did not know rust, and now I feel confident that I can program anything in rust, and utilize it's unique (and annoying) functions of safe memory borrowing. I feel confident to add this to my resume.

I have also learned a lot more about git than I knew before. I learned how to properly merge branches, properly deal with merge conflicts, and manage a pull request. I think these will be valuable skills to have when entering the workforce.

This project has given me loads of respect for the engineers behind todays processors and ISAs. I had no idea the complexity of work that goes into making something like this, and this is only a simulator.

Chapter 3

How The Simulator Works

The graphical user interface is automatically built and deployed to l-e-g.github.io/simulator via a continuous integration pipeline.

All the simulator code is written in Rust and the graphical user interface is written in JavaScript.

The simulator code is compiled to Web Assembly, which can be run directly in the browser, and invoked by the JavaScript GUI.

In the browser Web Assembly has its own execution environment and memory space. To connect the Web Assembly compiled simulator and JavaScript GUI a cross domain interface is defined. The simulator's internal state is kept in the Web Assembly memory space and only data which needs to be directly displayed by the GUI is shipped over to the JavaScript domain. This allows us to achieve some measure of performance.

The simulator has 6 distinct systems:

JavaScript Land

- GUI

JavaScript & Web Assembly / Rust Land

- Cross Domain Interface

Web Assembly / Rust Land

- Assembler
- Control Unit
- Memory
- Instructions

Simulator GUI

The user interface is written in JavaScript using React JS which allowed us to easily display data values.

It uses the cross domain interface to start the simulation execution in the Web Assembly domain and get data it needs to display to the user out of the Web Assembly memory space.

Simulator Cross Domain Interface

Rust has native support for a Web Assembly compilation target. There also exists a library (`wasm_bindgen`) which defines a standard data serialization format and function call interface that Rust and JavaScript can use to interface with each other.

In this interface we initialize all the other Rust simulator systems and provide hooks for the GUI to use these systems.

Simulator Assembler

The assembler is a standard 2 pass assembler with support for labels. During the first pass we parse instructions and figure out what data we need to pack into bits. On the second pass we resolve the location of labels, allowing for instruction like jump which use these label values.

It can either be executed via the command line. Or the GUI can invoke the assembler on user input.

Simulator Control Unit

The control unit simulates what a control unit would do in hardware. It has access to interfaces for the memory system, an understanding of how to decode all the ISA instructions, and the ability to execute these instructions through a pipeline.

The pipeline is a 5 stage pipeline:

- **Fetch:** Retrieve next instruction's bits from memory
- **Decode:** Interpret instruction operands and retrieve any register values
- **Execute:** Run core instruction logic, typically an ALU operation
- **Memory Access:** Access memory
- **Write Back:** Write results back to registers

Every operation which takes place in the pipeline returns the number of cycles it took to complete. These values are accumulated and added to the total control unit cycle count. This allows us to more efficiently simulate clock delays.

Due to a constraint on the number of man hours available when implementing the pipeline data dependencies are not handled automatically. This is a feature we would have liked to have in the pipeline but could not implement it.

Simulator Memory

The simulator has 4 layers of memory:

- L1 Cache: 16 lines, 1 word each, 1 cycle delay
- L2 Cache: 32 lines, 1 word each, 10 cycle delay
- L3 Cache: 512 lines, 1 word each, 40 cycle delay
- DRAM: 2^{32} address space, 100 cycle delay

All memory layers implement a standard memory interface with a get and set method, which among other things return the number of cycles they took. This allowed us to easily change our memory hierarchy whenever we wanted since every layer is used the exact same way.

Our caches read from the layer beneath them if there was a miss, and evicted data to the layer beneath them if there was a conflict. Unfortunately our caches do not provide much spacial locality because each layer only fetches the word requested and no additional words. This was a result of us running out of time and failing to implement this feature.

The caches are represented as contiguous blocks of memory which are directly indexed into using a portion of the memory address.

