1 Introduction

We have spent the last few weeks implementing our 32-bit datapath. The simple 32-bit GT-2200 is capable of performing advanced computational tasks and logical decision making. Now it is time for us to move on to something more advanced–the upgraded GT-2200a enables the ability for programs to be interrupted. Your assignment is to fully implement and test interrupts using the provided datapath and Brandonsim. You will hook up the various interrupt and data lines to the new devices, modify the datapath and microcontroller to support interrupt operations, and write interrupt handlers to operate these new devices.

2 Requirements

This project is broken up into two parts:

- Part 1 - Implementing a basic interrupt.
- Part 2 - Implementing priority and and second timer.

Before you begin, please ensure you have done the following:

- Download the proper version of Brandonsim. A copy of Brandonsim is available under Resources on T-Square.
- Brandonsim is not perfect and does have small bugs. In certain scenarios, files have been corrupted and students have had to re-do the entire project. Please back up your work using some form of version control, such as a local git repository.

3 What We Have Provided

- A reference guide to the GT-2200a is located in *Appendix B: GT-2200a Instruction Set Architecture*. **Please read this first before you move on!** The reference introduces several new instructions that you will implement for this project.
- An *incomplete* GT-2200a datapath circuit (`GT-2200a-part1.circ`) that you may add the basic interrupt support onto. You are also free to build off of your own Project 1 datapath. Most of the work can be easily carried over from one datapath to another.
- A Brandonsim library (`project2-devices.circ`) containing several subcircuits you will use for this project. **To load the library into an existing Brandonsim file, use Project → Load Library → Logisim Library.**
- A new microcode spreadsheet template `microcode.xlsx` with additional columns for the new signals that will be added in this project. We've provided you a complete microcode that meets the requirements of Project 1, but feel free to supply your own.
- A timer device that will generate an interupt signal at specified intervals. The pinout and functionality of this device are described in *Adding First External Timer Device*.
- A priority fence subcircuit that will be useful when implementing the priority feature in Part 2.
- An *incomplete* assembly program `prj2.s` that you will complete across both parts and use to test your interrupt capabilities.
- An assembler with support for the new instructions to assemble the test program.

4 Part 1: Implementing a Basic Interrupt

For this part of the assignment, you will add interrupt support to the GT-2200a datapath. Then, you will test your new capabilities to handle interrupts using an external timer device.

Work in the GT-2200a-part1.circ file. You will complete all work for Part 1 in this file. If you wish to use your existing datapath, make a copy with this name, and copy the Timer and Priority Fence subcircuits from the file we provided.

4.1 Initial Interrupt Hardware Support

First, you will need to add the initial hardware support for interrupts.

You must do the following:

1. Our processor needs a way to turn interrupts on and off. Create a new one-bit "Interrupt Enable" (IE) register. You'll connect this register to your microcontroller in a later step.
2. Create the INT line. The external device you will create in 4.2 will pull this line high (assert a '1') when they wish to interrupt the processor. Because in later part of this project, multiple devices can share a single INT line, the devices must use a tri-state circuit, similar to the main bus. When a device does not have an interrupt, it neither pulls the line high or low. To ensure your INT line reads as low (i.e., '0') when no devices are requesting an interrupt, add a pull-down resistor (Brandonsim contains a component to do this).
3. When a device receives an **IntAck** signal, it will drive a 32-bit device ID onto the I/O data bus. To prevent misbehaving devices from interfering with the processor, the I/O data bus is attached to the main bus with a tri-state driver. Create this driver and the bus, and attach the microcontroller's **DrIO** signal to the driver.
4. Modify the datapath so that the PC starts at 0x10 when the processor is reset. Normally the PC starts at 0x00, however we need to make space for the interrupt vector table (IVT). Therefore, when you actually load in the test code that you will write, it needs to start at 0x10. Please make sure that your solution ensures that datapath can never execute from below 0x10 - or in other words, force the PC to drive the value 0x10 if the PC is pointing in the range of the vector table.
5. Create hardware to support selecting the register $k0 within the microcode. This is needed by some interrupt related instructions. Because we need to access $k0 outside of regular instructions, we cannot use the Rx / Ry / Rz bits. **HINT:** Use only the register selection bits that the main ROM already outputs to select $k0.

