TinyRISC project

Volume 1

Table of Contents

[Introduction 4](#_Toc86781274)

[Project Outline and goals 4](#_Toc86781275)

[Architectural outline 4](#_Toc86781276)

[The TinyRISC machine model: Registers 6](#_Toc86781277)

[General Purpose Registers 6](#_Toc86781278)

[Registers GP0-GP6 6](#_Toc86781279)

[Auxiliary registers 6](#_Toc86781280)

[Program Counter 7](#_Toc86781281)

[Status register 7](#_Toc86781282)

[Stack pointer 7](#_Toc86781283)

[LINK register 8](#_Toc86781284)

[MEM register 8](#_Toc86781285)

[The TinyRISC machine model: Instruction syntax 9](#_Toc86781286)

[R-type Instructions 9](#_Toc86781287)

[I-type Instructions 9](#_Toc86781288)

[B-type Instructions 10](#_Toc86781289)

[Instruction listing: 11](#_Toc86781290)

[Logical operations 13](#_Toc86781291)

[Logical AND 13](#_Toc86781292)

[Logical OR 14](#_Toc86781293)

[Logical XOR 14](#_Toc86781294)

[Logical NAND 14](#_Toc86781295)

[Shift Left 14](#_Toc86781296)

[Rotate Left 15](#_Toc86781297)

[Shift Right 15](#_Toc86781298)

[Rotate Right 15](#_Toc86781299)

[Set Bit Low 15](#_Toc86781300)

[Set Bit High 16](#_Toc86781301)

[Mathematical operations 16](#_Toc86781302)

[ADD with Carry: 16](#_Toc86781303)

[Subtract with carry: 16](#_Toc86781304)

[Jump Operations 17](#_Toc86781305)

[Jump Direct 17](#_Toc86781306)

[Jump Offset Register 17](#_Toc86781307)

[Jump Offset Immediate 18](#_Toc86781308)

[Jump and link 18](#_Toc86781309)

[Branch operations 19](#_Toc86781310)

[Branch on carry 19](#_Toc86781311)

[Branch on negative 19](#_Toc86781312)

[Branch on zero 19](#_Toc86781313)

[Branch on overflow 20](#_Toc86781314)

[Branch on carry and link 20](#_Toc86781315)

[Branch on negative and link 20](#_Toc86781316)

[Branch on zero and link 20](#_Toc86781317)

[Branch on overflow and link 21](#_Toc86781318)

[Memory operations 21](#_Toc86781319)

[Load byte 21](#_Toc86781320)

[Load Immediate 21](#_Toc86781321)

[Store byte 22](#_Toc86781322)

[Move 22](#_Toc86781323)

[Stack operations 22](#_Toc86781324)

[POP Stack 22](#_Toc86781325)

[PUSH stack 22](#_Toc86781326)

[PUSH Stack Immediate 23](#_Toc86781327)

[Program Counter (PC) 25](#_Toc86781328)

[High-level diagram 25](#_Toc86781329)

[Circuit diagram 26](#_Toc86781330)

[Board diagram and 3D view 26](#_Toc86781331)

[Arithmetic and Logic Unit (ALU) 28](#_Toc86781332)

[High-level diagram 31](#_Toc86781333)

[Circuit Diagram 32](#_Toc86781334)

[Board diagram and 3D view 32](#_Toc86781335)

[General Purpose Register File 33](#_Toc86781336)

[High-level diagram 34](#_Toc86781337)

[Circuit Diagram 34](#_Toc86781338)

#### Chapter 1: An Overview

# Introduction

Project TinyRISC is an attempt to build a simple processor based off off-the-shelf logic (74 series), with the final goal being to have a functional computer capable of executing programs and displaying an image onto a screen.

Previous attempts failed in part due to being overly ambitious; goals for 32bit word size and deep pipelines resulting in unmanageable complexity at this scale. We see it as necessary to constrain this project from the get-go to ensure a greater chance of success.

# Project Outline and goals

The goal of this project is to create a computer using discrete off-the-shelf logic chips of the likes of the Texas Instrument 74-series of TTL and CMOS chips. In order to avoid ballooning complexity we will strive to reduce both chip and in particular board count, and ideally, we would like the CPU itself to fit on a series of 10\*10cm PCBs (which are inexpensive) that we can easily stack.

Phase 1 of the project will revolve around building the actual CPU portion of the build (capable of reading and writing to memory according to a specified program).

Phase 2 will then require the development of a simple Assembler program to assist in future software development.

Phase 3 will revolve around the construction of a video display unit as well as additional I/O devices/adapters (we would like to be able to plug a keyboard into the computer eventually).

# Architectural outline

As the project name implies, TinyRISC is a simple processor based on a RISC-inspired architecture, heavily inspired by MIPS-type architectures, with design elements from the 6502 and Z80. It will feature the following:

**“Big-A” (ISA-level) details:**

* 8-bit word size and 16-bit instructions
* 16-bit flat address space
* Von-Neuman organization, Little-Endian
* 7 General Purpose addressable registers, in addition to Program Counter, Link register, Stack Pointer and status register
* 32 instructions (5-bit opcode)

**“Small-A”** (**implementation-level)** **details:**

* In-order execution
* 3-stage pipeline
* Clock speed on the order of a couple MHz
* Static logic will be used to scale clock speeds down to any arbitrary frequency

Additionally, for expansion’s sake, the processor should be able to address external devices or co-processors, such as video adapters with their own RAM or IO devices.

