Preface

This manual is intended as an introduction to the REX boards equipped with WRAMP processors in Lab 1. It's purpose is to get you started in programming for the WRAMP processor, and using the REX boards. It covers the basics of WRAMP assembly language, the WRAMP tools, as well as the I/O devices supplied on the REX boards.

It is designed to complement the lecture material of the COMP201 course.

Although every effort has been made to make this manual accurate, it is possible that there may be errors in it. If you find any please report them to *contact-comp201@cs.waikato.ac.nz*

This manual is mainly the product of a lot of hard work by Matt Jervis and Jamie Curtis. Other people that have contributed to it's content are Murray Pearson, Tony McGregor, and Dean Armstrong.

Contents

Chapter 1

Introduction

1.1 Introducing REX and WRAMP

During this year you will be using the REX boards in Lab 1. They should help you gain a practical understanding of the internal workings of a computer. They were developed at this university by Dean Armstrong, as part of an undergraduate degree.

1.1.1 The WRAMP CPU

The ***Central Processing Unit*** (***CPU***) is responsible for carrying out instructions. CPUs have developed at an astounding rate. From the humble Z80 to the 386, and beyond the GHz barrier. But as the speed increases, so does the complexity. Due to this complexity it is difficult learning to program at a low level.

The WRAMP CPU was designed to solve this problem. It is much easier to understand because it does not have the quirks of performance orientated CPUs. Some features of the WRAMP CPU are:

- An easy to understand instruction set. All instructions conform to a clear and consistent structure.
- All operations are carried out on 32 bit words.
- Only a single instruction is executed at a time.
- Memory addressing is easy to understand.

1.1.2 The REX board

For a CPU to be usable a motherboard is needed. The REX board was developed to host the WRAMP CPU. The REX boards can be found mounted on the walls of Lab 1. Each is connected to the Linux machine below it and the terminal above it.

The REX board is made up of the following components:

CPU The WRAMP processor is the brain of the REX board.

RAM Capable of storing 128K words, the RAM resides at addresses 0x00000 to 0x1FFFF. The last 2K words are reserved for the monitor program. The rest of the RAM is used to store and execute your programs, which, by default, are loaded starting from address 0x00000.

ROM Capable of storing 256K words, the ROM resides at addresses 0x80000 to 0xBFFFF. It is used to store the *WRAMPmon* monitor program.

Serial Interface Allows the WRAMP processor to interface with two serial ports. One is connected the Linux machine next to it. The other connects to the terminal on the shelf above. The terminal will be used in later exercises.

Parallel Interface Allows the WRAMP processor to interface with:

- a set of eight switches from which the CPU can read
- two pushbuttons from which the CPU can read
- two seven segment displays to which the CPU can write

Timer Counts down from a certain value and can interrupt the CPU when finished.

Control Panel Provides the user with the ability to control execution of the WRAMP CPU. It also has an LCD display showing the current status of the CPU.

Use of these I/O devices will be covered in chapter 3.

1.1.3 WRAMPmon

The REX board has installed on it monitor software known as *WRAMPmon*. It communicates with the user through the Linux machine connected to the serial port. *WRAMPmon* is used to upload your programs to the REX board and to debug them. This will be covered in more detail in section 1.3.

1.2 Introduction to the WRAMP Assembly Language

To help you get started we will begin by analysing, assembling, executing and debugging a simple WRAMP assembly language program. Using *emacs* (or your preferred text editor) enter the program listed in Figure 1.1. Save it as *intro.s*

This program first displays a message asking you to enter four numbers. It is then meant to input four numbers, add them up and display the result. If you have noticed the bug in it, don't worry as we will correct this later.

As you read through the source file look for the following:

comments are lines beginning with a hash (`#`). These work in the same way as `//` comments in C and are ignored by the assembler.

labels such as `main:` are used to refer to a location in the memory. This may be used to move execution to a different area of code, or to refer to data in the memory.

directives are commands beginning with a period (`.`). They pass information to the assembler.

subroutine calls are used in this program to perform hardware operations such as reading from the serial port. For example the lines `jal putstr` and `jal readnum` are subroutine calls.

WRAMP instructions such as `add $4, $0, $0` get converted to machine code by the assembler. They can then be executed by the WRAMP CPU.

```
.text
.global main
main:
     # Get the address of the welcome message
     la     $2, welcome_msg
     # Display the message
     jal    putstr
     # Clear our sum register
     add    $4, $0, $0
     # Initialise the loop counter
     addi   $5, $0, 1
loop:
     # Read a number from the user
     jal    readnum
     # Add it to our running total
     add    $4, $4, $1
     # Increment our loop counter
     addi   $5, $5, 1
     # Test to see if we have done all 4 numbers
     slti   $1, $5, 4
     # Keep looping until 4 numbers have been entered
     bnez   $1, loop
     # Get the output message
     la     $2, output_msg
     # Print it
     jal    putstr
     # Move our sum into register $2 to display it
     add    $2, $0, $4
     # Print out the number
     jal    writenum
     # Return to the monitor
     j      exit
.data
     # This is our welcome message
welcome_msg:
     .asciiz  "Welcome to the world of WRAMP!!!\n\nPlease
              type four numbers, pressing enter after each:\n"
     # This is the output message
output_msg:
     .asciiz    "The sum of the numbers is : "
```

Figure 1.1: WRAMP Assembly Language Program

1.2.1 Labels

A label is a means of referencing a location in memory. Rather than specifying absolute addresses when accessing data in memory, a label can be used. This makes it much easier for the programmer to write and understand the code. Labels can also be declared within the code. They can then be used as a destination for branch and jump instructions.

A label declaration consists of a name (containing no spaces) followed by a colon (`:`).

The example code in figure 1.1 has labels `main`, `loop`, `welcome_msg`, `output_msg`. The labels `main` and `loop` refer to locations within the code. The labels `welcome_msg` and `output_msg` refer to data.

1.2.2 Directives

Directives (commands preceeded by a `.`) do not become part of the final executable. They are used to pass information to the assembler, like `#` directives do in C.

`.text` , `.data` and `.bss`

A program written in WRAMP assembly can be split into three sections. These are used to seperate executable code, initialised data, and uninitialised data. The directives `.text` , `.data` and `.bss` are used to do this. When the assembler encounters one of these directives it knows that all the code following it belongs in the given section.

The `.text` section contains WRAMP assembly instructions which will be converted to machine language which can be executed on the WRAMP CPU.

Within the `.data` section, space in the memory can be reserved and initialised. This is useful for strings, constants, and variables which have an initial value.

The `.bss` section allows memory space to be reserved but not initialised. The advantage of this is that space can be reserved in chunks using a single command. This feature makes the `.bss` section useful for arrays.

Assigning Space

Within the `.data` and `.bss` sections memory space is assigned using directives. To be able to access this you need to place a label before the assigning directive. In the example code, the label `welcome_msg:` refers to the memory reserved by the `.asciiz` directive.

```
welcome_msg:
    .asciiz "Welcome to the world of WRAMP!!!\n\n"
```

Some of the memory assigning directives you may come across:

`.word n` This assigns one word of memory space and initialises it to the number `n`.

`.asciiz "str"` reserves and initialises space for a NULL terminated ASCII string (`"str"`). This means that the string is followed immediatly by a NULL character (`0`). The NULL character can then be used to identify the end of the string.

`.ascii "str"` reserves and initialises space for the ASCII string `"str"` without NULL terminating.

`.space n` is used to allocate a chunk of space of size `n` in the `.bss` section.

The `.ascii` and `.asciiz` directives can not be used in the `.bss` section as they initialise the space they reserve. The `.word` directive may be used as long as it is not provided with an argument.

The `.global` Directive

When linking multiple files, we need to share functions and data. However we do not want to expose all the labels in a file. Only labels declared as global (using the `.global` directive) are accessible outside the current file.

In the example program we have declared the 'entry point' `main` as global. To do this we used the following directive:

```
.global main
```

1.2.3 Registers

Registers are the CPU's equivalent of variables. There are 16 general purpose registers numbered 0 to 15. They can be used for temporarily storing data while it is used in operations. Registers are generally referred to by a $-sign followed by a number. For example `$0` refers to register zero. The contents of registers can be transferred to and from main memory. This done using load and store instructions which are covered in a later section.

Some of the general purpose registers have special uses:

Register `$0` is always zero. Any attempts to write to it are ignored. This provides a constant source of zero that can be used for comparing and initialising registers.

Register fourteen is denoted `$sp`. This register is defined by convention to be the stack pointer. While the hardware imposes no special conditions on this register, failure to follow this convention may affect the ability of code to interoperate with other software.

Register fifteen is denoted `$ra`. It is defined by convention to be the subroutine return address register. When a jump and link instruction is executed this register is loaded with the address of the next instruction after the jump and link. We will discuss stacks and this register in a later chapter.

In addition to these general purpose registers is a set of special purpose registers. The use of these is covered in chapter 4.

Various types of instructions are introduced below.

1.2.4 Instructions

Executable machine code is create by assembling WRAMP instructions. Instructions tell the WRAMP CPU what to do. For example an `add` instruction will add the contents of two registers and place the result in another.

Arithmetic

Arithmetic instructions come in four forms, which are listed below, using `add` as an example:

`add` R_d, R_s, R_t
Simply performs the specified operation, on R_s and R_t, placing the result in R_d. For example `add $1, $2, $0` will add the contents of registers `$2` and `$0` then place the result in `$1`.

`addi` R_d, R_s, `Immediate`
This is known as immediate form. The argument `Immediate` is a constant. Thus the specified operation is performed on R_s and `Immediate`, and the result placed in R_d. For example `addi $1, $2, 4` will add 4 to the contents of `$2` and place the result in `$1`.

`addu` R_d, R_s, R_d
The *u* implies that this is an unsigned instruction. Thus it is assumed that both the operands are positive. This is important when we consider twos complement representation of negative numbers. For example `1111 1111 1111 1111 1111 1111 1111 1111` represents `-1` when treated as signed, and 0xFFFFFFFF, a very large positive number, when treated as unsigned.

`addui` R_d, R_s, `Immediate`
This is the combination of the two forms above. It performs an unsigned operation on R_s and the `Immediate` value.

For a full listing of arithmetic instructions see Appendix A.

Memory I/O

The WRAMP CPU has a 20 bit memory space. Of this the memory locations 0x00000 to 0x1FFFF are RAM.

Unlike other CPUs the WRAMP has only a single method of referencing external memory. It consists of a base address added to an offset. The base address is specified as a constant in the instruction. The offset is the contents of a specified register.

The notation for this is `base(`R_s`)`. The `base` can either be a label or an integer. To load from the memory location specified by a label, `$0` can be used for R_s. For the first few chapters we will use this method of referencing the memory. Figure 1.2 shows an example. It loads a word of data from memory into a register, adds 1 to it, then stores it back into the memory.

`lw` R_d, `base(`R_s`)`
Used to get the contents of the specified memory location and place it in the register R_d.

`sw` R_d, `base(`R_s`)`
Will place the contents of R_d into the specified memory location.

Sometimes you will need to know the address of a variable. The instruction 'load address' (`la`) is used for this. It has the structure:

`la` R_d, `label`
Load the address that `label` refers to into register R_d.

Set Instructions

These instructions are used to compare either two registers, or a register and an immediate value. They take the same form as arithmetic instructions. If the test passes the destination register is set to `1`, otherwise it is cleared to `0`.

For example to test if registers `$2` and `$3` were equal and store the result in `$1` we would use the 'set if equal' instruction: `seq $1, $2, $3`.

A full listing of test instructions can be found in the instruction set reference, Appendix A.

Program Control

We move execution to a different part of the program using two types of instruction: the unconditional jump, and the conditional branch.

```
.text
     . . .

     #Read from the memory location ‘counter’ into $4
     lw $4, counter($0)

     #Add 1 to $4
     addi $4, $4, 1

     #Store the contents of $4 into the memory location ‘counter’
     sw $4, counter($0)

     . . .

.data
     # This is our counter
counter:
     .word 0
```

Figure 1.2: Memory I/O Example

Jump Instructions
The jump instructions simply move execution to a different line of code. They are unconditional and so do not perform any tests first. There are four different varieties:

`j label` - Jump
Jumps to the specified label.

`jal label` - Jump and link
Stores the address of the next instruction into the return address register `$ra`. It then jumps to the specied label. This is used when calling subroutines.

`jr` R_s - Jump Register
Jumps to the location specified by the contents of the register R_s. Because the WRAMP CPU has only a 20 bit address space the upper 12 bits of the register are ignored. Jump register is typically used to return from a subroutine using `jr $ra`

`jalr` R_s - Jump and Link Register
A combination of `jal` and `jr`. It works in the same way as `jr` but it stores the address of the next instruction into the return address register before jumping.