The DRAM is represented as a hash map. Although this achieves slightly worse performance it allows us to simulate access to the entirety of our address space. Since most simulated programs are short we determined the access time penalty of a hash map was negligible.

Simulator Instructions

All instructions implement a standard instruction interface which has a method to be run during each stage of the pipeline. Stages which need access to registers or memory are given access via method arguments.

This allowed us to easily write the pipeline since every instruction could be treated the same.

Chapter 4

How to use Simulator

The simulator can be accessed at l-e-g.github.io/simulator.

Using Web Assembly and React Js for our frontend allows us to not only easily create a user interface but also make it accessible to anyone with a web browser.

Once on our website, you can begin using our simulator by writing your own assembly code into the text box, or copying and pasting assembly code in. If you don't want to use assembly code and want to use binary instructions, then you can hit the button "Select File" and a file explorer will pop up and you can select a binary file.

To start the program, there are two options: First is to use the "Step" button, which will walk through the program one instruction at a time. The other option is to use the "Run" button which will execute the entire program at once.

When your program executes the pipeline will begin to fill up with your instructions, and you can watch them flow down the pipeline. At the top the cycle count will grow as the program advances, and the `PC` value will change as the program steps. If your program includes jumps, you can watch the `PC` value change as it jumps around the instructions in memory.

There are a few options that can be set before executing the program: to run with or without cache, and to run with or without the pipeline. When cache is set, the program will utilize the 3 layers of cache to store data. When the cache is not set the program will use all DRAM to store data. When the pipeline is set the program will execute all instructions through the pipeline. When the pipeline is not set the program will execute all stages of the instruction at once.

Chapter 5

Performance Results

The following results are from our Matrix Multiply program that has 63 lines of assembly code.

Pipeline?	Cache?	Clock Cycle Count
Enabled	Enabled	9580
Enabled	Disabled	6367
Disabled	Enabled	9828
Disabled	Disabled	6615

As we can see, running the simulator without the cache is the fastest. This results from the fact that our benchmark did all calculations in registers and only access memory for instructions. This was a flaw in our benchmark which we did not have time to resolve. To further compound this problem our caches do not provide spacial locality, so loading instructions through them will always incur an additional access time penalty.

We can see that running the simulator with the pipeline is faster then without the pipeline. This is an encouraging sign, showing that the pipeline does provide performance increases as it is supposed to.

However due to the limitations discussed above with our cache, running with pipeline and no cache is the fastest.

Chapter 6

Memory

Endianess: Little
Memory Organization: Princeton

Data and Instruction Memory

Word Size: 32 bits
Addressing Unit: Word
Address Space: 2^{32}

Holds data and instructions. Can be manipulated directly via the load and store instructions.

Memory Hierarchy:

1. Level 1, SRAM: direct mapped, 1 cycle delay
2. Level 2, SRAM: direct mapped, 10 cycle delay
3. Level 3, SRAM: direct mapped, 40 cycle delay
4. DRAM: $2^{32} \cdot 32\text{bits} \simeq 17\text{GB}$ (100 cycle delay)

Chapter 7

Registers

Referred to in assembly as `R#` where `#` is a number.
Some registers have aliases.

General Purpose

27 mixed 32-bit registers.

`R0` through `R25`.

Initially all set to 0.

Program Counter

32-bit register.

`R28`, `PC`.

Stores the address of the current instruction being executed.
Initially set to 0.

Status

6-bit register.

`R29`, `STS`.

Initially set to null status with the interrupt flag unset.

See the Status Codes section for details.

Stack Pointer

32-bit register.

`R30`, `SP`.

Stores the address of the bottom of the stack in memory. Initially set to 0.

Link Register

32-bit register.

`R31`, `LR`.

Store the address to return to after a subroutine has completed.
Initially set to 0.

Chapter 8

Status Codes

The status register is 6-bits large.

The least significant 5 bits are used to store the status of comparisons and arithmetic operations, called condition codes.

The most significant bit is an interrupt flag.