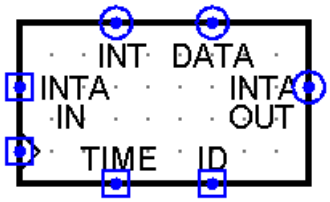
4.2 Adding First External Timer Device

Hardware timers are an essential device in any CPU design. They allow the CPU to monitor the passing of various time intervals, without dedicating CPU instructions to the cause.

The ability of timers to raise interrupts also enables preemptive multitasking, where the operating system periodically interrupts a running process to let another process take a turn. Timers are also essential to ensuring a single misbehaving program cannot freeze up your entire computer.

You will connect a external timer device to the datapath. And it should have a device ID of 0x1 and a 1000-cycle tick timer interval

The pinout of the timer device is described below. If you like, you may also examine the internals of the device in Brandonsim.



- **INT**: The device will begin to assert this line when its time interval has elapsed. It will not be lowered until the device receives an INTA signal.
- **INTA IN** and **INTA OUT**: When the INTA IN line is asserted while the device has asserted the INT line, it will lower the INT line and drive its device ID to the DATA line in the next clock cycle. If the device receives an INTA signal while it has not asserted INT, it will pass the signal onto the next device through INTA OUT. This functionality can be used to daisy-chain devices.
- **ID** and **DATA**: The user may configure the device's ID through the ID pin. The device ID is passed to the DATA pin when the device receives an INTA signal after asserting the INT line.
- **TIME**: The user may configure the device's timer interval through this pin. The interval is specified in number of clock cycles. When the interval has elapsed, the device will raise the INT line.

The INT and ID lines from the timer should be connected to the appropriate buses that you added in the previous section.

4.3 Microcontroller Interrupt Support

Before beginning this part, be sure you have read through *Appendix A: Microcontroller Unit* and *Appendix B: GT-2200a Instruction Set Architecture* and pay special attention to the new instructions.

In this part of the assignment you will modify the microcontroller and the microcode of the GT-2200a to support interrupts. You will need to do the following:

1. Be sure to read the appendix on the microcontroller before starting this section.
2. Modify the microcontroller to support asserting five new signals (You could ignore the **PriClr** signal for Part 1):
   (a) **LdInt** & **EnInt** to control whether interrupts are enabled/disabled. You will use these 2 signals to control the value of your interrupts enabled register.
   (b) **IntAck** to send an interrupt acknowledge to the device.
   (c) **DrIO** to drive the value on the I/O bus to the main bus.
   (d) **PriClr** will be used in Part 2. Include this signal as an output from your ROM but do not connect it to anything for now.
3. Extend the size of the ROM accordingly.
4. Add the fourth ROM described in *Appendix A: Microcontroller Unit* to handle onInt.
5. Modify the FETCH macrostate microcode so that we actively check for interrupts. Normally this is done within the INT macrostate (as described in Chapter 4 of the book and in the lectures) but we are rolling this functionality in the FETCH macrostate for the sake of simplicity. You can accomplish this by doing the following:
   (a) First check to see if the CPU should be interrupted. To be interrupted, two conditions must be true: (1) interrupts are enabled (i.e., the IE register must hold a '1'), and (2), a device must be asserting an interrupt.

(b) If not, continue with FETCH normally.

(c) If the CPU should be interrupted, then perform the following:

i. Save the current PC to the register $k0.

ii. Disable interrupts.

iii. Assert the interrupt acknowledge signal (IntAck). Next, drive the device ID from the I/O bus and use it to index into the interrupt vector table to retrieve the new PC value. The should be done in the same clock cycle as the IntAck assertion.

iv. This new PC value should then be loaded into the PC.

Note: `onInt` works in the same manner that ChkCmp did in Project 1. The processor should branch to the appropriate microstate depending on the value of `onInt`. `onInt` should be true when interrupts are enabled AND when there is an interrupt to be acknowledged.

Note: The mode bit mechanism discussed in the textbook has been omitted for simplicity.