Branch Instructions
Branching instructions are conditional, they look at whether a register is zero or not, and act on that information. There are two varieties: ‘branch if equal to zero’ `beqz` and ‘branch if not equal to zero’ `bnez`. Both of these instructions take a single register and a label as arguments.

For example the ‘branch if not equal to zero’ command ‘`bnez $1, loop`’ will branch to label `loop` if `$1` is non-zero.

1.2.5 Assembling and Linking

Before we can execute our program on the REX board we need to translate it into machine code, so that the WRAMP CPU can understand it. This is a two step process.

Assembly

We firstly have to ***assemble*** the source code using a program called an *assembler*. This involves checking that all the directives and instructions within the program make sense (are syntactically correct). The program is then translated into a file called an ***object file***. This contains portions of machine code along with other information about the program.

To assemble our file we use the WRAMP ***assembler***, *wasm*:

```
wasm intro.s
```

There should now exist a file called *intro.o*. This is the ***object file***.

Linking

Next we must ***link*** the ***object file***. A program may be comprised of many separate parts. Even our example program contains functions (subroutines) contained in a separate ***library***. Linking involves joining all these parts together to create a final program.

The WRAMP ***linker*** is called *wlink*. We will use it to link our file to the library `lib_ex2.o` in the 201 home directory. This file ***does not*** need to be copied into your home directory. Issue the command:

```
wlink -o intro.srec intro.o /home/201/ex2/lib_ex2.o
```

This command links together the object files to create file called *intro.srec*. This file is in a form known as S-Record, which is suitable to be uploaded to the REX board and executed.

1.3 Introduction to WRAMPmon

The *WRAMPmon* monitor is a program which runs on the REX boards. It provides you with basic facilities for interacting with the REX board. It is important for hardware and software development, debugging, testing and troubleshooting. It uses a command line interface to perform these functions.

WRAMPmon communicates through the serial port that is connected to the Linux machine. By running a terminal program on the Linux machine we can talk to the REX board.

1.3.1 WRAMPmon commands

A set of commands is provided for the user to interact with *WRAMPmon* including:

- a help command (`help`, or `?`)
- commands to view and alter the memory and register contents
- a command to upload your programs
- commands to execute programs in the memory
- commands for debugging

For a listing of these commands see Appendix C.

1.3.2 Getting started

The REX board uses serial ports as its primary form of communication. The main serial port on the REX board is connected to a Linux machine. The Linux machine runs a terminal program, called *remote*, which transmits anything typed on the keyboard directly to the REX board, and displays any text received from REX.

To execute the terminal program type `remote` from a console. Once you have `remote` running you will need to reset the REX board by pressing the red `RESET` button. You should see something like Figure 1.3.

```
+-----------------------------------------------+
|                 WRAMPmon 0.6                  |
| Copyright 2002, 2003 The University of Waikato |
|                                               |
|          Written by Dean Armstrong            |
+-----------------------------------------------+

Type ? and press enter for available commands.

 >
```

Figure 1.3: *WRAMPmon*

Type `?` to obtain a list of commands available within *WRAMPmon*.

Help on individual commands can be obtained by typing the command and then a question mark, eg:

```
load ?
```

Each command gives the required format and available options. An item enclosed in square brackets, `[` and `]`, indicates an optional item. An item enclosed in angled brackets, `<` and `>`, indicates a compulsory item.

If you have any problems with the monitor not responding at any stage you should always be able to get back to the board's initial state with the start-up message displayed by pressing the red `RESET` button on the REX board.

1.3.3 Uploading and Executing your Program

Before you can run your program on the REX board it must first be loaded into the board's memory. The easiest way to do this is to set the board into load mode and use the upload option within the *remote* program to send the executable file from the Linux machine to the board. To place the board into load mode, type the following command (then press enter) at the *WRAMPmon* prompt:

```
load
```

After hitting return there should be a reminder response from the board indicating how to send an executable file. To send the executable type:

`<ctrl>-a` and then `s`

Where `<ctrl>-a` means press and hold the `control` key and hit the `a` key and then release both keys. The *remote* program does not pass this key sequence on to the REX board but instead enters a command mode. When you press `s`, remote will switch into upload mode. A dialog box appears which asks you to enter the name of the file you wish to upload. Type in the following file name:

```
intro.srec
```

After the file name has been entered a series of dots should appear on the screen indicating that the file is being uploaded, followed by a message which tells you that is has completed. You should then press the enter key to leave the upload mode of `remote` at which stage the *WRAMPmon* prompt should reappear.

Now that the program has been uploaded, you can run it by typing the command:

```
go
```

This will start executing your program from the entry point defined by the `main` label. The program will prompt you to enter four numbers. You should continue to type a number and hit enter until the program outputs the resulting sum. The output of your program should appear on your screen. When the *WRAMPmon* prompt appears again it indicates that your program has finished executing.

1.3.4 Debugging your Program

The program that you have just run on the REX board should have read four numbers from the keyboard and output their sum to the terminal. However, when run, only three numbers are read before their sum is output.

In this section of the exercise you are going to use the debugging features of *WRAMPmon* to locate the bug that causes this problem.

The debugging commands within *WRAMPmon* allow you to trace the execution of a program (ie. follow the path of execution) by inserting breakpoints.

If a breakpoint is set at a certain instruction, then when the program is running, and that instruction is about to be executed, control will be returned to the monitor. From here, you can view register and memory contents, and maybe resume execution, or step through the program one instruction at a time to try and identify bugs. The commands for setting, removing and viewing breakpoints are `sb`, `rb`, and `vb` respectively. The help facilty in *WRAMPmon* fully describes the operation of these commands. For example to find out more about the set breakpoint command type

```
sb ?
```

Insert a breakpoint at the beginning of the loop in the above program (ie. at memory location `0x00004`), and execute the program using the `go` command. When a break does occur, use the view registers command to find out the contents of the registers. You should also notice that the breakpoint register dump contains identical information to the `vr` command.

To continue execution type:

```
cont
```

This will cause the program to continue execution until either the breakpoint is encountered again or the program completes execution. Remember that the program will now wait for you to enter a number before it will break again.

Another feature of the monitor that can be used to help debug you programs is the single-step command `s`. This command causes only the immediately next instruction to be executed and then returns back to *WRAMPmon*.

To demonstrate the use of the single step command you are to reload the program. Create a new breakpoint at memory location `0x00005`. Once this is done run the program using the `go` command.

Once you have encountered the breakpoint try single stepping through the program using the `s` command. Pay careful attention to the loop counter in `$5` and the instructions that increment and test it. As we are only interested in the loop counter at this stage we will skip the `readnum` subroutine. We do this using the 'step over' `so` command. When you reach the instruction `jal readnum` instead of typing '`s`' type '`so`'.

At some point you may wish to remove the breakpoint using the `rb` command and then use `cont` to make the program run to completion.

As you may have realised the program is designed with a bug. Although it is intended to add 4 numbers it will only add 3. You should now use the debugging methods we have just covered to determine the bug. Then correct it and assemble, link and load the program to confirm it operates correctly.

Chapter 2

Stack Guide

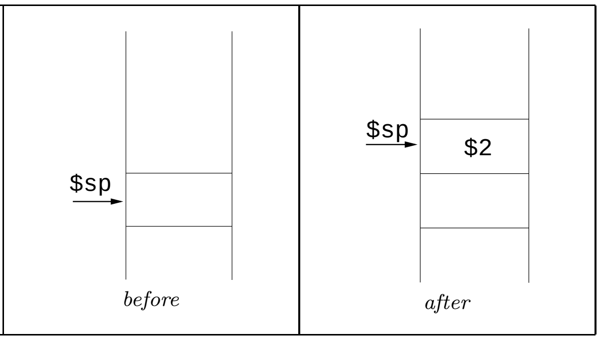
2.1 WRAMP Stack Frame

A stack is a term used to describe a 'first in, last out' buffer. New items are placed on the top of the stack, and must be removed before older items. Two terms are commonly used to refer to the operations of adding to, and removing values from a stack. A push operation puts a new value at the very top of the stack, a pop operation removes the item from the top of the stack. These are the only operations allowed to be performed on a stack, there is no way to remove an older item before a newer one. Stacks provide an ideal mechanism for passing parameters to a function and providing storage for local variables and temporary results inside a function. This document describes the reasons a stack is used on the WRAMP architecture, how the stack is created, and the conventions surrounding its use.

The WRAMP processor itself does not directly support a stack, however it is possible to setup a stack in software. To achieve this, a block of memory for the stack to reside in must be set aside. On the REX board the monitor does this for you. The stack starts at the top of memory, with new items placed at lower memory addresses. Because of this, stacks on the WRAMP architecture are often referred to as growing downwards.

```
push:
      # move the stack pointer down
      # by one to create space for
      # a value to be pushed onto
      # the stack.

      subui $sp, $sp, 1

      # store the contents of $2 onto
      # the new space on the stack

      sw $2, 0($sp)
```

(a) WRAMP code



(b) Stack Diagrams

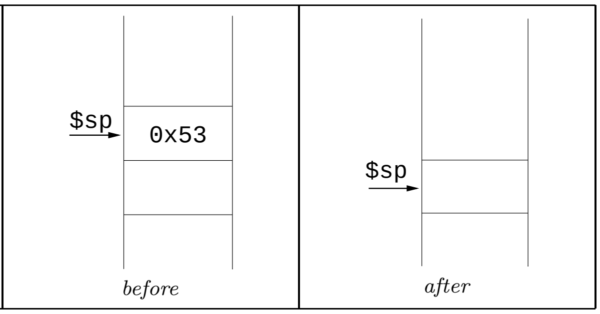
Figure 2.1: Push

To place new items onto and remove existing items from the stack you need a way to know the current address of the top of the stack. To enable this a register is set aside to store this address. This register is referred to as the "top of stack" pointer, or more often just the "stack pointer". By convention the stack pointer is stored in register number 14 in the WRAMP architecture and is referred to as `$sp` in WRAMP assembly code. Figure 2.1(a) shows example WRAMP code to "push" a new value onto the stack and

(b) shows the stack before and after the push operation. Figure 2.2 shows the WRAMP code and stack diagrams for a pop operation.

```
pop:
      # pop into register 3 the value
      # stored on the top of the
      # stack

      lw $3, 0($sp)

      # Move the stack pointer up
      # by one to remove item

      addui $sp, $sp, 1
```

(a) WRAMP code



(b) Stack Diagrams

Figure 2.2: Pop

2.2 WRAMP Stack Conventions

On the WRAMP architecture the stack is used to:

- store local variables that are not stored in registers
- temporarily store the contents of registers so that a subroutine can use them while making sure the previous contents are preserved.
- pass parameters to a subroutine

```
parent:
       addi $3, $0, 5
loop:
       beqz $3, endloop
        ...
       jal child
        ...
       subi $3, $3, 1
       j loop

endloop:
       j exit

child:
        ...
       add $3, $4, $5
        ...
       jr $ra
```

Figure 2.3: Incorrect Function

2.3 Saving Registers

When a program contains a number of subroutines that can call each other, a set of conventions is required to ensure that a subroutine does not use a register to modify values that a parent subroutine is also using. For example consider the code sequence in Figure 2.3. Notice that the section of code labelled `parent` is using `$3` as a loop counter that decrements each time through the loop. Inside this loop is a call to the subroutine `child` that uses `$3` to store an intermediate result. This would overwrite the loop counter value stored in that register by the subroutine `parent`. While it would be possible in this simple sequence to rearrange the code to fix the problem, it will not always be possible to do so. To ensure problems like this do not occur in code there needs to be a set of conventions controlling the way registers are used.

The convention used in the WRAMP architecture is that all subroutines must save the contents of a register to the stack before it can use it. The value must then be restored from the stack before the subroutine exits. It should be noted that it is up to the programmer to ensure these conventions are followed and the processor does not enforce them in any way. For code generated by a C compiler the compiler must ensure that these same conventions are followed. Figure 2.4 shows the corrected program that follows the conventions.

When a subroutine is called using the `jal` instruction, the return address for the subroutine is placed in register 15 (`$ra`). If this subroutine then uses `jal` to call another subroutine it will overwrite its own return address. Because of this, any subroutine that is going to call another subroutine needs to save `$ra` onto the stack before calling the routine and restore it before it returns. Figure 2.5 gives an example of this.

```
parent:
       addi  $3, $0, 5
loop:
       beqz  $3, endloop
        ...
       jal   child
        ...
       subi  $3, $3, 1
       j     loop

endloop:
       j exit

child:
        ...
       # save register 3 before we overwrite
       # the contents of it
       subui $sp, $sp, 1
       sw    $3, 0($sp)

       add   $3, $4, $5
        ...
       # restore the old contents of register
       # 3 before we return
       lw    $3, 0($sp)
       addui $sp, $sp, 1

       jr    $ra
```

Figure 2.4: Correct Function

There is one exception to the rule that all registers must be saved. For reasons discussed in the next section, register 1 (`$1`) never needs to be saved or restored.