Condition Codes

The special condition code `00000` is used to denote a null status `NS`.
This code will match every other condition code.

Valid codes are:

Binary	Assembly	Meaning
`00000`	`NS`	Null status, matches everything
`00001`	`NE`	Not equal
`00010`	`E`	Equal
`00011`	`GT`	Greater than
`00100`	`LT`	Less than
`00101`	`GTE`	Greater than or equal to
`00111`	`LTE`	Less than or equal to
`01000`	`OF`	Overflow
`01001`	`Z`	Zero
`01010`	`NZ`	Not zero
`01011`	`NEG`	Negative
`01100`	`POS`	Positive

Chapter 9

Binary Format

Once programs are assembled they will be in the LEG binary format.

The format is simple:

A little endian binary file. Every 32 bits will be interpreted as a memory address. Memory addresses will increment by one. The contents will be loaded directly into memory.

Program execution will begin at memory address 0. The program should be terminated by a Halt instruction.

Chapter 10

Assembly Format

One instruction per line. The first token in a line is a label. If no label is desired put a blank space or tab. The following tokens may be the mnemonics documented in the instructions section.

Immediate use the following syntax:

- `0dNN`: For unsigned decimal immediate values
- `0xNN`: For unsigned hexadecimal immediate values
- `0bNN`: For unsigned binary immediate values
- `0sdNN`: For signed decimal immediate values
- `0sxNN`: For signed hexadecimal immediate values
- `0sbNN`: For signed binary immediate values

The assembler will convert all immediate values to binary.

Any tokens after the `#` character will not be interpreted, use this for comments.

Chapter 11

Instructions

Assembly Documentation Syntax

Instruction assembly is documented using the following syntax:

- A word in curly brackets signifies a variation of an instruction's mnemonic. A table will be present which specifies valid values, the curly brackets and their contents should be replaced with one of these values.

- Example:

  `DO{OPERATION}`

  `{OPERATION}` indicates that part of the mnemonic must be replaced by a value from the `{OPERATION}` table below.

`{OPERATION}`	Operation
`F`	Foo
`B`	Bar

  For example a mnemonic of `DOF` indicates that the "Foo" operation should take place.

- A word in angle brackets signifies an instruction operand. Look for the "operands" section of the instruction documentation for more detail.

- Example:

  `DO <DEST> <OP1> <OP2>`

  In the above `<DEST>`, `<OP1>`, and `<OP2>` are all operands which should be replaced by operand values when writing assembly.

  For example the assembly line `DO R1 R2 R3` has a `<DEST>` operand value of `R1`, a `<OP1>` operand value of `R2`, and a `<OP2>` operand value of `R3`.

Bit Organization Syntax

Instructions have a bit organization section which details the binary format of the instruction itself.

The format of this section is a table, where the top header row indicates the purpose of the bits, and the box directly beneath each item in the header row indicates how many bits are reserved for the described purpose.

The leftmost part of the table represents the least significant bits, and the rightmost part of the table represents the most significant bits.

Example:

Type	Operation	<DEST>	<OP1>	<OP2>
4	6	4	8	2

Indicates that the "Type" field takes up 4 bits, the "Operation" field takes up 6 bits, the <DEST> field takes up 4 bits, and so on.

The binary number:

```
MSB                    LSB
|                       |
v                       v
10101100101111010001100 1
```

Would translate to the following values for the fields defined in the example:

Field	Value
Type	1001
Operation	010001
<DEST>	1111
<OP1>	10110010
<OP2>	10

Optional Parameters

Sometimes instructions have optional parameters. These parameters are specified by putting them inside square brackets. If this is the case the behavior if the parameter is not provided is documented below.

Example:

The instruction documentation:

`FOO[{TYPE}] <OP1>`

Indicates that the {TYPE} parameter is optional. However the <OP1> operand is still required.

Data Types

There are 2 data types which are supported in instructions:

- 32 bit two's complement integer
- 32 bit unsigned integer

Instructions have type variations when it matters, bit level operations do not.