6. Implement the microcode for three new instructions for supporting interrupts as described in Chapter 4. These are the EI, DI, and RETI instructions. You need to write the microcode in the main ROM controlling the datapath for these three new instructions. Keep in mind that:

(a) EI sets the IE register to 1.

(b) DI sets the IE register to 0.

(c) RETI loads $k0 into the PC, and enables interrupts.

4.4 Implementing the Timer Interrupt Handler

Our datapath and microcontroller now fully support interrupts from devices, BUT we must now implement the first interrupt handler `t1_handler` within the `prj2.s` file to support interrupts from the timer device while also not interfering with the correct operation of any user programs.

In `prj2.s`, we provide you with a program that runs in the background. For this part of the project, you need to write interrupt handler for only one timer device (device IDs 0x1). You should refer to Chapter 4 of the textbook to see how to write a correct interrupt handler. As detailed in that chapter, your handler will need to do the following:

1. First save the current value of $k0 (the return address to where you came from to the current handler)
2. Enable interrupts (which should have been disabled implicitly by the processor within the INT macrostate).
3. Save the state of the interrupted program.
4. Implement the actual work to be done in the handler. In the case of this project, we want you to **increment a counter variable in memory**, which we have already provided.
5. Restore the state of the original program and return using RETI.

The handler you have written for the timer device should run every time the device's interrupt is triggered. Make sure to write the handler such that interrupts can be nested. With that in mind, interrupts should be enabled for as long as possible within the handlers.

You will need to do the following:

1. Write the interrupt handler (should follow the above instructions or simply refer to Chapter 4 in your book). In the case of this project, we want the interrupt handler to keep time in memory at the predetermined location: 0xFFFFFD

2. Load the starting address of the first handler you just implemented in `prj2.s` into the interrupt vector table at the appropriate addresses (the table is indexed using the device ID of the interrupting device).

Test your design. If it works correctly, you should see a location in memory increment as the program runs.

5 Part 2: Implementing Priority and and Second Timer

Our datapath and microcontroller currently can support multiple devices through daisy chaining. However, this is often an inefficient approach, especially when one device connected to the processor is considered more important than another.

In this part of the project you will implement a priority mechanism in hardware such that only interrupts of higher priority than what is currently running will be processed. You will then connect a second timer device with a higher priority level.

For Part 2, make a copy of your Part 1 circuit file and name it GT-2200a-part2.circ. You must submit both files when you turn in the assignment. The microcode and assembly files can be shared between both parts.

5.1 About the Priority Encoder and Priority Fence

In order to implement the priority mechanism, you will make use of two devices–a priority encoder and the priority fence.

5.1.1 Priority Encoder

A **priority encoder** takes in multiple priority input lines. Each line corresponds to a numeric "priority". The encoder then selects the highest priority number and outputs this value. You may use Brandonsim's built-in priority encoder for this purpose, or you may make your own.

In following the typical operating system convention, we designate **lower numbers as higher priority**. Thus priority 0 is the highest priority. The GT-2200a supports only two priority levels, 0 and 1.

5.1.2 Priority Fence

When the CPU is handling an interrupt of priority n, it should only respond to an interrupt of higher priority, i.e. $x < n$. The **priority fence** device implements this logic using a hardware stack. It sits between the priority encoder and the CPU.

When the priority fence sees an interrupt acknowledge at level n, it locks the level n into the "fence" register. It will then only output an incoming interrupt to the CPU if its priority is less than n, i.e. it is of higher priority than the currently running interrupt.

When the CPU acknowledges a higher priority interrupt, it must remember the previous fence value for when the higher-priority interrupt returns. When the CPU returns from an interrupt, it signals the priority fence with the **PriClr** signal to indicate that it may return to its previous level.

We have designed this device and provided it to you.

5.2 Adding the Priority Encoder and Priority Fence

You must do the following:

1. Add a second timer device to your circuit, with reset time 600 and device ID 0x2. This device should use the same I/O bus as the first timer, but will have separate INT and INTA lines. **Do not daisy-chain this timer to the first timer.**

2. Add a priority encoder to your datapath circuit. This can be found in Brandonsim under the *Plexers* tab. The encoder should have two priority levels: level 0 (high priority) and level 1 (low priority). You should attach the second timer's INT line to level 0 and the first timer's INT line to level 1. Because higher-resolution timers (those with shorter intervals) should always take precedence over lower-resolution timers. The priority encoder should output two things:

   - Whether an interrupt is currently asserted.
   - The priority level of the asserted interrupt. If both lines are asserted, this will be the higher priority of the two.