#### Chapter 2: Programming model

# The TinyRISC machine model: Registers

The TinyRISC ISA defines a set of 10 addressable registers:

|  |  |  |
| --- | --- | --- |
| Register | ID | Width (bits) |
| GP0 | 0 | 8 |
| GP1 | 1 | 8 |
| GP2 | 2 | 8 |
| GP3 | 3 | 8 |
| GP4 | 4 | 8 |
| GP5 | 5 | 8 |
| GP6 | 6 | 8 |
| AUX | 7 |  |
|  |  |  |
| LINK register | var = b00 | 16 |
|  |  |  |
| Stack Pointer | var = b10 | 16 |
|  |  |  |
| Status Register | var = b01 | 8 |
|  |  |  |
| MEM Register | var = b11 | 8 |

## General Purpose Registers

General purpose registers can store any arbitrary string of 8 bits (a byte) and can be **written to** and/or **read** **from** as the programmer sees fit, providing the instruction allows it.

### Registers GP0-GP6

General-purpose 8-bit registers. Can be used for nearly anything. Are 8 bits wide. No specific constraints.

## Auxiliary registers

AUXiliary registers comprise all registers which aren’t consider general purpose. This includes the program counter, the status register, and the stack pointer. These registers mainly keep track of the operation of the computer itself, though they can be accessed by the programmer if need be. Because some of these registers are wider than the machine’s native word size, accessing some of these registers may result in temporary stalling while the multi-word operation is performed.

### Program Counter

The Program counter, or PC for short (also known as the Instruction pointer in x86 nomenclature) is a 16 bit register which contains (at the beginning of a given cycle) the address of the instruction to be executed. Over the course of the cycle this value is updated to reflect the address of the next instruction in line for execution.

Note: this value can be changed via Jump Instructions only. If need be, it can be read by Performing a Branch/Jump “and Link” instruction, which inserts it into the LINK register

### Status register

The Status Register, or SR for short, is an 8-bit register containing data relevant to the processor status, such as info pertaining to the last arithmetic operation (overflow, carry, negative flags) or to current operation (whether or not interrupts are masked, for example).

The Status Register is read-write (specifically to access/modify arithmetic status bits), though special care should be taken when modifying bits not pertaining to arithmetic.

**Reading** the SR can be done through regular register-register type instructions by pointing to register ID 7 (the AUX register) and passing [1]10 = [01]2 to the **var** instruction field.

**Writing** to the SR can be done using a register-register instructions by pointing to register ID 7 and passing [01]10 = [01]2 to the **var** instruction field.

### Stack pointer

The Stack Pointer, or SP for short, is a 16-bit register containing the address of the top-most element of the stack. PUSHing an element onto the stack will thus increment the stack pointer while POPing the stack will always decrement it. It can be addressed using register ID [7]10 = [111]2 and by setting the **var** field to [11]2 (which yields the lowest byte)

Note: The SP always points to an *occupied* memory address.

Note: Proper operation of the stack requires that the SP be initialized at startup (the stack base should be set).

### LINK register

16-bit register. When **a Jump and link** or **Branch and Link** instruction is executed and any branch conditions are met (assuming there are any), this register will contain the PC of the instruction which caused the jump.

Note: The LINK register can be addressed by passing register ID 7 and setting **var** to [00]2.

### MEM register

8-bit register. When a “Store byte” operation is called this register is implied to be the supplier of the data to be committed to memory (see additional info in instruction reference).

The rational for such a system (compared to having an instruction capable of encoding both the destination address and the source register) comes from a combination of limited instruction encoding space and limited register file bandwidth (particularly on the read port).

For the sake of limiting complexity, the General-Purpose Register File in this implementation of TinyRISC is limited to having 2 8-bit read ports and 1 8-bit write port. In conjunction to this, the ISA’s 16-bit limit on instructions makes it impossible to encode addresses directly into addresses, to say nothing of computing addresses dynamically; the “address” argument of a memory instruction is thus a result of concatenating the contents of two registers, which necessarily saturates the GPRF’s read ports, leaving no more bandwidth for the actual data to store.

While a possible solution could be to stall the pipeline and access this information over two or more cycles, having a separate implicit register for memory writes was seen as being easier to implement.

Note: The MEM register can be addressed by passing register ID 7 and setting **var** to [11]2.

# The TinyRISC machine model: Instruction syntax

TinyRISC uses fixed length, 16bit instructions to encode operations. The TinyRISC ISA itself defines up to 32 separate instructions (5-bit fixed-length opcode field).

The available instructions can be loosely classed into three distinct categories: Register-Register types (“R-type”) Immediate-Register types (“I -types”) and Bit-manipulation types (“B-type”)

Their following formats are as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |
| R-type | opcode [5] | | | | | Rd [3] | | | Rs [3] | | | Ro [3] | | | var [2] | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| I-type | opcode [5] | | | | | Rd [3] | | | Immediate [8] | | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B-type | opcode [5] | | | | | Rd [3] | | | Immediate [6] | | | | | | var [2] | |  |

## R-type Instructions

R-type instructions are all instructions where one or all operands both originate and terminate from one of the processor’s registers. These instructions comprise the bulk of available instructions.

R-type instructions which take two operands will generally operate as follows:

Where **Rd** is the ID of the destination (where the result is stored), **Rs** is the “source” operand (one of the two operands and **Ro** is the other, with ^ being the stand-in for the desired operator (*such as an ADD* operator).

*Instruction fields:*

* **Opcode:** <*5 bits>* contains the unique identifier of the instruction
* **Rd:** <3 bits> contains the Register ID of the register acting as a destination
* **Rs/Ro:** <3 bits> contains the Register ID of the registers acting as operands
* **Var:** <2 bits> variable field containing data specific to instruction being executed; no fixed function. See instruction and register listings for detail.

## I-type Instructions

I-type instructions are instructions which process an immediate (a number) as one the operands. These instructions permit hard-coded information to be passed as operands.