2.4 Parameter Passing

In Chapter 1, parameters were passed to subroutines using regsiters. While this works in this simple case consider what would happen if a subroutine required a large number of parameters or called other subroutines. It is not difficult to see that with a large program it would not take long to exhaust the registers available to the programmer on the WRAMP processor.

```
child:
        # save the return address before we
        # call our subroutine
        subui $sp, $sp, 1
        sw $ra, 0($sp)

        jal my_child
         ...

        # get our return address back off of the
        # stack so we can return there.
        lw $ra, 0($sp)
        addui $sp, $sp, 1

        jr $ra
```

Figure 2.5: Calling a Function

A convention needs to be defined so that a subroutine knows how to find the parameters it has been passed, and knows how to pass parameters to subroutines it calls.

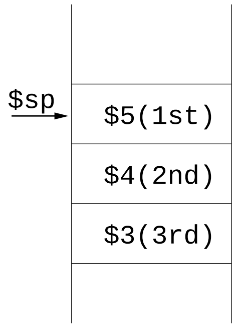
On the WRAMP processor the convention is to pass all parameters to a subroutine using the stack. Before a subroutine is called all of the parameters that are going to be passed to it must first be pushed onto the stack. Parameters appear on the very top of the stack when a subroutine is entered. If code is being generated for a C function call, then the convention is to push the parameters onto the stack in the reverse order so that the first C parameter ends up on the top of the stack just before the function is called. Figure 2.6(a) shows a C function call, (b) WRAMP code to implement it and (c) a diagram of the stack at the time of the function call.

```
cat(first, second, third);
```

(a) C code

```
# push the third parameter on
subui $sp, $sp, 1
sw $3, 0($sp)

# push the second parameter on
subui $sp, $sp, 1
sw $4, 0($sp)

# push the first parameter on
subui $sp, $sp, 1
sw $5, 0($sp)

jal cat

# remove all three parameters
addui $sp, $sp, 3
```

(b) WRAMP code



(c) Stack before call

Figure 2.6: Passing Parameters to a Function

As you will notice in the previous example a significant proportion of the WRAMP code is associated with manipulating the stack pointer. An alternative and more efficient approach is to calculate the maximum size that a stack will grow to in a function and pre-allocate this space as the function is entered. Just before this "parent" function calls another function it copies the parameters to the appropriate place in the pre-allocated space. If the function accepts one parameter, that parameter must be placed at the top of the stack. If a function accepts two parameters the first parameter must be placed at the top of the stack and the second placed beneath it. If you think of the parameters as a list numbered from zero, the position of any parameter can be calculated as follows:

$$Address = \$sp + ParameterNumber$$

For example if a function `cat(first, second, third)` is going to be called then first will be placed at `$sp` + 0, second at `$sp` + 1 and third at `$sp` + 2. Figure 2.7(a) shows an example C code sequence containing two function calls (one with a single parameter and one with three parameters) and Figure 2.7(b) shows the WRAMP assembler code for this sequence.

Once a subroutine has been called it then has to retrieve these parameters off of the stack so that it can use them. This requires a number of loads from the current stack. The important part to notice is that these are not pop operations, as they do not reduce the size of the stack. The function simply is looking into the stack of the function that called it to discover the parameters it has been called with. A function must ***never*** return with the stack pointer pointing to a different location to when the function was called.

A function also needs to be able to return a value to its parent. Traditional languages only ever allow a function to return a single value, therefore the use of the stack to return a value is probably over complex. On the WRAMP architecture, values are returned to the parent in register 1 (`$1`). Because of this fact `$1` is the only exception to the rule that all registers must be returned with their original contents when a function returns. A function is actually allowed to change the contents of `$1` even if it doesn't return any value to its parent. Figure 2.8 shows a simple maximum function that is passed two parameters and returns the larger of the two. As this function is a leaf function (i.e. calls no other functions) it need not save the contents of it's return address.

```
 ...
dog(last);
 ...
cat(first, second, third);
 ...
```

(a) C code

```
# allocate the amount of space on the stack
# to allow for the call with the largest number
# or parameters. (in this case 3)
subui $sp, $sp, 3
 ...

# The parameter 'last' must be placed on the stack
# and is currently being stored in $6.
sw $6, 0($sp)

# Call the function
jal dog
 ...

# Put the parameters for cat onto the stack
sw $3, 0($sp)    # 'first'
sw $4, 1($sp)    # 'second'
sw $5, 2(sp)     # 'third'

# Call the function
jal cat
 ...

# remove the space from the stack.
addui $sp, $sp, 3
```

(b) WRAMP code

Figure 2.7: Calling multiple functions

2.5 Local Variables

So far we have kept all local variables, such as temporary storage, loop counters etc. in registers. As there are a small number of registers it would be a major limit to a language to enforce that it could have no more local variables than the architecture has registers. To overcome this limit, the stack is set up so that local variables can be stored on the stack and only be loaded into registers temporarily as required. A code segment is shown in Figure 2.10(a). Figure 2.10(b) provides an example of how the WRAMP code would look if the local variables are kept on the stack. As you can see, even in this small piece of code there is a large proportion of the code dealing with fetching and storing the variables to and from the stack. If you are writing C code it is the job of a compiler to optimise these areas of the assembly code and reduce to a minimum the number of these load and stores.

2.6 The Stack Frame

All of the discussion so far has been treating the uses of the stack as separate concepts. In reality all of these are used by C functions to create a concept called the stack frame. The stack frame is an area on the top of the stack that has a standard format. Inside this block is space for local variables, register save and parameter passing for the current function. We precalculate the size of the stack frame by summing the sizes of each of the areas. We need space for one item on the stack for every local variable, one item for each register we save, and one item for each parameter of the function with the largest number of parameters that we call.

For example if we need to setup a stack frame for a function that needs to store three local variables and save two registers and calls no other function we will need a stack frame of size 5.

If we have a non-leaf function that needs 3 local variables, save 4 registers and the return address, and calls two functions, one of which takes 2 parameters and the other takes 4 parameters, we will need a stack frame of size 12.

The layout for a complete stack frame is shown in Figure 2.9

Figure 2.11 shows a non-leaf function that calls two subroutines. The function has zero local variables stored on the stack, but is a fully compliant function. It sets up a stack frame on entry and tears it down on exit. It is strongly suggested that you walk through this code and draw a diagram of the stack frame that this function creates. Any functions you write that needs to be compliant should contain very similar entry and exit code to this function.

The function is a successive addition multiplication system. It uses a function called `add` to add the two numbers. At the end of the function it displays the result to the seven segment display using the `writessd` function.

```
parent:
       addi $3, $0, 5
loop:
       beqz $3 endloop
        ...
       jal child
        ...
       subi $3, $3, 1
       j loop

endloop:
       j exit

child:
        ...
       # save register 3 before we overwrite
       # the contents of it
       subui $sp, $sp, 1
       sw $3, 0($sp)

       add $3, $4, $5
        ...
       # restore the old contents of register
       # 3 before we return
       lw $3, 0($sp)
       addui $sp, $sp, 1

       jr $ra
```

Figure 2.8: Example

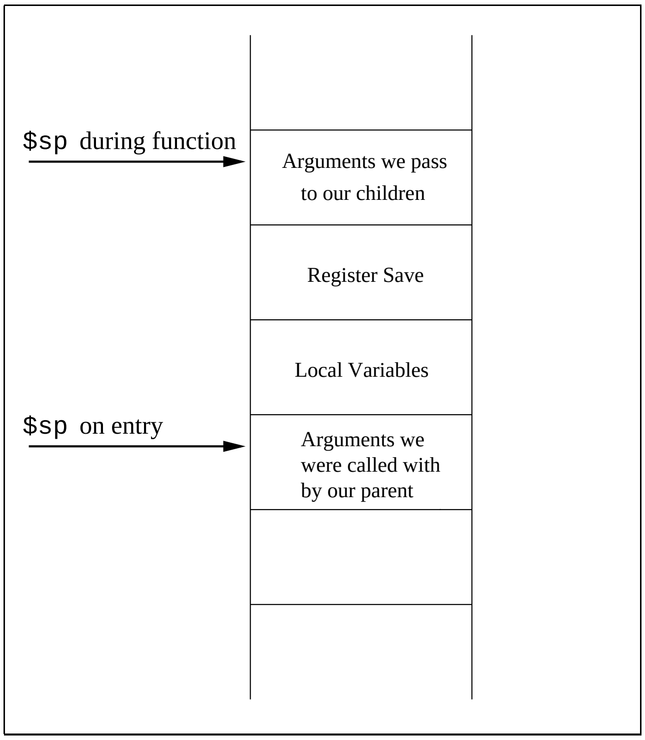


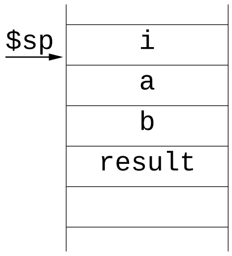
Figure 2.9: The Stack Frame

```c
int i;
int a = 2;
int b = 3;
int result = 0;

for(i = 0; i < a; i++){

   result = result + b;

}

 ...
```

(a) C code

```
func:
       # Allocate space for 4 locals
       subui $sp, $sp, 4

       # Initialise a = 2
       addi $2, $0, 2
       sw $2, 1($sp)

       # Initialise b = 3
       addi $2, $0, 3
       sw $2, 2($sp)

       # Initialise result = 0
       sw $0, 3($sp)

       # Initialise i = 0
       sw $0, 0($sp)

loop:
       # Do the for loop test
       lw $2, 0($sp)  # get i
       lw $3, 1($sp)  # get a
       slt $4, $2, $3
       beqz $4, end

       # perform the loop
       lw $2, 3($sp)  # get result
       lw $3, 2($sp)  # get b
       add $2, $2, $3
       sw $2, 3($sp)  # put result

       # increment i
       lw $2, 0($sp)  # get i
       addi $2, $2, 1
       sw $2, 0($sp)  # put i

       j loop

end:
        ...
```

(b) WRAMP code



(c) Stack

Figure 2.10: Using Variables

```
multiply:
        # Setup a stack frame (2 parameters, 5 registers to be saved)
        subui   $sp, $sp, 7
        # Save some registers for us to use
        sw      $6, 2($sp)
        sw      $7, 3($sp)
        sw      $8, 4($sp)
        sw      $9, 5($sp)
        # This is a non-leaf function so we must save the return address
        sw      $ra, 6($sp)

        # Initialise the 'result' variable to zero
        addu    $7, $0, $0
        # Initialise our loop counter
        addu    $6, $0, $0
        # Get our first parameter into $8
        lw      $8, 7($sp)
        # Get our second parameter into $9
        lw      $9, 8($sp)

loop:
        # Use our 1st parameter to control how many times we add our
        # second parameter to itself
        slt     $1, $6, $8
        beqz    $1, exit_loop

        # The first parameter to add is the existing 'result'
        sw      $7, 0($sp)

        # The second parameter we pass is the same as our 2nd parameter
        sw      $9, 1($sp)
        jal     add

        # Save the return value from add back into our 'result' variable
        addu    $7, $0, $1

        # Increment our loop counter
        addui   $6, $6, 1

        j       loop

exit_loop:
        # Write the result to the seven segment display
        sw      $7, 0($sp)
        jal     writessd

        # Return our result to our parent
        addu    $1, $0, $7

        # Restore all the registers we used
        lw      $6, 2($sp)
        lw      $7, 3($sp)
        lw      $8, 4($sp)
        lw      $9, 5($sp)

        # Get our return address back
        lw      $ra, 6($sp)

        # Destroy our stack frame
        addui   $sp, $sp, 7
        # Return
        jr      $ra
```

Figure 2.11: Example

Chapter 3

REX I/O Devices

3.1 Introduction

The REX board provides a number of I/O devices. This document describes the devices and the way in which WRAMP code can interact with them. There are three major I/O devices, a dual serial port, a parallel port and a programmable timer.

All REX I/O devices are memory mapped. This means that to access a device, WRAMP code simply reads or writes to special memory locations using the standard load word (`lw`), and store word (`sw`) instructions.

The base memory addresses of all of the devices are provided in Table 3.1. The details of how to use each of the devices forms the body of this document.

Device	**Base Address**
First Serial Port	0x70000
Second Serial Port	0x71000
Timer Base	0x72000
Parallel Port	0x73000

Table 3.1: I/O Device Base Addresses

3.2 Serial Devices

The REX board provides two independent RS232 serial interfaces.