   NOTE: Although the priority encoder in Brandonsim considers 1 to be the higher priority, in real-world operating system development, lower numbers normally correspond to higher priority. Therefore, your priority encoder should consider level 0 to be high priority and level 1 to be low priority. **Make sure that you account for this in your design.**

3. Add the priority fence to your datapath circuit. This is a device provided to you as part of the template. Examine the internals of this circuit and make sure you understand its purpose, as this may be a demo question. Attach the outputs of the priority encoder, as well as the **IntAck** line from the microcontroller, to the appropriate inputs on the fence.

4. The **IntAck** signal should only be passed to devices of the currently asserted priority level. Otherwise, a device of a lower priority could see the same signal and also attempt to respond to the CPU. You must ensure through hardware that this signal is passed to the correct device.

5. Finally, when we finish handling an interrupt, we must tell the priority fence to resume allowing interrupts at that priority level. To accomplish this, attach the **PriClr** line from the microcontroller to the priority fence, and update your RETI instruction to assert this line.

5.3 Implement the Second Timer Interrupt Handler

Now that our priority mechanism is working, we are ready to process interrupts from the second timer.

Building off of your work for Part 1 in `prj2.s`, write a separate handler for the second timer device that increments a separate variable at memory address 0xFFFFFE.

You may be able to copy most of your handler code from the first handler to the second. Also, remember to update the interrupt vector table to contain the address of your second timer interrupt handler.

6 Deliverables

Please submit all of the following files in a **.tar.gz** archive generated by our Makefile.

The Makefile will work on any Unix or Linux-based machine (on Ubuntu, you may need to `sudo apt-get install build-essential` if you have never installed the build tools).

Run `make submit` to automatically package your project into the correct archive format. The generated archive should contain at a minimum the following files:

- GT-2200a-part1.circ
- GT-2200a-part2.circ
- microcode.xlsx
- assembly/prj2.s

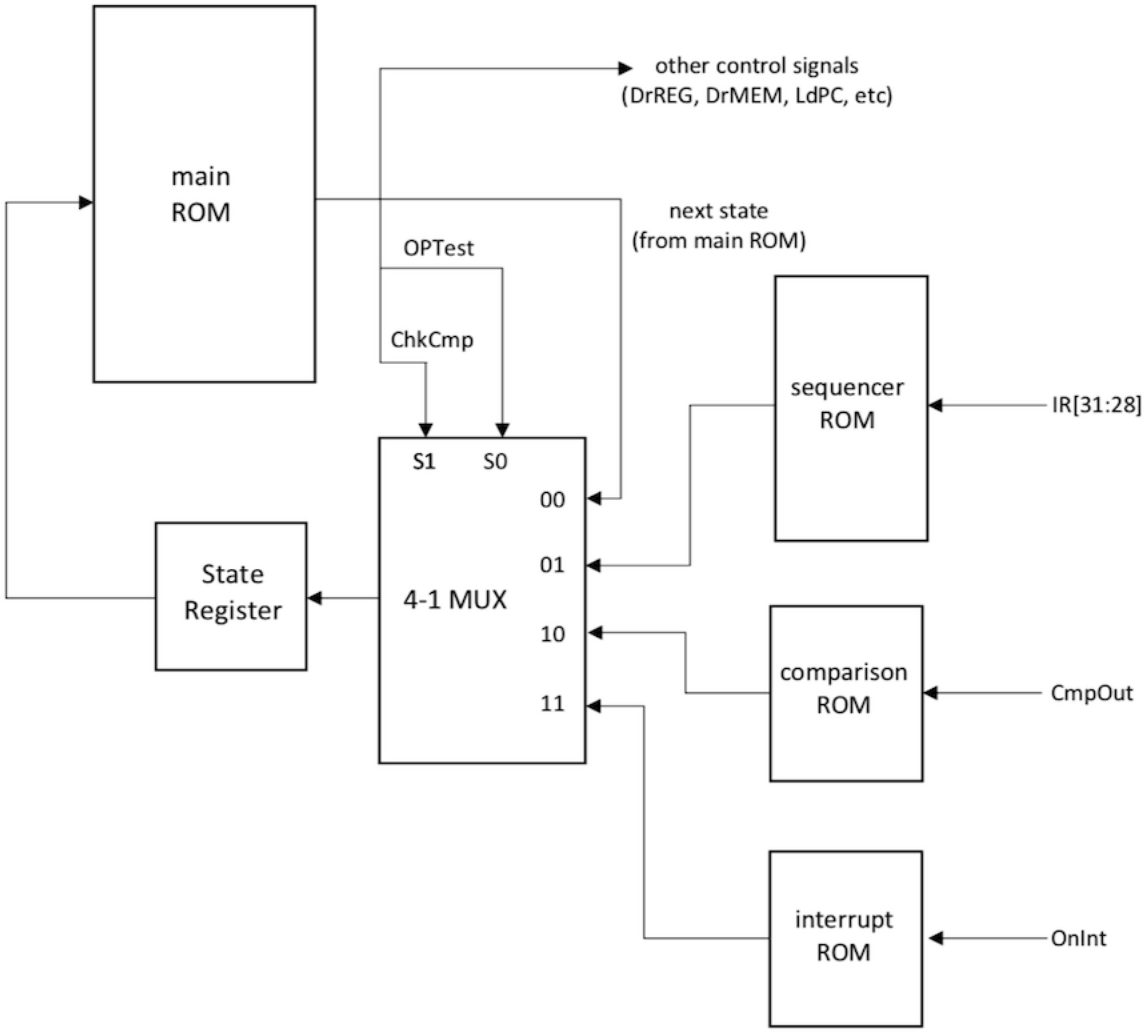
You must submit separate circuit files for Part 1 and Part 2 in order to receive credit. Both should work with the submitted microcode and assembly files. This ensures you can receive partial credit for your work on Part 1 independent of Part 2.

Always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.

This project will be demoed. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released.

7 Appendix A: Microcontroller Unit

As you may have noticed, we currently have an unused input on our multiplexer. This gives us room to add another ROM to control the next microstate upon an interrupt. You need to use this fourth ROM to generate the microstate address when an interrupt is signaled. The input to this ROM will be controlled by your interrupt enabled register and the interrupt signal asserted by the timer interrupt from the part 1 and 2. This fourth ROM should have a 2-bit input and 6-bit output. The most significant input bit of the ROM should be set to 0.



The outputs of the FSM control which signals on the datapath are raised (asserted). Here is more detail about the meaning of the output bits for the microcontroller:

Table 1: ROM Output Signals

Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose
0	NextState[0]	7	DrMEM	14	LdA	21	ALULo	28	DrIO
1	NextState[1]	8	DrALU	15	LdB	22	ALUHi	29	PriClr
2	NextState[2]	9	DrPC	16	LdCmp	23	OPTest		
3	NextState[3]	10	DrOFF	17	WrREG	24	ChkCmp		
4	NextState[4]	11	LdPC	18	WrMEM	25	LdInt		
5	NextState[5]	12	LdIR	19	RegSelLo	26	EnInt		
6	DrREG	13	LdMAR	20	RegSelHi	27	IntAck		

Table 2: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX (IR[27:24])
0	1	RY (IR[23:20])
1	0	RZ (IR[3:0])
1	1	$k0

8 Appendix B: GT-2200a Instruction Set Architecture

The GT-2200a is a simple, yet capable computer architecture. The GT-2200a combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

The GT-2200a is a **word-addressable**, **32-bit** computer. **All addresses refer to words**, i.e. the first word (four bytes) in memory occupies address 0x0, the second word, 0x1, etc.