I-type instructions only have a 3-bit Register ID field due to encoding space constraints (it was seen as necessary for the immediate to be a word wide). Additionally, this instruction type does not feature a **var** field. This restricts the range of addressable registers to general purpose registers GP0 – GP7, with no possibility of referencing AUXiliary registers.

I-type instructions which take two operands will generally operate as follows:

Rd = Rs ^ Imm

Where Rs and Rd are the source and destination register respectively, Imm is the immediate value being passed, and ^ is the operator specified by the opcode.

*An example: we wish to load [8C]16 into register GP0. This operation does not specify a source operand but does specify a destination register. A pseudocode representation would thus be:*

R0 = Imm

After execution, register GP0 should contain value *[8C]16 regardless of previous register contents.*

*Instruction fields:*

* **Opcode:** <*5 bits>* contains the unique identifier of the instruction
* **Rs/Rd:** <3 bits> contains the Register ID of the register acting as source operand and destination.
* **Immediate:** <8 bits> contains the Immediate value used as operand.

## B-type Instructions

B-type instructions were specifically designed for bit manipulation instructions such as Set Bit Low and Set Bit High, sacrificing encoding space usually reserved for the immediate to expand the range of addressable registers via the addition of a **var** field.

*Instruction fields:*

* **Opcode:** <*5 bits>* contains the unique identifier of the instruction
* **Rd:** <3 bits> contains the Register ID of the register acting as a destination
* **Immediate:** <6 bits> contains the immediate acting as an operand
* **Var:** <2 bits> variable field containing data specific to instruction being executed; no fixed function. See instruction and register listings for detail.

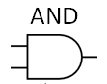
# Instruction listing:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Instruction name | Instruction class | Opcode (dec) | Opcode (hex) | Desciption | Notes |
| Logical | AND | R-type | 01 | 1 | Logical AND |  |
| OR | R-type | 02 | 2 | Logical OR |  |
| XOR | R-type | 03 | 3 | Logical XOR |  |
| NAND | R-type | 04 | 4 | Logical XOR |  |
| SHIFT left | R-type | 05 | 5 | left shift |  |
| ROTATE left | R-type | 07 | 7 | left rotate (wraparound) |  |
| SHIFT right | R-type | 06 | 6 | right shift |  |
| ROTATE right | R-type | 08 | 8 | right rotate (wraparound) |  |
| Set Bit LOW | B-type | 11 | B | Bitwise Set bit to 0 |  |
|  | Set Bit HIGH | B-type | 15 | F | Bitwise Set bit to 1 |  |
|  |  |  |  |  |  |  |
| Math | ADD with carry | R-type | 09 | 9 | Adds two numbers with carry-in |  |
| SUBTRACT with carry | R-type | 10 | A | subtracts two numbers with borrow |  |
|  |  |  |  |  |  |  |
| Jump | Jump direct | R-type | 19 | 13 | Directs PC to address in register |  |
| Jump offset register | R-type | 20 | 14 | Directs PC to itself + register |  |
| Jump offset immediate | I-type | 21 | 15 | Directs PC to itself + immediate |  |
| Jump and link | R-type | 22 | 16 | Directs PC to address in register and loads last PC onto stack | See section of subroutine calls and consequence on PC |
|  |  |  |  |  |  |  |
| Branch | Branch on Carry | R-type | 16 | 10 | If carry flag == 1, branch to address in register |  |
| Branch on negative | R-type | 17 | 11 | If negative flag == 1, branch to address in register |  |
| Branch on carry and link | R-type | 24 | 18 | If carry flag == 1, branch to address in register and load last PC onto stack | See section of subroutine calls and consequence on PC |
| Branch on negative and link | R-type | 25 | 19 | If negative flag == 1, branch to address in register and load last PC onto stack | See section of subroutine calls and consequence on PC |
| Branch on overflow | R-type | 19 | 13 |  |  |
| Branch on overflow and link | R-type | 27 | 1B |  |  |
| Branch on zero | R-type | 18 | 12 | If zero flag == 1, branch to address in register |  |
| branch on zero and link | R-type | 26 | 1A | If carry flag == 1, branch to address in register and load last PC onto stack | See section of subroutine calls and consequence on PC |
|  |  |  |  |  |  |  |
| Memory | Load Byte | R-type | 29 | 1D | Loads byte from memory address in register to register | See section on memory addressing |
| Load Immediate | I-type | 30 | 1E | Load immediate value to register |  |
| Store Byte | R-type | 31 | 1F | Stores value in operator register to memory address in source register | See section on memory addressing |
| Move | R-type | 00 | 0 | Moves value between two registers |  |
|  |  |  |  |  |  |  |
| Stack | Pop stack | R-type | 12 | C | Decrements SP | See sections on stack operation |
| Push stack | R-type | 13 | D | Increments SP and and stores value in register into stack |
| Push stack immediate | I-type | 14 | E | Increments SP and and stores immediate value into stack |
|  |  |  |  |  |  |  |
| Misc | No-Op (NOP) | R-type | 00 | 0 | Performs no operation | Encoded as a data move with source as destination -> no net effect |
| Get High Byte | R-type | 23 | 17 | Moves Highest-order byte of operand register to destination register. For use with 16-bit registers such as SP or LINK. |  |
| Set High Byte | R-type | 28 | 1C | Sets Highest Order byte of destination register to match the contents of operand register. For use with 16-bit registers such as SP or LINK. |  |

## Logical operations

**Logical Operations** are all instructions which rely on bitwise manipulation of data and not words as wholes.

### Une image contenant table Description générée automatiquementLogical AND

Bitwise AND operation. Compares bits of same rank in both operands and returns 1 if they’re both 1, 0 otherwise.

***R-type instruction.***

***Opcode .***

### Une image contenant table Description générée automatiquementLogical OR

Une image contenant texte, sport athlétique, sport

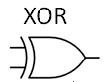
Description générée automatiquementBitwise OR operation. Compares bits of same rank in both operand and returns 1 if either (or both) operands are 1, 0 if otherwise.