For the Computer Systems course each of these ports are attached to separate display devices. The first serial port is attached to the Linux machine. This port is used by the monitor software on the REX board to communicate with the user and to allow software to be uploaded to the REX board. The second serial port is attached to a Digital VT320 dumb serial terminal.

The programmers view of a serial interface consists of five registers. The names of these registers and their addresses, expressed as offsets from the base address, are provided in Table 3.2. The base address for the first serial port is `0x70000` and the base address for the second serial port is `0x71000`.

The serial ports provided on the REX board can operate in either polled or interrupt driven I/O modes. Interrupt I/O will be disabled by default.

Register name	Offset
Serial Transmit Data Register	0
Serial Receive Data Register	1
Serial Control Register	2
Serial Status Register	3
Serial Interrupt Acknowledge Register	4

Table 3.2: Serial Port Register Offsets

3.2.1 Serial Transmit Data Register

The Transmit Data Register (TDR) is a write-only register. A character will be transmitted by writing the value into this register. The serial port status register indicates if a value is permitted to be written to this register. If a character is written to this register without first checking the status register it will be possible to lose characters. If transmit data sent interrupts are enabled, an interrupt will be triggered when this register becomes empty indicating that another character can now be sent. Some example WRAMP code to transmit a single character is given in Figure 3.1.

```
         . . .
        # Put the character we want to send in $9
        addi $9, $0, 'A'

check:
        # Get the first serial port status
        lw   $11, 0x70003($0)
        # Check if the TDS bit is set
        andi $11, $11, 0x2
        # If not, loop and try again
        beqz $11, check
        # Serial port is now ready so
        # transmit character
        sw   $9,  0x70000($0)
         . . .
```

Figure 3.1: Simple Transmit Code

3.2.2 Serial Receive Data Register

The Receive Data Register (RDR) is a read-only register. When a character is received from the serial line, it appears in this register. When a character arrives the status register will reflect this change. If receive data ready interrupts are enabled, an interrupt will be triggered when data arrives in this register. An example of a simple polled receive routine is shown in Figure 3.2.

```
      . . .
check:
      # Get the first serial port status
      lw   $11, 0x70003($0)
      # Check if the RDR bit is set
      andi $11, $11, 0x1
      # If not, loop and try again
      beqz $11, check
      # Serial port now has a character.
      # Get it into $9
      lw   $9,  0x70001($0)
       . . .
```

Figure 3.2: Simple Receive Code

3.2.3 Serial Control Register

This register allows line parameters such as serial bit rate to be set. The serial ports are configured appropriately by the monitor for the device setup used by the Computer Systems course. Unless you know that you specifically need to change something in this register you should leave it as default.

The control register also controls when the serial port will cause an interrupt. The serial port can selectively cause an interrupt when a character is received into the receive data register, the transmit data register becomes empty or on an error condition. Any combination of these can be enabled or disabled at one time. To enable interrupts to be used by the serial port under specific circumstances a '1' should be written to the appropriate location. If an interrupt has occurred it must be acknowledged by writing into the serial interrupt acknowledge register.

The REX board monitor will initialise the serial port so that no interrupts are enabled

The control register is a read/write register. Writes to this register have an immediate effect on the line settings.

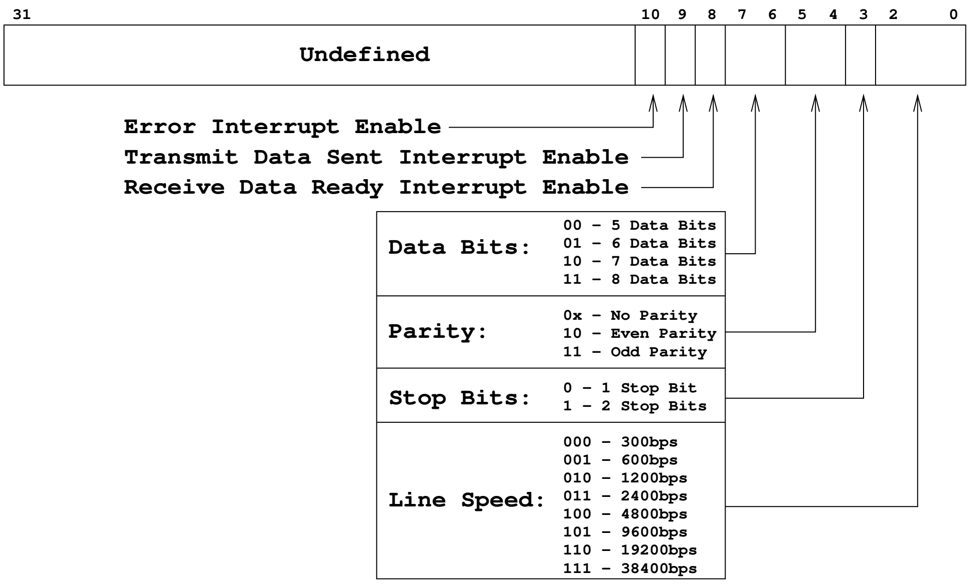


Figure 3.3: The Serial Control Register

eg. To configure a serial interface to operate with no interrupts enabled, at 9600 bits per second, with 8 data bits, no parity and 1 stop bit, the value '`00011000101`' would be written to the control register.

3.2.4 Serial Status Register

The status register is a read-only register, which gives error and status information about the serial interface. It allows the programmer to see if data has been received, sent, or if an error condition is present.

The Transmit Data Sent (TDS) bit will be set to '1' as soon as the transmit data register is empty. Checking that this bit is set allows WRAMP code to ensure that it will not overwrite any data by placing another character into the transmit data register. This bit will automatically be cleared if the transmit data register becomes full.

Similarly the Receive Data Ready bit will be set to '1' as soon as there is new valid data in the receive data register. A read from the receive data register automatically clears this bit.

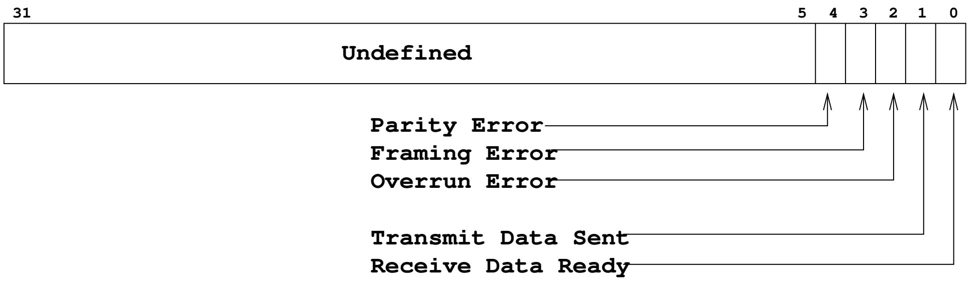


Figure 3.4: The Serial Status Register

eg. If the value '00001' was read from the status register then we could determine that a character has been received without error, and is available in the receive data register.

3.2.5 Serial Interrupt Acknowledge Register

The interrupt acknowledge register is a read/write register. When the serial interface has generated an interrupt this register allows the program to determine the reason for the interrupt as well as acknowledge interrupts that have been dealt with.

To acknowledge an interrupt a zero ('0') should be written over the current status field for the type of interrupt being acknowledged. Most often it will be the desire of the programmer to acknowledge all of the possible serial port interrupts in one instruction. This can be achieved by storing register `$0` to the interrupt acknowledge register.

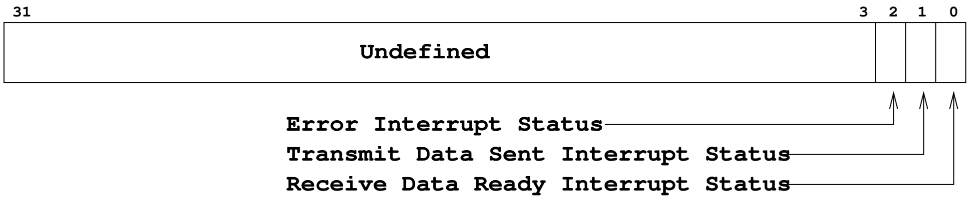


Figure 3.5: The Serial Interrupt Acknowledge Register

eg. If the value '010' was read from the interrupt acknowledge register we could determine that the cause of the interrupt was the transmit data register becoming empty. If the value '000' was written to the interrupt acknowledge register all outstanding serial port interrupts would be acknowledged.

3.3 Parallel Interface

The parallel interface on the REX board provides an input interface from a bank of 8 on-off switches and two momentary push-buttons, as well as an output interface to two LED Seven Segment Displays (SSDs). Parallel interrupts, if enabled, will be generated on any switch or push-button state change.

The programmers view of the parallel interface consists of six registers. The names of these registers and their addresses, expressed as offsets from the base address, are provided in Table 3.3. The base address for the parallel port is `0x73000`.

Register name	**Offset**
Parallel Switch Register	0
Parallel Push Button Register	1
Parallel Left SSD Register	2
Parallel Right SSD Register	3
Parallel Control Register	4
Parallel Interrupt Acknowledge Register	5

Table 3.3: Parallel Port Register Offsets

3.3.1 Parallel Switch Register

The switch register is a read-only register. A read from this register returns a bit pattern with bits set corresponding to the switches that are on.

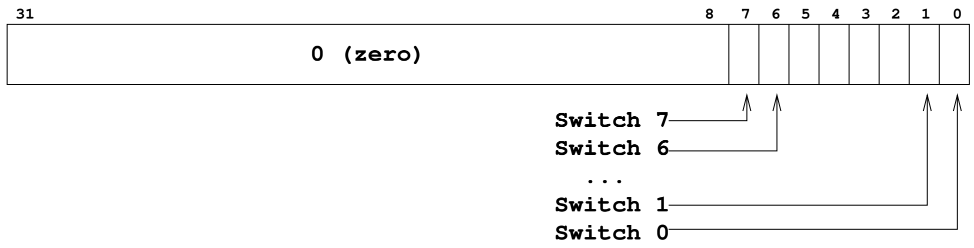


Figure 3.6: The Switch Register

3.3.2 Parallel Push Button Register

The push button register is a read-only register. A read from this register returns a bit pattern in the low order 2 bits corresponding to the push buttons that are currently being depressed.

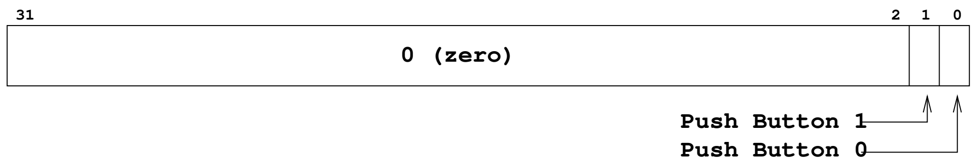


Figure 3.7: The Push Button Register

3.3.3 Parallel Left and Right SSD Registers

The Left and Right SSD Registers are read/write registers. These registers contain the value to be displayed on their respective Seven Segment Display.

If the hexadecimal to seven-segment decode bit is enabled in the parallel control register, four bits of input will be decoded into a single hexadecimal digit and displayed on the seven-segment display.

If the hexadecimal to seven-segment decode bit is turned off, then each segment can be individually controlled by a single bit of the input. The displays are made up of seven segments and a decimal point. The first eight bits of input turn on the segments as shown in Figure 3.8.

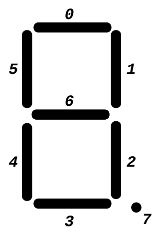


Figure 3.8: Seven-segment display bit encoding

3.3.4 Parallel Control Register

The Parallel Control Register is a read/write register, which allows for control over the parallel interface.

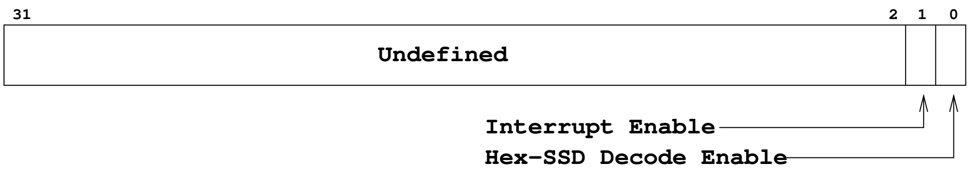


Figure 3.9: The Parallel Control Register

eg. To enable interrupts on switch changes and force hex-SSD decoding on the displays, a value of ‘11’ would be written to the parallel control register.

3.3.5 Parallel Interrupt Acknowledge Register

The Interrupt Acknowledge Register is a read/write register. This register allows a program to determine the parallel port interrupt status as well as acknowledge interrupts that have been dealt with.