Condition Fields

All instructions currently have space for a condition field.

This field allows for predicated execution of instructions.

Currently only the jump instructions use this condition field.

All other instructions do not use this field at the moment, in the future they may. For right now the condition field will be set to null status.

Instruction Types

There are 3 instruction types:

Type Field Binary	Type
`00`	Control
`01`	ALU
`10`	Memory
`11`	Graphics

Arithmetic Logic Unit

Instructions:

Typed arithmetic instructions:

- Add
- Subtract
- Multiply
- Divide

Typed general instructions:

- Compare
- Shift
- Arithmetic Right
- Arithmetic Left

Untyped general instructions:

- Shift
- Logical Right
- Logical Left
- 3 operand logic
- And
- Or
- Xor
- 2 operand logic
- Not
- Move

Bit Organization:

The operation field of each ALU instruction has the following meaning:

Decimal	Operation
0	Add unsigned integer register direct
1	Add signed integer register direct
2	Add unsigned integer immediate
3	Add signed integer immediate
4	Subtract unsigned integer register direct
5	Subtract signed integer register direct
6	Subtract unsigned integer immediate
7	Subtract signed integer immediate
8	Multiply unsigned integer register direct
9	Multiply signed integer register direct
10	Multiply unsigned integer immediate
11	Multiply signed integer immediate
12	Divide unsigned integer register direct
13	Divide signed integer register direct
14	Divide unsigned integer immediate
15	Divide signed integer immediate
-	-
16	Move
-	-
17	Compare
-	-
18	Arithmetic shift left register direct
19	Arithmetic shift right register direct
20	Arithmetic shift left immediate
21	Arithmetic shift right immediate
-	-
22	Logical shift left register direct
23	Logical shift left immediate
24	Logical shift right register direct
25	Logical shift right immediate
-	-
26	And register direct
27	And immediate
28	Or register direct
29	Or immediate
30	Xor register direct
31	Xor immediate
-	-
32	Not

Arithmetic Instructions

Assembly:

`{OPERATION}{TYPE} <DEST> <OP1> <OP2>`

5 operations * 2 types * 2 addressing modes = 20 total instructions.

Bit Organization:

Register direct:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>	Not Used
5	2	6	5	5	5	4

Immediate:

Condition	Type	Operation	<DEST>	<OP1>	<OP2>
5	2	6	5	5	9

Behavior:

Performs a basic arithmetic operation, determine by {OPERATION}:

{OPERATION}	Behavior
ADD	<OP1> + <OP2>
SUB	<OP1> - <OP2>
MLT	<OP1> * <OP2>
DIV	<OP1> / <OP2>
MOD	<OP1> % <OP2>

The type of numbers used in the arithmetic operation is specified by appending {TYPE}:

{TYPE}	Type
U	Unsigned integer
S	Signed integer

{TYPE} defaults to unsigned integer mode.

In the case that the MLT instruction yields a result that is larger than 32 bits, the result will be cut off, and only the first 32 least significant bits will be used.

Operands:

- <DEST>: Register to store result
- <OP1>: Register containing first number
- <OP2>: Register containing second number or a 9-bit immediate value

Move

Assembly:

MOV <DEST> <SRC>

1 total instruction.

Bit Organization:

Condition	Type	Operation	<DEST>	<SRC>	Not Used
5	2	6	5	5	9

Behavior:

Transfers the contents of the `<SRC>` register to the `<DEST>` register.

Operands:

- `<DEST>`: The destination register
- `<SRC>`: The source register

Compare

Assembly:

`CMP[{TYPE}] <OP1> <OP2>`

1 instruction.

Bit Organization:

Condition	Type	Operation	`<OP1>`	`<OP2>`	Not Used
5	2	6	5	5	9

Behavior:

Compares `<OP1>` to `<OP2>` and stores the result in the status register.