***R-type instruction.***

***Opcode .***

### Logical XOR

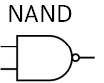
Une image contenant table

Description générée automatiquementBitwise XOR operation. Compares bits of same rank in both operands and returns 1 if one *or* the other is 1 and will return 0 otherwise.

***R-type instruction.***

***Opcode .***

### Une image contenant table Description générée automatiquementLogical NAND

Bitwise NAND operation. Compares bits of same rank in both operands and returns 0 if both bits are 1 and will return 0 otherwise.

**Note**: NAND gates are universal logic gates; they can be strung together to simulate any other gate.

***R-type instruction.***

***Opcode .***

### Shift Left

Logical shift left. Each bit at rank *n* is moved to rank *n+1, effectively moving from lesser bit positions to more impactful ones (“leftwards”).*

Bits of previously uncovered rank (eg bit -1 🡪 appearing as LSB after a single left shift) will default to zero. *Ex: b01001100 << 1 = b10011000*

**Note**: A single logical left shift of an unsigned number is equivalent to a multiplication by a factor of 2. By extension, a left shift by *k* places is equivalent to a multiplication by .

**Note**: effect on flags

* + - Carry flag set to be bit which was shifted out

***R-type instruction.***

***Opcode***

### Rotate Left

Logical Rotate left. Similar to logical left shift with the caveat that bits that that overflow (previously the MSB) are piped by to the least significant positions. Ex: b10010001 ROTL 1 = b00100011.

**Note**: effect on flags

* + - Carry flag set to be bit which was shifted out

***R-type instruction,***

***Opcode***

### Shift Right

Logical shift right. Each bit at rank *n* is moved to rank *n-1, effectively moving from higher bit positions to lesser ones (“rightwards”).*

Bits of previously uncovered rank (e.g. bit 8 🡪 appearing as MSB after a single right shift) will default to zero. *Ex: b01001100 >>1 = b00100110*

**Note**: A single logical right shift of an unsigned number is equivalent to a division by a factor of 2. By extension, a right shift by *k* places is equivalent to a division by .

**Note**: effect on flags

* + - Carry flag set to be bit which was shifted out

***R-type instruction.***

***Opcode***

### Rotate Right

Logical Rotate Right. Similar to logical right shift with the caveat that bits that that overflow (previously the LSB) are piped by to the most significant positions. Ex: b10010001 ROTR 1 = b11001000

**Note**: effect on flags

* + - Carry flag set to be bit which was shifted out

***R-type instruction,***

***Opcode***

### Set Bit Low

Sets a bit in the destination register to 0. The bit’s order can be set depending on the immediate value passe as an argument.

Example: *SBL [111]2,[000000]2,[01]2* Sets The Status Register’s LSB to 0.

Note: Implemented Internally as AND operation.

***B-type instruction.***

***Opcode***

### Set Bit High

Sets a bit in the destination register to 1. The bit’s order can be set depending on the immediate value passe as an argument.

Example: *SBL [111]2 ,[000111]2,[01]2* Sets the Status Register’s MSB to 1

Note: Implemented Internally as OR operation.

***B-type instruction.***

***Opcode***

## Mathematical operations

**Math operations** are instructions which execute algebraic operations on words as wholes.

### 

### ADD with Carry:

Addition operation which returns sum of two operands and a carry in (from status register):

Note: effect on flags

* Sets **Carry**flag if unsigned sum exceeds 255
* Sets overflow flag if carry is different from 7th bit of sum )
* Sets Negative flag:

**R-type instruction. Opcode [09]16.**

### Subtract with carry:

Subtraction operation which returns the difference of two operands, minus the complement of the carry flag (from status register).

**Note:** Treated internally as

**Note:** effect on flags

* + - Sets Carry flag to
    - Sets Overflow flag:
    - Sets Negative flag:

**R-type instruction. Opcode [10]16.**

## Jump Operations

**Jumps operations** are instructions which redirect the Program Counter to a specified target without any attached condition; they always execute regardless of machine state. As a rule, the next cycle’s PC will be the exact value passed on by these instructions (no implicit offsetting).

**Note:** Some of these instructions require 16-bit arguments to be passed (as is the case when handling addresses within registers) around despite the processor’s 8-bit arithmetic: two separate registers are thus required to store the complete set of bits. We’ll define the operator as concatenating two values into one, with the first operand’s bits taking the place of the final word’s more significant bits

### Jump Direct

A direct jump directs the PC to the value currently passed in the register passed as a source.

**R-type Instruction. Opcode [13]16**

### Jump Offset Register

A direct jump directs the PC to a sum/difference of its previous self + the contents of the register passed as an argument.

**Note**: Offset value is limited to an 8-bit value per register size constraints

**Note**: When performing addition between the current PC and the register contents, the missing 8 most significant bits of Rs’ translation into a 16bit value are assumed to be 0 à is filled in to become for the sake of addition. However, these bits can be changed to be all 1s for the purpose of subtracting said offset. Subtracting a positive offset is then simply a matter of using 2s complement to obtain said offset’s inverse and setting the leading bits to be 1.

*Ex*ample: we wish to subtract 45 from the PC. . 45’s complement (-45) is in 2s complement format. By setting all leading bits to 1, we’re asking to add the offset à a subtraction

Toggling the value of the tacked-on bits is done though the **var** field: they are 0 if and 1 if . Only **var**’s LSB is considered, the others are ignored.