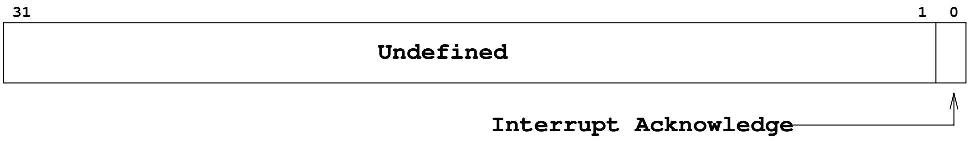


Figure 3.10: The Parallel Interrupt Acknowledge Register

eg. To acknowledge an outstanding parallel port interrupt ‘0’ would be written to the parallel interrupt acknowledge register.

3.4 Programmable Timer

The Programmable Timer on the REX board provides for generation of interrupts at time intervals from about 1ms to 30s, with a resolution of around 0.5ms.

The timer has an internal 16 bit register. This register is decremented at a constant rate of 2400Hz. Once this register reaches `0x0000` an interrupt is triggered. The starting value for the timer is controlled by altering the value in the timer load register.

The timer can be configured to automatically reload the starting count value and continue counting immediately after it expires.

The programmers view of the timer consists of four registers. The names of these registers and their addresses, expressed as offsets from the base address, are provided in Table 3.4. The base address for the timer is `0x72000`.

Register name	Offset
Timer Control Register	0
Timer Load Register	1
Timer Count Register	2
Timer Interrupt Acknowledge Register	3

Table 3.4: Timer Register Offsets

3.4.1 Timer Control Register

The Timer Control Register is a read/write register, that allows the user to enable and control aspects of the timer operation. The timer has two primary modes of operation, automatic restart and single-shot mode. If the timer is set to automatic restart, as soon as the timer expires, an interrupt is triggered and the timer immediately starts counting down again. In single-shot mode the timer will copy the value from the timer load register only once when the timer is enabled and will count down to zero. Once the timer reaches zero an interrupt will be triggered and the timer will be disabled.

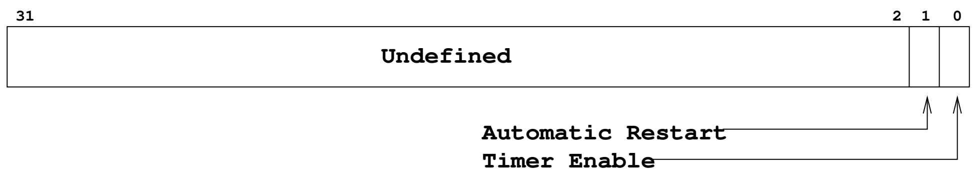


Figure 3.11: The Timer Control Register

3.4.2 Timer Load Register

The Timer Load Register is a read/write register. This register allows the user to specify the starting count value. The starting count value is a 16 bit value with the upper 16 bits being ignored.

3.4.3 Timer Count Register

The Timer Count Register is a read-only register. Reading from this register returns the current value in the 16 bit internal count register.

3.4.4 Timer Interrupt Acknowledge Register

The Interrupt Acknowledge Register is a read/write register. This register allows a program to detect a timer overrun as as well as acknowledge interrupts that have been dealt with.

The overrun detected bit will be set if the timer is set to automatic restart and the timer expired again before the previous interrupt was acknowledged. This allows a program to detect if it is unable to service the timer interrupt fast enough.

The overrun bit must be manually reset by writing a '0' to it's location.

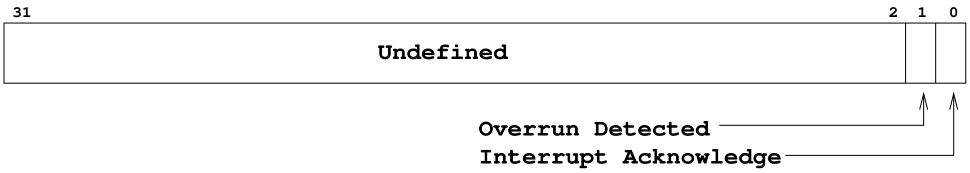


Figure 3.12: The Timer Interrupt Acknowledge Register

eg. If the value '11' was read from the interrupt acknowledge register we could determine that the timer has overrun since we last acknowledged an interrupt. If '00' was written to the timer interrupt acknowledge register we will acknowledge any outstanding interrupts and ensure the overrun bit is reset to zero.

3.4.5 Timer Example

To configure the timer to interrupt at a specific period the first step is to calculate the timer load value. This value can be calculated simply by multiplying the timer frequency by the required time between interrupts. For example, if we want the timer to generate an interrupt once every ten seconds we would calculate it as follows:

$$\text{Timer Load} = 2400\text{Hz} * 10\text{s} = 24000 = 0\text{x}5\text{dc}0$$

Some simple code to initialise the timer to automatically restart and to interrupt once every ten seconds is given in Figure 3.13.

```
 . . .
# Make sure there are no old interrupts
# still hanging around
sw   $0, 0x72003($0)
# Put our auto load value in
addi $11, $0, 0x5dc0
sw   $11, 0x72001($0)
# Enable the timer and autorestart
addi $11, $0, 0x3
sw   $11, 0x72000($0)
 . . .
```

Figure 3.13: Simple Timer Initialisation

Chapter 4

Exceptions

4.1 Introduction

Modern processors can execute millions of instructions each second. This means that when a processor is polling an I/O device for data or status information which may only change very infrequently, it is wasting a lot of time where it could be doing some worthwhile processing. It would be more efficient for a device to signal the CPU when something happens (eg. a character is received at the serial port, the user flicks a switch, or a certain time has elapsed).

Also consider what should happen if something goes wrong when a program is executing. What should happen if an attempt is made to divide by zero? What should happen if you add two numbers and the result will not fit in 32 bits?

This is why almost all modern processors provide support for exceptions. Exceptions provide a mechanism which allows the processor to be executing code, and when a certain condition occurs, to deal with that condition, and then return to what it was doing initially.

The terms 'exception' and 'interrupt' are often used interchangeably. There are varying opinions on the exact definitions of these terms, however the term 'interrupt' generally refers only to the exceptions which are caused by something outside the processor (eg. the serial port, or the timer).

The WRAMP processor allows for four internal exceptions. These are:

- Arithmetic Exception (ie. Divide-by-zero, or Overflow)
- Breakpoint Exception (a 'break' instruction has been executed)
- System Call Exception (a 'syscall' instruction has been executed)
- General Protection Fault Exception (eg. an illegal instruction is encountered)

WRAMP provides eight external interrupts. The external interrupts are simply wires coming into the processor, and so can be connected to any devices. These are called IRQ0 (for Interrupt ReQuest) to IRQ7. On the REX board these are connected as follows:

IRQ #	Description	IRQ #	Description
0	Unconnected	4	Serial Port 1 Interrupt
1	User Interrupt Button	5	Serial Port 2 Interrupt
2	Timer Interrupt	6	Unconnected
3	Parallel Interrupt	7	Unconnected

Exceptions can be thought of as similar to subroutine calls. The processor is executing a block of code, when an exception occurs, causing the processor to jump to a location called the ‘exception vector’. The processor then executes the code at this location (known as the ‘exception handler’ or ‘exception routine’), and returns to the point at which it was executing when the exception occurred.

For this mechanism to work, we will need registers to store things like the exception vector (the address of the exception handler), and the address of the instruction to return to after the exception has been handled. The general purpose registers are not suitable for this, because a program may be using them, and if an exception occurs, then there may be unpredictable results. For this reason the WRAMP processor provides a special set of registers that are used for advanced processor features like exceptions.

Like the general purpose registers (`$0 - $ra`), there are 16 special purpose registers. Because each has a specific use, they are called by their names rather than their numbers. The special registers concerned with exceptions are:

- `$cctrl` - CPU Control Register
- `$estat` - Exception Status Register
- `$evec` - Exception Vector Register
- `$ear` - Exception Address Register
- `$ers` - Exception Register Save

These special purpose registers cannot be operated on directly like the general purpose registers. Rather, two instructions are provided to allow register contents to be copied from a general purpose register to a special purpose register, or vice-versa. These instructions are `movsg` (move special register to general register), and `movgs` (move general register to special register). Details on these instructions can be found in the WRAMP instruction reference in Appendix A.

The next sections will describe the format of each of these special purpose registers. Contained in these descriptions will often be introductions to new concepts and ideas. As such this chapter is best read once end-to-end to ensure that all concepts are introduced fully.

4.2 CPU Exception Control Registers

4.2.1 $cctrl - CPU Control Register

31 12	11 4	3	2	1	0
Undefined	Interrupt Mask	KU	OKU	IE	OIE

Figure 4.1: $cctrl - CPU Control Register

The CPU control register controls almost all of the functionality related to the WRAMP exception mechanism. There are three main sections of this register:

- Interrupt Enable (IE)
- Kernel/User Mode (KU)
- Interrupt Mask

Interrupt Enable

This flag provides a global interrupt enable. If this location is set to '0' then no interrupts can be triggered. This flag *only* affects external interrupts. There is no way on the WRAMP processor to disable internal exceptions.

Interrupts that occur while the global interrupt enable is turned off will be held back. As soon as interrupts are again enabled by writing a '1' into this location the interrupt mask will be consulted to discover if that specific interrupt is enabled. See Section 4.2.1 for more information about the interrupt mask.

The CPU will automatically set the IE bit to '0' whenever an exception of any type occurs. This prevents the exception handler from being interrupted by another interrupt.

Interrupt Mask

This provides a way to selectively turn on and off individual external interrupts. This field has a bit corresponding to each of the eight possible external interrupts (IRQ0 - IRQ7). Bit 4 of the CPU control register corresponds to IRQ0, bit 5 to IRQ1 and so on.

The interrupt mask field is only consulted if the global interrupt enable (IE) flag is set. If an interrupt occurs and the global interrupt flag is set but the individual interrupt mask is disabled then the interrupt will be held back. As soon as both the global interrupt enable and the specific interrupt mask bits are set then the interrupt will occur.

Kernel User Mode

The WRAMP CPU has two modes of operation, kernel and user mode. If there is a '1' in the KU bit the CPU is in kernel mode. If the KU bit is set to '0' then the CPU is in user mode.

If the CPU is running in kernel mode it will execute all instructions and allow access to all areas of memory. If the CPU is running in user mode programs are not allowed to use any of the three instructions which deal with the special register file (`movsg, movgs, rfe`) and may not be able to access all memory locations. If a program running in user mode attempts to use one of these instructions, or to access protected memory the CPU will cause a General Protection Fault exception.

The CPU will automatically set the KU bit to '1' whenever an exception of any type occurs. This allows the exception handler to operate in kernel mode, allowing it full access to all instructions and memory locations.

4.2.2 $estat - Exception Status Register

31 — 16	15 — 12	11 — 4	3 — 0
Undefined	**SW Exc**	**HW Ints**	**Undef**

Figure 4.2: $estat - Exception Status Register

The exception status register provides the exception handler with the ability to discover which exceptions caused it to be invoked. The exception status register has a single bit flag for each external IRQ line. Bit 4 of the status register corresponds to IRQ0, bit 5 to IRQ 1 and so on. These locations are exactly the same as the locations of the interrupt mask fields in the CPU control register as described in Section 4.2.1.

In addition to the eight external interrupt sources the status register also provides the status for the four CPU internal exception sources.

The full list of all exception sources and their related status register bit is given in Table 4.1.

Most exception handlers will wish to check which exception caused them to be called. Provided in Figure 4.3 is code that checks if a specific interrupt caused the handler to be called. If any other interrupt or exception is currently high then the code will call an old handler to deal with the exception. The code to save the address of an old handler is given in Figure 4.4 and discussed in Section 4.2.3.

Exception source	**Bit location**
IRQ0	4
IRQ1 - User Interrupt Button	5
IRQ2 - Timer Interrupt	6
IRQ3 - Parallel Interrupt	7
IRQ4 - Serial Port 1 Interrupt	8
IRQ5 - Serial Port 2 Interrupt	9
IRQ6	10
IRQ7	11
General Protection Fault Exception	12
System Call Exception	13
Breakpoint Exception	14
Arithmetic Exception	15

Table 4.1: Exception Status Register Fields

4.2.3 $evec - Exception Vector Register

When an exception occurs the CPU needs to jump to the exception handler. The CPU must therefore know what address the exception handler starts at.

The address of the exception handler is stored in the exception vector register. The CPU jumps to this location whenever an exception occurs.

If a program is replacing an existing exception handler, but will still need to call the old handler for certain exceptions, the code must be careful to save the address of the old exception handler. This allows the new handler to decide if it will deal with this exception, and if not, call the old handler.

A section of WRAMP code to save the address of an old exception handler, load the address of the new handler and save this to the exception vector is given in Figure 4.4.

4.2.4 $ear - Exception Address Register

An exception routine must be able to return to the point in program code at which the exception occurred. To allow this, when an exception occurs the WRAMP pocessor automatically saves the address of the next instruction that would have been executed into `$ear` - the Exception Address Register.