Operands:

- `<OP1>`: Register containing first number to compare, on the left hand side of the comparison
- `<OP2>`: Register containing number to compare to `<OP1>`, on the right hand side of the comparison

Arithmetic Shift

Assembly:

`AS{DIRECTION} <DEST> <OP1>`

2 directions * 2 addressing modes: 4 total instructions.

Bit Organization:

`<OP1>` register direct:

Condition	Type	Operation	`<DEST>`	`<OP1>`	Not Used
5	2	6	5	5	9

`<OP1>` immediate:

Condition	Type	Operation	`<DEST>`	`<OP1>`
5	2	6	5	14

Behavior:

Performs an arithmetic shift (respects the sign of the number) on `<DEST>` and stores the result in `<DEST>`. Shifted by the amount specified in `<OP1>`.

<OP1> can either be an immediate value or a register.

The direction bits are shifted is specified by {DIRECTION}:

{DIRECTION}	Direction
L	Left
R	Right

Operands:

- <DEST>: Destination register
- <OP1>: 14-bit immediate value or register which contains amount to shift

Logical Shift

Assembly:

LS{DIRECTION} <DEST> <OP1>

2 directions * 2 addressing modes: 4 total instructions.

Bit Organization:

<OP1> register direct:

Condition	Type	Operation	<DEST>	<OP1>	Not Used
5	2	6	5	5	9

<OP1> immediate:

Condition	Type	Operation	<DEST>	<OP1>
5	2	6	5	14

Behavior:

Performs a logical shift (ignores the sign of the number) on <DEST> and stores the result in <DEST>. The amount to shift is specified by <OP1>.

<OP1> can either be an immediate value or a register.

The direction bits are shifted is specified by {DIRECTION}:

{DIRECTION}	Direction
L	Left
R	Right

Operands:

- <DEST>: Destination register
- <OP1>: 14-bit immediate value or register which contains amount to shift.

3 Operand Logic

Assembly:

`{OPERATION} <DEST> <OP1> <OP2>`

3 operations * 2 addressing modes = 6 total instructions.

Bit Organization:

`<OP2>` register direct:

Condition	Type	Operation	`<DEST>`	`<OP1>`	`<OP2>`	Not Used
5	2	6	5	5	5	4

`<OP2>` immediate:

Condition	Type	Operation	`<DEST>`	`<OP1>`	`<OP2>`
5	2	6	5	5	9

Behavior:

Performs a logic operation on `<OP1>` and `<OP2>` and stores the result in the `<DEST>` register.

The logic operation is specified by `{OPERATION}`:

`{OPERATION}`	Operation
`AND`	And
`OR`	Or
`NOT`	Not

Operands:

- `<DEST>`: Register result will be placed
- `<OP1>`: Register containing value to perform logic operation on
- `<OP2>`: 9-bit immediate value or register to use as second operand in logic operation

Not

Assembly:

`NOT <DEST> <OP1>`

1 total instruction.

Bit Organization:

Condition	Type	Operation	`<DEST>`	`<OP1>`	Not Used
5	2	6	5	5	9

Behavior:

Inverts all the bits in `<OP1>` and stores them in `<DEST>`.

Operands:

- `<DEST>`: Register to store result
- `<OP1>`: Register containing value to invert

Memory

5 total instructions.

Word based operations:

- Load
- Store
- Push
- Pop

Bit Organization:

The operation field of each memory instruction has the following meaning:

Binary	Operation
`0`	Load register direct
`1`	Load immediate
`2`	Store register direct
`3`	Store immediate
`4`	Push
`5`	Pop

Load

Assembly:

`LDR <DEST> <ADDR>`

2 addressing modes = 2 total instructions.

Bit Organization:

Register direct:

Condition	Type	Operation	`<DEST>`	`<ADDR>`	Not used
5	2	3	5	5	12

Immediate:

Condition	Type	Operation	`<DEST>`	`<ADDR>`
5	2	3	5	17

Behavior:

Reads a word of memory from the address specified by the `<ADDR>` register into the `<DEST>` register.