**R-type Instruction. Opcode [14]16**

### Jump Offset Immediate

A direct jump directs the PC to a sum/difference of its previous self and the immediate passed as an argument

**Note:** Offset size is limited to 8 bits per immediate field size constraints

**Note**: When performing addition between the current PC and the immediate, the missing 8 most significant bits of the Immediate’s translation into a 16bit value are assumed to be 0 à is filled in to become for the sake of addition. However, these bits can be changed to be all 1s for the purpose of subtracting said offset. Subtracting a positive offset is then simply a matter of using 2s complement to obtain said offset’s inverse and setting the leading bits to be 1, just as with the **Jump Offset Register** instruction.

Toggling the value of the tacked-on bits is done though the **var** field: they are 0 if and 1 if . Only **var**’s LSB is considered, the others are ignored.

**R-type Instruction. Opcode [15]16**

### Jump and link

A direct jump to the address present in the concatenation of both registers. **Additionally, the current PC is pushed onto the stack.**

**R-type instruction. Opcode [16]16.**

## Branch operations

Branch operations are similar in concept to jump instructions with the exception that their execution is conditional. This enables the program flow to change dynamically depending on program operation.

### Branch on carry

Jumps to address present in registers if the carry flag is currently set to 1. Continues to PC+2 otherwise.

**If C = 1:**

**If C = 0 (fall-through):**

**R-type instruction. Opcode [10]16**

### Branch on negative

Jumps to address present in registers if the Negative flag is currently set to 1. Continues to PC+2 otherwise.

**If N = 1:**

**If N = 0 (fall-through):**

**R-type instruction. Opcode [11]16**

### Branch on zero

Jumps to address present in registers if the Zero flag is currently set to 1. Continues to PC+2 otherwise.

**If Z = 1:**

**If Z = 0 (fall-through):**

**R-type instruction. Opcode [12]16**

### Branch on overflow

Jumps to address present in registers if the Overflow flag is currently set to 1. Continues to PC+2 otherwise.

**If O = 1:**

**If O = 0 (fall-through):**

**R-type instruction. Opcode [13]16**

### Branch on carry and link

Jumps to address present in registers if the carry flag is currently set to 1, and *then pushes current PC to stack*. Continues to PC+2 otherwise.

**If C = 1:**

**If C = 0 (fall-through):**

**R-type instruction. Opcode [18]16**

### Branch on negative and link

Jumps to address present in registers if the Negative flag is currently set to 1, and *then pushes current PC to the stack*. Continues to PC+2 otherwise.

**If N = 1:**

**If N = 0 (fall-through):**

**R-type instruction. Opcode [19]16**

### Branch on zero and link

Jumps to address present in registers if the Zero flag is currently set to 1, and then *pushes current PC to the stack.* Continues to PC+2 otherwise.

**If Z = 1:**

**If Z = 0 (fall-through):**

**R-type instruction. Opcode [1A]16**

### Branch on overflow and link

Jumps to address present in registers if the Overflow flag is currently set to 1 and then *pushes current PC to the stack*. Continues to PC+2 otherwise.

**If O = 1:**

**If O = 0 (fall-through):**

**R-type instruction. Opcode [1B]16**

## Memory operations

**Memory instructions** are all instructions which the pure transfer if data between two mediums (register-register, register-memory, or memory-register) and don’t involve the stack explicitly (see stack instructions and stack operation for more detail).

### Load byte

Moves a single byte (8 bits) of data from a memory location stored in register operands to the specified destination register.

**R-type instruction. Opcode [1D]16**

### Load Immediate

Moves an immediate value into the specified destination register.

This instruction is useful for loading constants into registers.

**I-type instruction. Opcode [1E]16**

### Store byte

Moves the contents of MEM register into a memory location specified by operand registers.

Note: See info on MEM register for more explanation regarding the rational for this instruction.

**R-type instruction. Opcode [1F]16**

### Move

Transfers the contents of 1 register into another.

Note: Contents of Ro are technically irrelevant here, as the input/output of the register being pointed to will be disabled. It should not be register 7/AUX however, which means direct transfers between AUX registers are not possible and should instead use an intermediate register.

**R-type instruction. Opcode [00]16**

## Stack operations

TinyRISC supports the use of a hardware stack via its Stack Pointer register. The ISA defined a few instructions allowinga programmer to PUSH (add data onto the stack) and POP (remove data from the stack) data as they please.

**A note on the stack:** TinyRISC’s stack uses an ascending, full-stack convention. This means that PUSH operations will result in the Stack Pointer incrementing by 1, while a POP operation will decrement it by 1 (hence the term “ascending”). Additionally, at any given moment, the Stack Pointer register will contain the address of the stack’s *topmost* element, **NOT** the topmost empty slot.

### POP Stack

POPs the topmost element of the stack, effectively removing it. Internally, this is simply done by decrementing the Stack Pointer by 1. The previous topmost element, while still present in memory, is effectively removed from the stack as a result.

Note: None of the operand registers do anything here.

**R-type instruction. Opcode [0C]16**

### PUSH stack

PUSHes the data in the operand register onto the stack. Internally, the processor does the following:

* + - Increments the stack pointer by 1
    - Stores the contents of register Rs to the memory location pointed to by the new SP.

**R-type instruction. Opcode [0D]16**

### PUSH Stack Immediate

PUSHes the Immediate data onto the stack. Internally, the processor does the following:

* + - Increments the stack pointer by 1
    - Stores the immediate value to the memory location pointed to by the new SP.

**I-type instruction. Opcode [0E]16**

## Miscellaneous Instructions

Miscellaneous instructions are all instructions which don’t fit into the previously defined categories. This is only a semantics issue; the expected behaviour is as described.

### NOP instruction

**NOP**, or **No-Operation instruction.** This instruction has no effect on the processor state and is thus ideal when one needs to create some execution delay or pad-out space.

**Note:** processed internally as a MOVE instruction where the destination and operand registers are the same (although the choice of register is arbitrary, it should not be one of the AUX registers).