When the exception routine has completed its processing, it executes a Return From Exception, or `rfe` instruction. Amongst other things, this instruction causes a jump to the address contained in `$ear`.

In some circumstances an exception routine may wish to know the address at which the exception was invoked, or may wish to alter the address to which the `rfe` will return. This can be achieved by inspecting and/or modifying the contents of the `$ear`.

```
        . . .
handler:

      # Get the status register
      movsg   $13, $estat

      # Inspect only the bits we are interested in. We want
      # to check that no bits from the sw exceptions or
      # hardware exceptions, other than the one we were
      # expecting, are enabled.
      #
      # This code is looking for an IRQ4 interrupt.
      andi    $13, $13, 0xfef0

      # If the result of this is zero then no other
      # exceptions are enabled so it must be our interrupt
      # that caused us to be called.
      beqz    $13, handle_interrupt

      # Otherwise there was another exception that has
      # occurred, so call the old handler
      lw      $13, old_vector($0)
      jr      $13

handle_interrupt:

      # This is where we deal with the interrupt.
         . . .
```

Figure 4.3: Checking the Status Register

4.2.5 $ers - Exception Save Register

It is vital that an exception routine does not change the contents of the general purpose registers when it returns to the main program, as changes may cause the main program to behave in an unpredictable fashion.

However, for the exception handler to determine the cause of the exception, it requires a general purpose register into which it can copy `$estat`. The WRAMP processor makes general purpose register `$13` available for this by automatically copying it to the Exception Save Register (`$ers`) when an exception occurs. The opposite of this happens when an `rfe` instruction is executed - `$ers` is copied into `$13`.

This means that exception handler code must only change `$13`. If it needs other registers it must save their contents before using them. They must then be restored before returning to the main program.

4.3 User Interrupt Button

The REX board provides a simple method for creating an interrupt. There is a button located on the front of the case, next to the reset button labeled "INTR". When this button is pressed IRQ1, the "User Interrupt Button", interrupt will be triggered. If this interrupt is unmasked and the global interrupt enable is turned on in `$cctrl` an exception will occur.

This button provides a simple way to test an exception handler as it avoids problems that could be caused by mis-configuration of the I/O device that is being used to provide the exception.

Like all other REX interrupt sources the user interrupt button needs to be acknowledged each time an exception occurs. To acknowledge a "User Interrupt" you store zero to the address `0x7f000`. Unlike the other I/O devices on the REX board you do not need to enable or disable the user interrupt button. If

```
        . . .
        # Get the old exception vector
        movsg   $4, $evec
        # And save it
        sw      $4, old_vector($0)

        # Get the address of our handler
        la      $4, handler
        # And put it in the exception vector register
        movgs   $evec, $4

            . . .

handler:
        # The exception handler goes here

            . . .

old_vector:
        .word   0
```

Figure 4.4: Saving and Initialising the Exception Vector

IRQ1 is unmasked in `$cctrl` and interrupts are enabled the button will cause an exception when pressed.

4.4 Using Exceptions

Writing a program that uses exceptions is best done as a step by step process. If you attempt to write an entire program that uses exceptions from start to finish in one hit, then there is a good chance you will never debug any problems that may arise.

The first thing that you will need when writing your first handler is a simple program that will run as the main loop of your code. You use this to ensure that the exception routine is returning to your original code correctly. A program which reads the value on the switches and writes this to the seven segment display is ideal. Obviously you should do this in a polled fashion.

Next you should write a very simple piece of code that will constitute your exception handler. A suggested program is one which writes a single character to a serial port. As this code will eventually be run from inside your exception handler it must transmit this character using polled I/O.

Test that both of these pieces of code work in a normal environment with no exceptions. Also ensure that the code which will act as your exception handler makes use only of `$13`.

The next step is to actually enable an interrupt and get the handler you just wrote to be run. As suggested above the best source for your first interrupt is the user interrupt button, IRQ1. The things that you need to do to get this working are:

- Save the old exception handler address
- Put the address of your new handler into `$evec`
- Make sure that there are no old interrupts hanging around by storing `$0` to the acknowledge register of the device you will be using. For IRQ1 the acknowledge register is at address `0x7f000`.
- Configure the CPU control register to enable interrupts. This takes a number of smaller steps:
  - Get the current value of `$cctrl`

– Disable all interrupts.
– Enable the interrupt you wish to use (IRQ1) and set the global interrupt enable to '`1`'. Be careful that you do not alter any other locations in this register besides the ones specified.
– Store this value back into `$cctrl`.

You will need to add some code to the simple exception handler that you just wrote to make it a complete exception handler. Your exception handler must start with code to detect if the interrupt is one that you wish to deal with. Some example code for this is given in Figure 4.3. Beware that you will need to alter this code so that it checks for the correct interrupt. If you are using the user interrupt button you should make sure the code checks that only IRQ1 is high.

Next you must remember to acknowledge the interrupt. If you do not acknowledge the interrupt then as soon as your exception routine exits it will be instantly called again. This means you code will be stuck in an infinite loop, probably printing character after character to the serial port.

The very final instruction of you exception handler must be an `rfe`. Only use an `rfe` instruction in code that you are sure will only be called as part of an exception handler. If you use the `rfe` instruction when you are not inside an exception handler you may find your code will get stuck in an infinite loop or crash.

If this code is now working, then you should have a character appearing each time you push the user interrupt button as well as the value on the switches constantly being displayed on the seven segment display. If so, try altering your code so that you use one of the other I/O devices to cause exceptions.

4.5 Exception Procedure

What actually happens when an exception occurs? The CPU performs a number of operations when an exception occurs but they are all pretty simple. The CPU does the following:

- Copy the IE bit into the OIE bit
- Set IE to zero
- Copy the KU bit into the OKU bit
- Set KU to one
- Set the `$estat` register to reflect the cause of the exception
- Copy `$13` into `$ers`
- Save the program counter into `$ear`
- Set the program counter to the contents of `$evec`

At this point the next instruction is fetched. This instruction is the first instruction of the exception handler and therefore the handler is now running.

Once the exception handler has finished the final instruction it will call will be an `rfe`. The CPU takes the following steps to execute an `rfe` instruction:

- Set the IE bit to OIE
- Set the KU bit to OKU
- Copy `$ers` into `$13`
- Set the program counter to the contents of `$ear`

This means that the next instruction fetched will normally be the next instruction of the original code. The IE and KU bits will also normally be restored to the value they had when the exception occurred.

4.6 Compliant Exception Routine

The exception proceedure above provides a mechanism whereby an exception routine can run when an exception occurs and restore control to the main program without affecting its operation. In this way an exception routine can be general purpose to be used during the operation of any program. Such a routine should:

- Only use`$13` freely
- Save the contents of any other register before using it.
- Restore the saved contents of any other register before finishing
- Ensure that the contents of the OIE bit, the OKU bit, `$ers` and `$ear` are not modified
- Finish with an `rfe` instruction.

In addition any new exception routine that is installed must:

- Save the address of the system exception handler
- Check the exception type
- Pass any exceptions it cannot handle to the system exception handler

An exception routine that meets all these requirements is termed *compliant* with the WRAMP conventions.

Appendix A

Instruction Set

WRAMP General Purpose Registers

The WRAMP general purpose register file consists of 16 registers, each being 32 bits wide. The hardware imposes special uses on only two of these. Certain software register use conventions have been applied to some of the remaining registers, but in essence they remain true general purpose registers, as the hardware does not restrict their use.

Register	Description
`$0`	Hardwired zero
`$1 - $13`	General purpose registers
`$sp`	Stack pointer
`$ra`	Return address register

Figure A.1: The WRAMP General Purpose Registers

Register zero (denoted `$0`) always contains the value zero. Any writes to this register have the value discarded. This provides a constant source of zero that can be used for comparing and initialising registers.

The fourteenth register is denoted `$sp`. This register is defined by the conventions to be the stack pointer. While the hardware imposes no special conditions on this register, failure to follow this convention may affect the ability of code to interoperate with other software.

The fifteenth register is denoted `$ra`. It is defined to be the subroutine return address register. When a jump and link instruction is executed this register is loaded with the address of the next instruction after the jump and link. A return from subroutine is performed by executing a jump to register `$ra`, ie. `jr $ra`.

WRAMP Instruction Set Architecture

This section contains the details of the WRAMP instruction set. All machine instructions are listed, with their encoding and a brief description of their function. The instructions are grouped into arithmetic instructions, bitwise instructions, test instructions, branch instructions, memory instructions, and special instructions.

Each CPU instruction is a word (32 bits) in length. An instruction is encoded in one of the three formats shown in figure A.2.

I-Type instruction

4 bits	4 bits	4 bits	4 bits	16 bits
OPcode	R_d	R_s	Func	Immediate

R-Type instruction

4 bits	4 bits	4 bits	4 bits	12 bits	4 bits
OPcode	R_d	R_s	Func	0000 0000 0000	R_t

J-Type instruction

4 bits	4 bits	4 bits	20 bits
OPcode	R_d	R_s	Address / Offset

`OPCode`	4 bit operation code
R_d	4 bit destination register specifier
R_s	4 bit source register specifier
R_t	4 bit source register specifier
`Func`	4 bit function specifier
`Immediate`	16 bit immediate field
`Address / Offset`	20 bit absolute or relative address field

Figure A.2: WRAMP Instruction encoding formats

Arithmetic Instructions

Addition

`add` R_d, R_s, R_t

0000	R_d	R_s	0000	0000 0000 0000	R_t
4	4	4	4	12	4

Put the sum of register R_s and register R_t into register R_d. Generate an overflow exception on signed overflow.

Addition, immediate

`addi` R_d, R_s, `Immediate`

0001	R_d	R_s	0000	Immediate
4	4	4	4	16

Put the sum of register R_s and the sign-extended immediate into register R_d. Generate an overflow exception on signed overflow.

Addition, unsigned
`addu` R_d, R_s, R_t

0000	R_d	R_s	0001	0000 0000 0000	R_t
4	4	4	4	12	4

Put the sum of register R_s and register R_t into register R_d. Generate an overflow exception on unsigned overflow.

Addition, unsigned, immediate
`addui` R_d, R_s, `Immediate`

0001	R_d	R_s	0001	Immediate
4	4	4	4	16

Put the sum of register R_s and the zero-extended immediate into register R_d. Generate an overflow exception on unsigned overflow.

Subtraction
`sub` R_d, R_s, R_t

0000	R_d	R_s	0010	0000 0000 0000	R_t
4	4	4	4	12	4

Put the difference of register R_s and register R_t into register R_d. Generate an overflow exception on signed overflow.

Subtraction, immediate
`subi` R_d, R_s, `Immediate`

0001	R_d	R_s	0010	Immediate
4	4	4	4	16

Put the difference of register R_s and the sign-extended immediate into register R_d. Generate an overflow exception on signed overflow.

Subtraction, unsigned
`subu` R_d, R_s, R_t

0000	R_d	R_s	0011	0000 0000 0000	R_t
4	4	4	4	12	4

Put the difference of register R_s and register R_t into register R_d. Generate an overflow exception on unsigned overflow.

Subtraction, unsigned, immediate
`subui` R_d, R_s, `Immediate`

0001	R_d	R_s	0011	Immediate
4	4	4	4	16

Put the difference of register R_s and the zero-extended immediate into register R_d. Generate an overflow exception on unsigned overflow.

Multiplication
`mult` R_d, R_s, R_t

0000	R_d	R_s	0100	0000 0000 0000	R_t
4	4	4	4	12	4

Put the product of the signed multiplication of register R_s and register R_t into register R_d. Generate an overflow exception on signed overflow.

Multiplication, immediate
`multi` R_d, R_s, `Immediate`

0001	R_d	R_s	0100	Immediate
4	4	4	4	16

Put the product of the signed multiplication of register R_s and the sign-extended immediate into register R_d. Generate an overflow exception on signed overflow.

Multiplication, unsigned
`multu` R_d, R_s, R_t

0000	R_d	R_s	0101	0000 0000 0000	R_t
4	4	4	4	12	4

Put the product of the unsigned multiplication of register R_s and register R_t into register R_d. Generate an overflow exception on unsigned overflow.

Multiplication, unsigned, immediate
`multui` R_d, R_s, `Immediate`

0001	R_d	R_s	0101	Immediate
4	4	4	4	16

Put the product of the unsigned multiplication of register R_s and the zero-extended immediate into register R_d. Generate an overflow exception on unsigned overflow.

Division
`div` R_d, R_s, R_t

0000	R_d	R_s	0110	0000 0000 0000	R_t
4	4	4	4	12	4

Put the result of the signed integer division of register R_s by register R_t into register R_d. Generate a divide-by-zero exception if the contents of R_t is zero.