All memory addresses are truncated to 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

8.1 Registers

The GT-2200a has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

Table 3: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Reserved for the Assembler	NA
2	$v0	Return Value	No
3	$a0	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No

1. **Register 0** is always read as zero. Any values written to it are discarded. **Note:** for the purposes of this project, you must implement the zero register. Regardless of what is written to this register, it should always output zero.
2. **Register 1** is a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.
3. **Register 2** is where you should store any returned value from a subroutine call.
4. **Registers 3 - 5** are used to store function/subroutine arguments. **Note:** registers 2 through 8 should be placed on the stack if the caller wants to retain those values. These registers are fair game for the callee (subroutine) to trash.
5. **Registers 6 - 8** are designated for temporary variables. The caller must save these registers if they want these values to be retained.
6. **Registers 9 - 11** are saved registers. The caller may assume that these registers are never tampered with by the subroutine. If the subroutine needs these registers, then it should place them on the stack and restore them before they jump back to the caller.

7. **Register 12** is reserved for handling interrupts. You should use this to store the return address when an interrupt occurs.
8. **Register 13** is your anchor on the stack. It keeps track of the top of the activation record for a subroutine.
9. **Register 14** is used to point to the first address on the activation record for the currently executing process. Don't worry about using this register.
10. **Register 15** is used to store the address a subroutine should return to when it is finished executing.

8.2 Instruction Overview

The GT-2200a supports a variety of instruction forms. The instructions we will implement in this project are summarized below.

Table 4: GT-2200a Instruction Set

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ADD	0000	DR	SR1	unused SR2
ADDI	0001	DR	SR1	immval20
NAND	0010	DR	SR1	unused SR2
SKP	0011	mode	SR1	unused SR2
GOTO	0100	0000	unused	PCoffset20
LEA	0101	DR	unused	PCoffset20
EI	0110	unused		
DI	0111	unused		
LW	1000	DR	BaseR	offset20
SW	1001	SR	BaseR	offset20
RETI	1010	unused		
JALR	1100	AT	RA	unused
HALT	1111	unused		

8.2.1 Conditional Branching

Conditional branching in the GT-2200a ISA is provided via two instructions: the SKP ("skip") instruction and the GOTO ("unconditional branch") instruction.

The SKP instruction compares two registers and skips the immediately following instruction if the comparison evaluates to true. If the action to be conditionally executed is only a single instruction, it can be placed immediately following the SKP instruction. Otherwise a GOTO can be placed following the SKP instruction to branch over to a longer sequence of instructions to be conditionally executed.

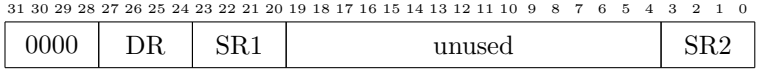
8.3 Detailed Instruction Reference

8.3.1 ADD

Assembler Syntax

```
ADD    DR, SR1, SR2
```

Encoding



Operation

```
DR = SR1 + SR2;
```

Description

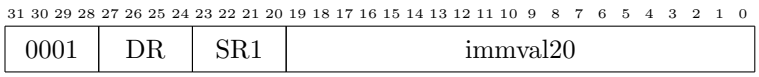
The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

8.3.2 ADDI

Assembler Syntax

```
ADDI   DR, SR1, immval20
```

Encoding



Operation

```
DR = SR1 + SEXT(immval20);
```

Description

The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

8.3.3 NAND

Assembler Syntax

```
NAND   DR, SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
0010	DR	SR1	unused	SR2

Operation

```
DR = ~(SR1 & SR2);
```

Description

The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

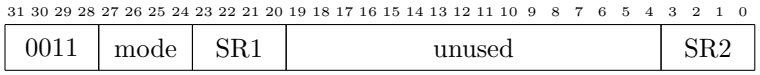
> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> `NAND DR, SR1, SR1`
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

8.3.4 SKP

Assembler Syntax

```
SKPE    SR1, SR2
SKPGT   SR1, SR2
```

Encoding



`mode` is defined to be `0x0` for SKPE, and `0x1` for SKPGT.

Operation