**R-type instruction. Opcode [00]16**

### Get High Byte

Some of TinyRISC’s available AUX registers are 16-bits wide, separated over 2 bytes (noted high and low for the most and least significant portions of the data, respectively). By default, any instruction referencing one of these 16-bit registers will only access the lower byte of the value (ex: MOVE R1, SP will move SP\_low’s contents to R1).

This instruction effectively acts as a move instruction that specifically targets the high-order byte of the destination operand. This implies that Rs should be one of the 16-bit AUX registers (SP or LINK) and not any of the 8-bit registers, lest one should encounter undocumented behaviour.

**R-type instruction. Opcode [17]16**

### Set High Byte

Similar semantics to “Get High Byte” instruction.

This instruction effectively acts as a move instruction that specifically targets the high-order byte of the destination operand. This implies that Rd should be one of the 16-bit AUX registers (SP or LINK) and not any of the 8-bit registers, lest one should encounter undocumented behaviour.

**R-type instruction. Opcode [1C]16**

#### Chapter 3: High-level hardware overview

#### Chapter 4: The parts

## Program Counter (PC)

Description: Computes the address of the instruction to be next executed. Capable of jumping to any arbitrary yet valid (16bit space) address via offsetting or direct placement (see available instructions).

Functional requirements:

* 16-bit data width (one address) for storage
* Capable of computing
  + - à normal instruction hopping
    - à offset jump
    - à direct jump
* Capable of stalling: during stalls the PC’s input does not propagate
* Clocked

I/O:

* PC\_OUT [16] (output of PC module)
* JMP\_TGT16] (offset or absolute target of jump if applicable)
* [1] (control signal: whether or not PC values propagate à stalls)
* JMP\_EN[1] (control signal: whether or not next PC is incremented or if offset/target is to be used instead)
* JMP\_ABS[1] (control signal: whether or not next PC value is absolute target or PC[n] + k)
* REL\_SIGN (control signal: dictates whether the bits affixed to the end of the 8-bit PC offset are 1s or 0s).
* CLK\_IN[1] (input clock signal)

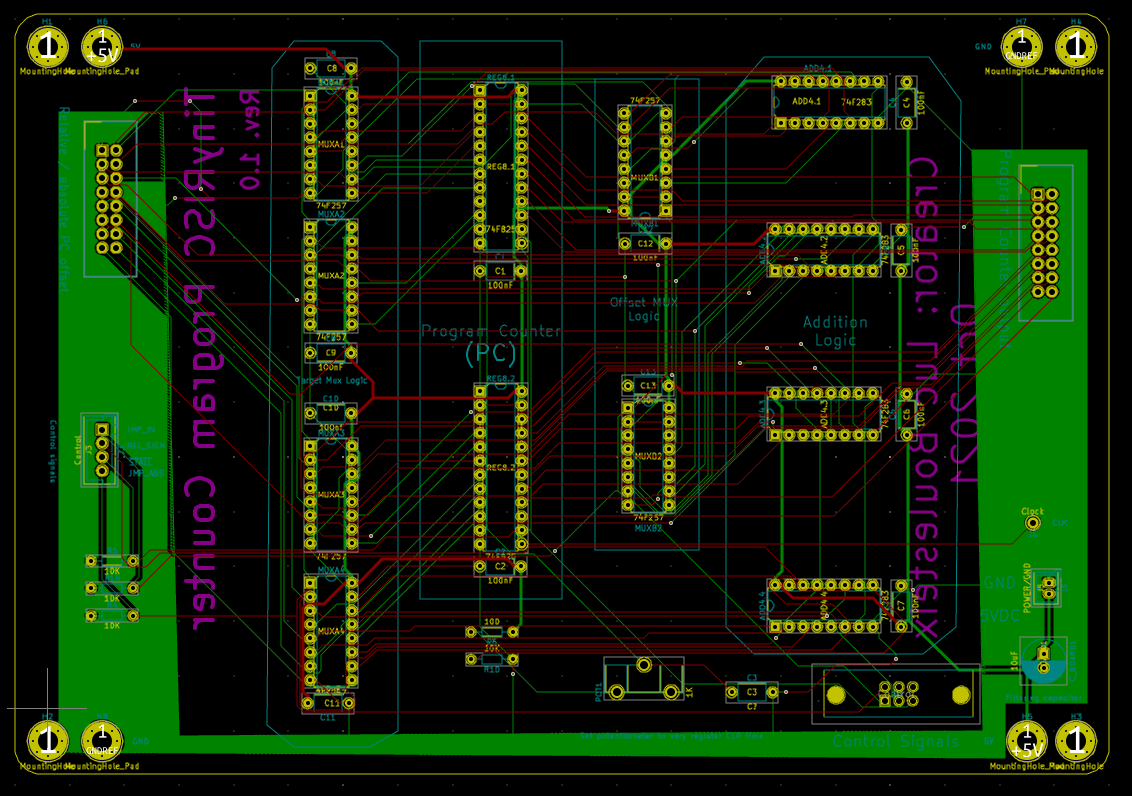
### High-level diagram

Chip candidates:

|  |  |  |  |
| --- | --- | --- | --- |
| Adder | 74F283 (x4)  () |  |  |
| Mux | 74HC(T)157 (x6) | 74HCT257 (x6) T | 74F257 (x6) ( |
| Register | 74HC377 (x2) | 74F377 (x2) | 74F825 (x2) à ideal pinout |

### Circuit diagram

### Board diagram and 3D view



Une image contenant texte, équipement électronique

Description générée automatiquement

## Arithmetic and Logic Unit (ALU)

Description: Performs a variety of arithmetic and logic instructions for the processor and handles the bulk of the processor’s EXEcution capabilities. This subunit takes in two binary operands and an ALU command code, and then returns the result, along with a few status bits which inform the processor on the manner of execution which took place.