Division, immediate
`divi` R_d, R_s, `Immediate`

0001	R_d	R_s	0110	Immediate
4	4	4	4	16

Put the result of the signed integer division of register R_s by the sign-extended immediate into register R_d. Generate a divide-by-zero exception if the immediate value is zero.

Division, unsigned
`divu` R_d, R_s, R_t

0000	R_d	R_s	0111	0000 0000 0000	R_t
4	4	4	4	12	4

Put the result of the unsigned division of register R_s by register R_t into register R_d. Generate a divide-by-zero exception if the contents of R_t is zero.

Division, unsigned, immediate
`divui` R_d, R_s, `Immediate`

0001	R_d	R_s	0111	Immediate
4	4	4	4	16

Put the result of the unsigned division of register R_s by the zero-extended immediate into register R_d. Generate a divide-by-zero exception if the immediate value is zero.

Remainder
`rem` R_d, R_s, R_t

0000	R_d	R_s	1000	0000 0000 0000	R_t
4	4	4	4	12	4

Put the remainder of the signed division of register R_s by register R_t into register R_d. Generate a divide-by-zero exception if the contents of R_t is zero.

Remainder, immediate
`remi` R_d, R_s, `Immediate`

0001	R_d	R_s	1000	Immediate
4	4	4	4	16

Put the remainder of the signed division of register R_s by the sign-extended immediate into register R_d. Generate a divide-by-zero exception if the immediate value is zero.

Remainder, unsigned
`remu` R_d, R_s, R_t

0000	R_d	R_s	1001	0000 0000 0000	R_t
4	4	4	4	12	4

Put the remainder of the unsigned division of register R_s by the register R_t into register R_d. Generate a divide-by-zero exception if the contents of R_t is zero.

Remainder, unsigned, immediate
`remui` R_d, R_s, `Immediate`

0001	R_d	R_s	1001	Immediate
4	4	4	4	16

Put the remainder of the unsigned division of register R_s by the zero-extended immediate into register R_d. Generate a divide-by-zero exception if the immediate value is zero.

Load high immediate
`lhi` R_d, `Immediate`

0011	R_d	0000	1110	Immediate
4	4	4	4	16

Put the 16 bit immediate into the upper 16 bits of register R_d, and set the lower 16 bits to zero.

Load address
`la` R_d, `Address`

1100	R_d	0000	Address
4	4	4	20

Put the zero-extended 20 bit address into register R_d.

Bitwise instructions

And

`and` R_d, R_s, R_t

0000	R_d	R_s	1011	0000 0000 0000	R_t
4	4	4	4	12	4

Put the result of the logical AND of registers R_s and R_t into register R_d.

And, immediate

`andi` R_d, R_s, `Immediate`

0001	R_d	R_s	1011	Immediate
4	4	4	4	16

Put the result of the logical AND of register R_s and the zero-extended immediate into register R_d.

Or

`or` R_d, R_s, R_t

0000	R_d	R_s	1101	0000 0000 0000	R_t
4	4	4	4	12	4

Put the result of the logical OR of registers R_s and R_t into register R_d.

Or, immediate

`ori` R_d, R_s, `Immediate`

0001	R_d	R_s	1101	Immediate
4	4	4	4	16

Put the result of the logical OR of register R_s and the zero-extended immediate into register R_d.

Xor

`xor` R_d, R_s, R_t

0000	R_d	R_s	1111	0000 0000 0000	R_t
4	4	4	4	12	4

Put the result of the logical exclusive-OR of registers R_s and R_t into register R_d.

Xor, immediate

`xori` R_d, R_s, `Immediate`

0001	R_d	R_s	1111	Immediate
4	4	4	4	16

Put the result of the logical exclusive-OR of register R_s and the zero-extended immediate into register R_d.

Shift left logical

`sll` R_d, R_s, R_t

0000	R_d	R_s	1010	0000 0000 0000	R_t
4	4	4	4	12	4

Shift the value in register R_s left by the unsigned value given by the least significant 5 bits of register R_t, and put the result in register R_d, inserting zeros into the low order bits.

Shift left logical, immediate

`slli` R_d, R_s, `Immediate`

0001	R_d	R_s	1010	Immediate
4	4	4	4	16

Shift the value in register R_s left by the unsigned value given by the least significant 5 bits of the immediate, and put the result in register R_d, inserting zeros into the low order bits.

Shift right logical

`srl` R_d, R_s, R_t

0000	R_d	R_s	1100	0000 0000 0000	R_t
4	4	4	4	12	4

Shift the value in register R_s right by the unsigned value given by the least significant 5 bits of register R_t, and place the result in register R_d, inserting zeros into the high order bits.

Shift right logical, immediate

`srli` R_d, R_s, `Immediate`

0001	R_d	R_s	1100	Immediate
4	4	4	4	16

Shift the value in register R_s right by the unsigned value given by the least significant 5 bits of the immediate, and place the result in register R_d, inserting zeros into the high order bits.

Shift right arithmetic
`sra` R_d, R_s, R_t

0000	R_d	R_s	1110	0000 0000 0000	R_t
4	4	4	4	12	4

Shift the value in register R_s right by the unsigned value given by the least significant 5 bits of register R_t, and place the result in register R_d, sign-extending the high order bits.

Shift right arithmetic, immediate
`srai` R_d, R_s, `Immediate`

0001	R_d	R_s	1110	Immediate
4	4	4	4	16

Shift the value in register R_s right by the unsigned value given by the least significant 5 bits of the immediate, and place the result in register R_d, sign-extending the high order bits.

Test instructions

Set on less than
`slt` R_d, R_s, R_t

0010	R_d	R_s	0000	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is less than register R_t according to a signed comparison, and 0 otherwise.

Set on less than immediate
`slti` R_d, R_s, `Immediate`

0011	R_d	R_s	0000	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is less than the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on less than, unsigned
`sltu` R_d, R_s, R_t

0010	R_d	R_s	0001	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is less than register R_t according to an unsigned comparison, and 0 otherwise.

Set on less than, unsigned, immediate
`sltui` R_d, R_s, `Immediate`

0011	R_d	R_s	0001	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is less than the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Set on greater than
`sgt` R_d, R_s, R_t

0010	R_d	R_s	0010	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is greater than register R_t according to a signed comparison, and 0 otherwise.

Set on greater than, immediate
`sgti` R_d, R_s, `Immediate`

0011	R_d	R_s	0010	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is greater than the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on greater than, unsigned
`sgtu` R_d, R_s, R_t

0010	R_d	R_s	0011	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is greater than register R_t according to an unsigned comparison, and 0 otherwise.

Set on greater than, unsigned, immediate
`sgtui` R_d, R_s, `Immediate`

0011	R_d	R_s	0011	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is greater than the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Set on less than or equal to
`sle` R_d, R_s, R_t

0010	R_d	R_s	0100	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is less than or equal to register R_t according to a signed comparison, and 0 otherwise.

Set on less than or equal to, immediate
`slei` R_d, R_s, `Immediate`

0011	R_d	R_s	0100	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is less than or equal to the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on less than or equal to, unsigned
`sleu` R_d, R_s, R_t

0010	R_d	R_s	0101	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is less than or equal to register R_t according to an unsigned comparison, and 0 otherwise.

Set on less than or equal to, unsigned, immediate
`sleui` R_d, R_s, `Immediate`

0011	R_d	R_s	0101	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is less than or equal to the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Set on greater than or equal to
`sge` R_d, R_s, R_t

0010	R_d	R_s	0110	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is greater than or equal to register R_t according to a signed comparison, and 0 otherwise.

Set on greater than or equal to immediate
`sgei` R_d, R_s, `Immediate`

0011	R_d	R_s	0110	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is greater than or equal to the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on greater than or equal to, unsigned
`sgeu` R_d, R_s, R_t

0010	R_d	R_s	0111	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is greater than or equal to register R_t according to an unsigned comparison, and 0 otherwise.

Set on greater than or equal to, unsigned, immediate
`sgeui` R_d, R_s, `Immediate`

0011	R_d	R_s	0111	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is greater than or equal to the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Set on equal to
`seq` R_d, R_s, R_t

0010	R_d	R_s	1000	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is equal to register R_t according to a signed comparison, and 0 otherwise.

Set on equal to immediate
`seqi` R_d, R_s, `Immediate`

0011	R_d	R_s	1000	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is equal to the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on equal to, unsigned
`sequ` R_d, R_s, R_t

0010	R_d	R_s	1001	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is equal to register R_t according to an unsigned comparison, and 0 otherwise.

Set on equal to, unsigned, immediate
`sequi` R_d, R_s, `Immediate`

0011	R_d	R_s	1001	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is equal to the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Set on not equal to
`sne` R_d, R_s, R_t

0010	R_d	R_s	1010	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is not equal to register R_t according to a signed comparison, and 0 otherwise.

Set on not equal to immediate
`snei` R_d, R_s, `Immediate`

0011	R_d	R_s	1010	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is not equal to the sign-extended immediate according to a signed comparison, and 0 otherwise.

Set on not equal to, unsigned
`sneu` R_d, R_s, R_t

0010	R_d	R_s	1011	0000 0000 0000	R_t
4	4	4	4	12	4

Set register R_d to 1 if register R_s is not equal to register R_t according to an unsigned comparison, and 0 otherwise.

Set on not equal to, unsigned, immediate
`sneui` R_d, R_s, `Immediate`

0011	R_d	R_s	1011	Immediate
4	4	4	4	16

Set register R_d to 1 if register R_s is not equal to the zero-extended immediate according to an unsigned comparison, and 0 otherwise.

Branch instructions

Jump
`j Address`

0100	0000	0000	Address
4	4	4	20

Unconditionally jump to the instruction whose address is given by the address field.

Jump to register
`jr` R_s

0101	0000	R_s	0000 0000 0000 0000 0000
4	4	4	20

Unconditionally jump to the instruction whose address is in the least significant 20 bits of register R_s.

Jump and link
`jal Address`

0110	0000	0000	Address
4	4	4	20

Unconditionally jump to the instruction whose address is given by the address field. Save the address of the next instruction in register `$ra`.

Jump and link register

`jalr` R_s

0111	0000	R_s	0000 0000 0000 0000 0000
4	4	4	20

Unconditionally jump to the instruction whose address is in the least significant 20 bits of register R_s. Save the address of the next instruction in register `$ra`.

Branch on equal to zero

`beqz` R_s, `Offset`

1010	0000	R_s	Offset
4	4	4	20

Conditionally branch the number of instructions specified by the sign-extended offset if register R_s is equal to 0.

Branch on not equal to zero

`bnez` R_s, `Offset`

1011	0000	R_s	Offset
4	4	4	20

Conditionally branch the number of instructions specified by the sign-extended offset if register R_s is not equal to 0.

Memory instructions

Load word

`lw` R_d, `Offset`(R_s)

1000	R_d	R_s	Offset
4	4	4	20

Add the contents of register R_s and the sign-extended offset to give an effective address. Load the contents of the memory location specified by this effective address into register R_d.

Store word

`sw` R_d, `Offset`(R_s)

1001	R_d	R_s	Offset
4	4	4	20

Add the contents of register R_s and the sign-extended offset to give an effective address. Store the contents of register R_d into the memory location specified by this effective address.

Special instructions

Move general register to special register

`movgs` R_d, R_s

0011	R_d	R_s	1100	0000 0000 0000 0000
4	4	4	4	16

Copy the contents of general purpose register R_s into special purpose register R_d.

Move special register to general register

`movsg` R_d, R_s

0011	R_d	R_s	1101	0000 0000 0000 0000
4	4	4	4	16

Copy the contents of special purpose register R_s into general purpose register R_d.

Break

`break`

0010	0000	0000	1100	0000 0000 0000 0000
4	4	4	4	16

Generate a breakpoint exception, transferring control to the exception handler.

System call

`syscall`

0010	0000	0000	1101	0000 0000 0000 0000
4	4	4	4	16

Generate a system call exception, transferring control to the exception handler.

Return from exception

`rfe`

0010	0000	0000	1110	0000 0000 0000 0000
4	4	4	4	16

Restore the saved interrupt enable and kernel/user mode bits and jump to the instruction at the address specified in the special register `$ear`.