```
if (MODE == 0x0) {
    if (SR1 == SR2) PC = PC + 1;
} else if (MODE == 0x1) {
    if (SR1 > SR2) PC = PC + 1;
}
```

Description

The SKP instruction compares the source operands SR1 and SR2 as signed two's-complement integers according to the rule specified by the mode field. For mode `0x0`, the comparison succeeds if SR1 equals SR2. For mode `0x1`, the comparison succeeds if SR1 is greater than SR2.

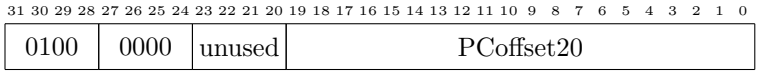
If the comparison succeeds, the incremented PC (address of instruction + 1) is incremented again, for a resulting PC of (address of instruction + 2). **This effectively "skips" the immediately following instruction.** If the comparison fails, the program continues execution as normal.

8.3.5 GOTO

Assembler Syntax

```
GOTO   LABEL
```

Encoding



Operation

```
PC = PC + SEXT(PCoffset20);
```

Description

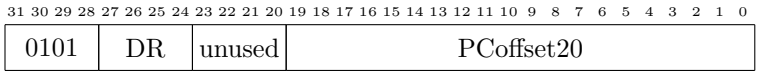
The program unconditionally branches to the location specified by adding the sign-extended PCoffset20 field to the incremented PC (address of instruction + 1). **In other words, the PCoffset20 field specifies the number of instructions, forwards or backwards, to branch over.**

8.3.6 LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding



Operation

```
DR = PC + SEXT(PCoffset20);
```

Description

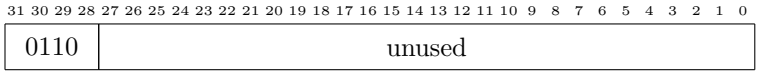
An address is computed by sign-extending bits [19:0] to 32 bits and adding this result to the incremented PC (address of instruction + 1). This instruction effectively performs the same computation as the GOTO instruction, but rather than performing a branch, merely stores the computed address into register DR.

8.3.7 EI

Assembler Syntax

```
EI
```

Encoding



Operation

```
IE = 1;
```

Description

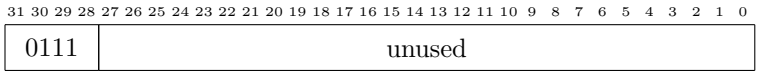
The Interrupts Enabled register is set to 1, enabling interrupts.

8.3.8 DI

Assembler Syntax

```
DI
```

Encoding



Operation

```
IE = 0;
```

Description

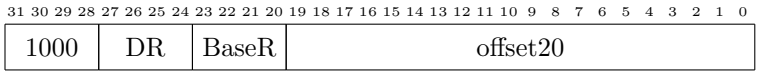
The Interrupts Enabled register is set to 0, disabling interrupts.

8.3.9 LW

Assembler Syntax

```
LW    DR, offset20(BaseR)
```

Encoding



Operation

```
DR = MEM[BaseR + SEXT(offset20)];
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

8.3.10 SW

Assembler Syntax

```
SW    SR, offset20(BaseR)
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1001	SR	BaseR	offset20

Operation

```
MEM[BaseR + SEXT(offset20)] = SR;
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

8.3.11 RETI

Assembler Syntax

```
RETI
```

Encoding

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1010	unused

Operation

```
PC = $k0;
IE = 1;
ClearPriority();
```

Description

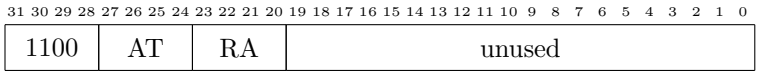
The PC is restored to the return address stored in $k0. The Interrupts Enabled register is set to 1, enabling interrupts. Also, a clear priority signal is sent to the priority fence, restoring the fence register to its previous state.

8.3.12 JALR

Assembler Syntax

```
JALR   AT, RA
```

Encoding



Operation

```
RA = PC;
PC = AT;
```

Description

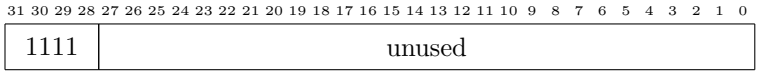
First, the incremented PC (address of the instruction + 1) is stored into register RA. Next, the PC is loaded with the value of register AT, and the computer resumes execution at the new PC.

8.3.13 HALT

Assembler Syntax

```
HALT
```

Encoding



Description

The machine is brought to a halt and executes no further instructions.