Supported instructions are as follows (see instruction set section for more information)

* Addition with carry
* Subtraction with carry
* Shift left by 1 position
* Shift right by 1 position
* Rotate left by 1 position
* Rotate right by 1 position
* Bitwise AND
* Bitwise OR
* Bitwise XOR
* Bitwise NAND

Note: shift operations allow for the value of the bit being shifted in to be adjusted via a control pin

Note: See instruction set section for information on flag behaviour

Note: The ALU’s adder sub-unit is 16-bits wide. This is to allow for single-cycle arithmetic on addresses, such as for SP incrementation and decrementation. It is limited to 8 bits for general arithmetic because of GP register size.

I/O:

* Op1[16] (input for operand 1 of the ALU)
* Op2[16] (input for operand 2 of the ALU)
* Res[16] (output of ALU)
* ALU\_COM[4] (input for ALU command codes, see reference sheet)
* C\_in[1] (input: carry in bit, for add and subtract operations)
* n\_bit[1] (input: shift-in bit for shift operations)

ALU command codes:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Code (binary) | Operation | Code (binary) |
| Rotate left | b0000 | AND | b100X |
| Shift left | b0001 | OR | b010X |
| Rotate Right | b0010 | XOR | b101X |
| Shift Right | b0011 | NAND | b011X |
| Add with C | b1100 |  |  |
| Subtract with C | b1101 |  |  |

The ALU can execute 12 distinct instructions, hence the use of a 4-bit ALU code to communicate every instruction type. Seeing as though the ALU command code LSB is meant to inform the operation of a particular sub-unit (e.g. whether to add or subtract for the ADDER), we can concentrate on the 3 most significant bits to generate an enable signal for each sub-unit (there are 7 sub-units accomplishing one or more functions, e.g. the Adder, which can add OR subtract)

We will denote the ALU command code as having the structure ABCD, where D is the LSB, and A is the MSB.

In order to generate the appropriate control signals for each sub-unit from the available codes we will use a 74F138 chip to generate the appropriate active low signals, with the following truth table:

Une image contenant table

Description générée automatiquement

The ALU command codes are

We will ignore situations where the enable bits are invalid, which leave us with the following:

* We wish to active the ADDER when ABC = 110 = HHL 🡪 We will connect to Y3
* We wish to active the left shifter when ABC = 000 = LLL 🡪 we will connect to Y0
* We wish to active the right shifter when ABC = 001 = LLH 🡪 we will connect to Y4
* We wish to active the AND array when ABC = 100 = HLL 🡪 We will connect Y1
* We wish to active the OR array when ABC = 010 = LHL 🡪 We will connect Y2
* We wish to active the XOR array when ABC = 101 = HLH 🡪 we will connect Y5
* We wish to active the NAND array when ABC = 011 = LHH 🡪 we will connect Y6

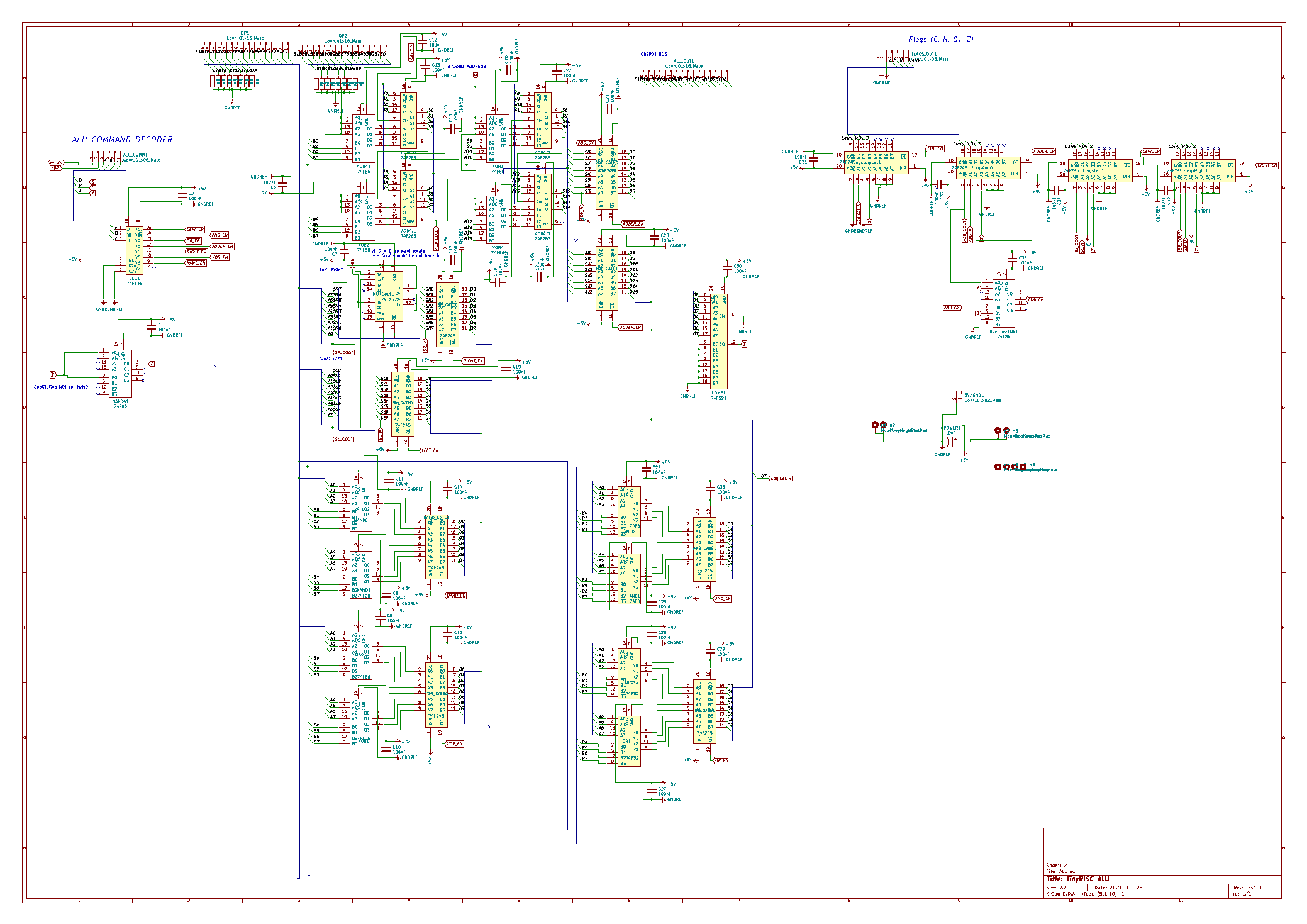
We note that all command codes for logical operations are set up such that A XOR B = 1 (both bits are different), hence we can use a XOR gate to determine whether an ALU code is requesting a logical operation in a general sense.