Appendix B

Small Instruction Set

WRAMP Instruction Set Summary

Assembler	Machine code	Function	Description
`add` R_d, R_s, R_t	0000 dddd ssss 0000 0000 0000 0000 tttt	$R_d \leftarrow R_s + R_t$	Add
`addi` R_d, R_s, immed	0001 dddd ssss 0000 iiii iiii iiii iiii	$R_d \leftarrow R_s + \int(immed)$	Add Immediate
`addu` R_d, R_s, R_t	0000 dddd ssss 0001 0000 0000 0000 tttt	$R_d \leftarrow R_s + R_t$	Add Unsigned
`addui` R_d, R_s, immed	0001 dddd ssss 0001 iiii iiii iiii iiii	$R_d \leftarrow R_s + immed$	Add Unsigned Immediate
`sub` R_d, R_s, R_t	0000 dddd ssss 0010 0000 0000 0000 tttt	$R_d \leftarrow R_s - R_t$	Subtract
`subi` R_d, R_s, immed	0001 dddd ssss 0010 iiii iiii iiii iiii	$R_d \leftarrow R_s - \int(immed)$	Subtract Immediate
`subu` R_d, R_s, R_t	0000 dddd ssss 0011 0000 0000 0000 tttt	$R_d \leftarrow R_s - R_t$	Subtract Unsigned
`subui` R_d, R_s, immed	0001 dddd ssss 0011 iiii iiii iiii iiii	$R_d \leftarrow R_s - immed$	Subtract Unsigned Immediate
`mult` R_d, R_s, R_t	0000 dddd ssss 0100 0000 0000 0000 tttt	$R_d \leftarrow R_s \times R_t$	Multiply
`multi` R_d, R_s, immed	0001 dddd ssss 0100 iiii iiii iiii iiii	$R_d \leftarrow R_s \times \int(immed)$	Multiply Immediate
`multu` R_d, R_s, R_t	0000 dddd ssss 0101 0000 0000 0000 tttt	$R_d \leftarrow R_s \times R_t$	Multiply Unsigned
`multui` R_d, R_s, immed	0001 dddd ssss 0101 iiii iiii iiii iiii	$R_d \leftarrow R_s \times immed$	Multiply Unsigned Immediate
`div` R_d, R_s, R_t	0000 dddd ssss 0110 0000 0000 0000 tttt	$R_d \leftarrow R_s \div R_t$	Divide
`divi` R_d, R_s, immed	0001 dddd ssss 0110 iiii iiii iiii iiii	$R_d \leftarrow R_s \div \int(immed)$	Divide Immediate
`divu` R_d, R_s, R_t	0000 dddd ssss 0111 0000 0000 0000 tttt	$R_d \leftarrow R_s \div R_t$	Divide Unsigned
`divui` R_d, R_s, immed	0001 dddd ssss 0111 iiii iiii iiii iiii	$R_d \leftarrow R_s \div immed$	Divide Unsigned Immediate
`rem` R_d, R_s, R_t	0000 dddd ssss 1000 0000 0000 0000 tttt	$R_d \leftarrow R_s \,\%\, R_t$	Remainder
`remi` R_d, R_s, immed	0001 dddd ssss 1000 iiii iiii iiii iiii	$R_d \leftarrow R_s \,\%\, \int(immed)$	Remainder Immediate
`remu` R_d, R_s, R_t	0000 dddd ssss 1001 0000 0000 0000 tttt	$R_d \leftarrow R_s \,\%\, R_t$	Remainder Unsigned
`remui` R_d, R_s, immed	0001 dddd ssss 1001 iiii iiii iiii iiii	$R_d \leftarrow R_s \,\%\, immed$	Remainder Unsigned Immediate
`lhi` R_d, immed	0011 dddd ssss 1110 iiii iiii iiii iiii	$R_d \leftarrow immed \ll 16$	Load High Immediate
`la` R_d, address	1100 dddd 0000 aaaa aaaa aaaa aaaa aaaa	$R_d \leftarrow address$	Load Address

Table B.1: Arithmetic Instructions

and R_d, R_s, R_t	0000 dddd ssss 1011 0000 0000 0000 tttt	$R_d \leftarrow R_s\ AND\ R_t$	Bitwise AND
andi R_d, R_s, immed	0001 dddd ssss 1011 iiii iiii iiii iiii	$R_d \leftarrow R_s\ AND\ immed$	Bitwise AND Immediate
or R_d, R_s, R_t	0000 dddd ssss 1101 0000 0000 0000 tttt	$R_d \leftarrow R_s\ OR\ R_t$	Bitwise OR
ori R_d, R_s, immed	0001 dddd ssss 1101 iiii iiii iiii iiii	$R_d \leftarrow R_s\ OR\ immed$	Bitwise OR Immediate
xor R_d, R_s, R_t	0000 dddd ssss 1111 0000 0000 0000 tttt	$R_d \leftarrow R_s\ XOR\ R_t$	Bitwise XOR
xori R_d, R_s, immed	0001 dddd ssss 1111 iiii iiii iiii iiii	$R_d \leftarrow R_s\ XOR\ immed$	Bitwise XOR Immediate
sll R_d, R_s, R_t	0000 dddd ssss 1010 0000 0000 0000 tttt	$R_d \leftarrow R_s \ll R_t$	Shift Left Logical
slli R_d, R_s, immed	0001 dddd ssss 1010 iiii iiii iiii iiii	$R_d \leftarrow R_s \ll immed$	Shift Left Logical Immediate
srl R_d, R_s, R_t	0000 dddd ssss 1100 0000 0000 0000 tttt	$R_d \leftarrow R_s \gg R_t$	Shift Right Logical
srli R_d, R_s, immed	0001 dddd ssss 1100 iiii iiii iiii iiii	$R_d \leftarrow R_s \gg immed$	Shift Right Logical Immediate
sra R_d, R_s, R_t	0000 dddd ssss 1110 0000 0000 0000 tttt	$R_d \leftarrow \int(R_s \gg R_t)$	Shift Right Arithmetic
srai R_d, R_s, immed	0001 dddd ssss 1110 iiii iiii iiii iiii	$R_d \leftarrow \int(R_s \gg immed)$	Shift Right Arithmetic Immediate

Table B.2: Bitwise Instructions

slt R_d, R_s, R_t	0010 dddd ssss 0000 0000 0000 0000 tttt	$R_d \leftarrow R_s < R_t$	Set on Less than
slti R_d, R_s, immed	0011 dddd ssss 0000 iiii iiii iiii iiii	$R_d \leftarrow R_s < \int(immed)$	Set on Less than Immediate
sltu R_d, R_s, R_t	0010 dddd ssss 0001 0000 0000 0000 tttt	$R_d \leftarrow R_s < R_t$	Set on Less than Unsigned
sltui R_d, R_s, immed	0011 dddd ssss 0001 iiii iiii iiii iiii	$R_d \leftarrow R_s < immed$	Set on Less than Unsigned Immediate
sgt R_d, R_s, R_t	0010 dddd ssss 0010 0000 0000 0000 tttt	$R_d \leftarrow R_s > R_t$	Set on Greater than
sgti R_d, R_s, immed	0011 dddd ssss 0010 iiii iiii iiii iiii	$R_d \leftarrow R_s > \int(immed)$	Set on Greater than Immediate
sgtu R_d, R_s, R_t	0010 dddd ssss 0011 0000 0000 0000 tttt	$R_d \leftarrow R_s > R_t$	Set on Greater than Unsigned
sgtui R_d, R_s, immed	0011 dddd ssss 0011 iiii iiii iiii iiii	$R_d \leftarrow R_s > immed$	Set on Greater than Unsigned Immediate
sle R_d, R_s, R_t	0010 dddd ssss 0100 0000 0000 0000 tttt	$R_d \leftarrow R_s \leq R_t$	Set on Less than or Equal
slei R_d, R_s, immed	0011 dddd ssss 0100 iiii iiii iiii iiii	$R_d \leftarrow R_s \leq \int(immed)$	Set on Less or Equal Immediate
sleu R_d, R_s, R_t	0010 dddd ssss 0101 0000 0000 0000 tttt	$R_d \leftarrow R_s \leq R_t$	Set on Less or Equal Unsigned
sleui R_d, R_s, immed	0011 dddd ssss 0101 iiii iiii iiii iiii	$R_d \leftarrow R_s \leq immed$	Set on Less or Equal Unsigned Imm
sge R_d, R_s, R_t	0010 dddd ssss 0110 0000 0000 0000 tttt	$R_d \leftarrow R_s \geq R_t$	Set on Greater than or Equal
sgei R_d, R_s, immed	0011 dddd ssss 0110 iiii iiii iiii iiii	$R_d \leftarrow R_s \geq \int(immed)$	Set on Greater or Equal Immediate
sgeu R_d, R_s, R_t	0010 dddd ssss 0111 0000 0000 0000 tttt	$R_d \leftarrow R_s \geq R_t$	Set on Greater or Equal Unsigned
sgeui R_d, R_s, immed	0011 dddd ssss 0111 iiii iiii iiii iiii	$R_d \leftarrow R_s \geq immed$	Set on Greater or Equal Unsigned Imm
seq R_d, R_s, R_t	0010 dddd ssss 1000 0000 0000 0000 tttt	$R_d \leftarrow R_s = R_t$	Set on Equal
seqi R_d, R_s, immed	0011 dddd ssss 1000 iiii iiii iiii iiii	$R_d \leftarrow R_s = \int(immed)$	Set on Equal Immediate
sequ R_d, R_s, R_t	0010 dddd ssss 1001 0000 0000 0000 tttt	$R_d \leftarrow R_s = R_t$	Set on Equal Unsigned
sequi R_d, R_s, immed	0011 dddd ssss 1001 iiii iiii iiii iiii	$R_d \leftarrow R_s = immed$	Set on Equal Unsigned Immediate
sne R_d, R_s, R_t	0010 dddd ssss 1010 0000 0000 0000 tttt	$R_d \leftarrow R_s \neq R_t$	Set on Not Equal
snei R_d, R_s, immed	0011 dddd ssss 1010 iiii iiii iiii iiii	$R_d \leftarrow R_s \neq \int(immed)$	Set on Not Equal Immediate
sneu R_d, R_s, R_t	0010 dddd ssss 1011 0000 0000 0000 tttt	$R_d \leftarrow R_s \neq R_t$	Set on Not Equal Unsigned
sneui R_d, R_s, immed	0011 dddd ssss 1011 iiii iiii iiii iiii	$R_d \leftarrow R_s \neq immed$	Set on Not Equal Unsigned Immediate

Table B.3: Test Instructions

Branch Instructions			
`j address`	0100 0000 0000 aaaa aaaa aaaa aaaa aaaa	$PC \leftarrow Address$	Jump
`jr` R_s	0101 0000 ssss 0000 0000 0000 0000 0000	$PC \leftarrow R_s$	Jump to Register
`jal address`	0110 0000 0000 aaaa aaaa aaaa aaaa aaaa	`$ra` $\leftarrow PC,\ PC \leftarrow Address$	Jump and Link
`jalr` R_s	0111 0000 ssss 0000 0000 0000 0000 0000	`$ra` $\leftarrow PC,\ PC \leftarrow R_s$	Jump and Link Register
`beqz` R_s, `offset`	1010 0000 ssss oooo oooo oooo oooo oooo	$if(R_s = 0)\ PC \leftarrow PC + offset$	Branch on equal to 0
`bnez` R_s, `offset`	1011 0000 ssss oooo oooo oooo oooo oooo	$if(R_s \neq 0)\ PC \leftarrow PC + offset$	Branch on not equal to 0
Memory Instructions			
`lw` R_d, `offset(`R_s`)`	1000 dddd ssss oooo oooo oooo oooo oooo	$R_d \leftarrow MEM[R_s + offset]$	Load word
`sw` R_d, `offset(`R_s`)`	1001 dddd ssss oooo oooo oooo oooo oooo	$MEM[R_s + offset] \leftarrow R_d$	Store word
Special Instructions			
`movgs` R_d, R_s	0011 0000 0000 1100 0000 0000 0000 0000	$R_d \leftarrow R_s$	Move General to Special Register
`movsg` R_d, R_s	0011 0000 0000 1101 0000 0000 0000 0000	$R_d \leftarrow R_s$	Move Special to General Register
`break`	0010 0000 0000 1100 0000 0000 0000 0000		Generate Break Point Exception
`syscall`	0010 0000 0000 1101 0000 0000 0000 0000		Generate Syscall Exception
`rfe`	0010 0000 0000 1110 0000 0000 0000 0000	$PC \leftarrow$ `$ear`	Return from Exception

Table B.4: Other Instructions

Appendix C

WRAMPmon Commands

`load`	Load an S-Record into RAM
`go [address]`	Begin executing a program
`dis [start_address [end_address]]`	Disassemble instructions from memory
`vm [start_address [end_address]]`	View memory contents
`sm <address> <value>`	Set the value of a memory location
`vr [reg]`	View register contents
`sr <reg> <value>`	Set register contents
`sb <address>`	Set a breakpoint
`vb`	View current breakpoints
`rb <address>`	Remove a breakpoint
`cont`	Continue executing a program
`s`	Step
`so`	Step over
`about`	Display information about this system
`help` or `?`	Show this help screen