If in the immediate form the address immediate is sign extended and added to the incremented program

counter value: `PC + Immediate Value`.

Operands:

- `<DEST>`: Register to store result
- `<ADDR>`: Register or signed immediate value which will be added to `PC + 1` containing the memory address to access

Store

Assembly:

`STR <SRC> <ADDR>`

2 addressing modes = 2 total instructions.

Bit Organization:

Register direct:

Condition	Type	Operation	`<SRC>`	`<ADDR>`	Not Used
5	2	3	5	5	12

Immediate:

Condition	Type	Operation	`<SRC>`	`<ADDR>`
5	2	3	5	17

Behavior:

Writes a word of data from the `<SRC>` register to the memory address specified by the `<ADDR>` register.

If in the immediate form the address immediate is sign extended and added to the incremented program counter value: `PC + Immediate Value`.

Operands:

- `<SRC>`: Register containing data
- `<ADDR>`: Register or signed immediate field added to `PC + 1` containing the memory address to store data

Push

Assembly:

`PUSH <SRC>`

1 total instruction.

Bit Organization:

Condition	Type	Operation	`<SRC>`	Not Used
5	2	3	5	17

Behavior:

Decrements the stack pointer and stores the contents of the `<SRC>` register at the address specified by stack pointer.

Operands:

- `<SRC>`: Register containing the data to be stored on stack

Pop

Assembly:

```
POP <DEST>
```

1 total instruction.

Bit Organization:

Condition	Type	Operation	`<DEST>`	Not Used
5	2	3	5	17

Behavior:

Reads a word from the memory address specified by the stack pointer register into the `<DEST>` register. Then increments the stack pointer register by one.

Operands:

- `<DEST>`: The destination register for data being popped off stack

Control

- Halt
- Jump

Bit Organization:

The operation field of each memory instruction has the following meaning:

Binary	Operation
0	Halt
1	Jump Register Direct
2	Jump Immediate
3	Jump Subroutine Reg. Dir.
4	Jump Subroutine Imm.
5	No Op

Halt

Assembly:

```
HALT
```

1 instruction.

Bit Organization:

Condition	Type	Operation	Not Used
5	2	3	21

Behavior:

Signals to the hardware that the program is done running. Any instructions after this will not be interpreted.

Note that due to the way the bit pattern is constructed a memory value of `0` will be interpreted as a halt instruction:

- Condition `00000` is the null status
- Type `00` is the control type
- Operation `000` is the halt instruction
- The rest of the bits are unused

Jump

Assembly:

`<CONDITION>JMP{SPECIAL} <ADDR>`

2 addressing modes + 2 types = 4 total instructions.

Bit Organization:

Register direct:

Condition	Type	Operation	`<ADDR>`	Not Used
5	2	3	5	19

Immediate:

Condition	Type	Operation	`<ADDR>`
5	2	3	24

Behavior:

Conditionally executes a jump based on if the `<CONDITION>` operand matches the condition in the status register.

By default jump just sets the program counter as described in the next paragraph. The `{SPECIAL}` part of the instruction can be set to perform a modified behavior jump:

`{SPECIAL}`	Behavior
`S`	Subroutine jump
`(Empty)`	Normal jump

A subroutine jump sets the link register to the program counter register plus one. Then it performs a normal

jump.

A normal jump sets the program counter register to the value specified by the `<ADDR>` operand.

In the register direct version of this instruction the program counter is set to the value in the `<ADDR>` register.

In the immediate version of this instruction the `<ADDR>` value is added to the program counter and the program counter is set to the result.

Operands:

- `<ADDR>`: Register containing new program counter value or a 24-bit immediate

No Op

Assembly:

`NOOP`

1 instruction.

Bit Organization:

Condition	Type	Operation	Not Used
5	2	3	21

Behavior:

Special instruction that does not do anything in the program, and acts as a filler.

This can be used when there are data dependencies in the assembly code, and need to introduce bubbles in the pipeline.