### High-level diagram

Chip candidates:

|  |  |
| --- | --- |
| Adder | 74F283 (x4) |
| Transceiver | 74F245 (x8) |
| 1b MUX | 74F157 |
| AND | 74F08 (x2) |
| OR | 74F32 (x2) |
| XOR | 74F86 (x2) |
| NAND | 74F00 (x2) |
| Invert | 74F04 |
| Comparator | 74F521 (x1) |

### Circuit Diagram



### Board diagram and 3D view

## General Purpose Register File

Description: The computer’s General-Purpose register File is the structure containing the bulk of the machine’s current state, namely in the form of loaded variables. Most of the processor’s instructions grab their operands from this structure.

Functional requirements:

* 7 8-bit registers, named 0 through 6.
* 2 reads and 1 write per cycle (to accommodate one instruction’s read and another’s writeback).
* If a simultaneous read and write both try to access the same register, the read should get the data being written.
* We want to be able to enable/disable each port and place it into High-Z mode if not needed.

Design implementation:

Approach: we will use latches to store data. Each read port will have its own set of latches (the RF is essentially doubled up), but are all connected to the same write port. When we want to write to a latch, a decoder will active the LATCH ENABLE control input on the desired pair of latches. When we want to read from a latch, we will pull low the Output Enable Pin for the latch on the port we wish to see the data on.

This approach was seen as being brute-force and expensive but overall, less complex and costly than having a set of MUXes.

I/O:

* Write Port 1 [8] (data input)
* Read Port 1 [8] (Read output 1)
* Read Port 2 [8] (Read output 2)
* Write ID [3] (ID of register to write to)
* Read 1 ID [3] (ID of register at read port 1)
* Read 2 ID [3] (ID of register at read port 2)
* Write Enable [1] (Enable Write Port)
* Read1 Enable [1] (Enable reading from Read 1 port)
* Read2 Enable [1] (Enable Reading from Read 2 Port)
* Register 7 LE [1] (output of the write decoder for register 7)
* (output of the read decoder for port 1 for register 7)
* (output of the read decoder for port 2 for register 7)

Chip Candidates:

|  |  |
| --- | --- |
| Latches | 74F573 (x14) |
| Decoder | 74F138 (x3) |
| Inverters | 74F04 (x2) |

### High-level diagram

### Circuit Diagram

### Board view and 3D image

## AUXiliary register file

Description: TinyRISC’s general-purpose register file is designed to handle the brunt of data traffic for most operations. However, it is limited in bandwidth (2 Read + 1 Write) and purpose (it is only ever referenced explicitly). We would like to create a separate register file to store more specialized data will affording us extra READ/WRITE ports for things such as Store operations. We would also like it to be able to output constants -1, 0, 1 and 2 for use as operands for operation such as stack push/pops (which require implicite operands).

Functional requirements:

* Should house the following registers (width between brackets)
  + - Stack Pointer [16]
    - LINK [16]
    - MEM [8]
    - Status [8]
* Each register should be able to be blocked at the input stage to avoid unwanted changes (ex: since most operations use the ALU internally, even instructions which don’t require it explicitly such as MOVE, we don’t necessarily want to update the status register at every cycle).
* Should constant a constant generator capable of creating (at least) constants -1, 0 and +1.
* We the AUX RF is not expected to be able to provide read/write access to LINK and SP registers simultaneously (the SP *or* the LINK register is written to in a given cycle, and the same for read), we need to have simultaneous possible access (Read AND Write) to the Status register, MEM register and the constant generator’s (CGU) output.

I/O:

* READ16 [16] (read port for LINK and SP registers)
* WRITE16 [16] (write port for LINK and SP registers)
* READSTAT [8] (read port for status register)
* WRITESTAT [8] (write port for status register)
* READMEM [8] (read port for MEM register)
* WRITEMEM [8] (write port for MEM register)
* CGU\_OUT [16] (CGU output)
* CGU\_COMM [?] (instructions for CGU regarding what constant to generate)
* LINK\_WE [1] (write enable for LINK register)
* SP\_WE [1] (write enable for SP register)
* STAT\_WE [1] (write enable for Status register)
* MEM\_WE [1] (write enable for MEM register)
* LINK\_OE [1] (output enable for LINK register)
* SP\_OE [1] (output enable for SP register)
* STAT\_OE [1] (output enable for status register)
* MEM\_OE [1] (output enable for MEM register)
* CGU\_OE [1] (output enable for CGU)
* READ\_FRAC [2] (selects whether to output all of LINK/SP, low byte or high byte)
* WRITE\_FRAC [2] (selects whether to write to all of LINK/SP, low byte or high byte)