TinyRISC project

Volume 1

Table of Contents

[Introduction 3](#_Toc86101050)

[Project Outline and goals 3](#_Toc86101051)

[Architectural outline 3](#_Toc86101052)

[The TinyRISC machine model: Registers 4](#_Toc86101053)

[General Purpose Registers 5](#_Toc86101054)

[Auxiliary registers 5](#_Toc86101055)

[Program Counter 5](#_Toc86101056)

[Status register 5](#_Toc86101057)

[Stack pointer 5](#_Toc86101058)

[Interrupt ID 6](#_Toc86101059)

[The TinyRISC machine model: Instruction syntax 7](#_Toc86101060)

[R-type Instructions 7](#_Toc86101061)

[I-type Instructions 7](#_Toc86101062)

[Instruction listing: 9](#_Toc86101063)

[Logical operations 11](#_Toc86101064)

[Logical AND 11](#_Toc86101065)

[Logical OR 11](#_Toc86101066)

[Logical XOR 11](#_Toc86101067)

[Logical NAND 12](#_Toc86101068)

[Shift Left 12](#_Toc86101069)

[Rotate Left 12](#_Toc86101070)

[Shift Right 12](#_Toc86101071)

[Rotate Right 12](#_Toc86101072)

[Mathematical operations 13](#_Toc86101073)

[ADD with Carry: 13](#_Toc86101074)

[Subtract with carry: 13](#_Toc86101075)

[Jump Operations 13](#_Toc86101076)

[Jump Direct 14](#_Toc86101077)

[Jump Offset Register 14](#_Toc86101078)

[Jump Offset Immediate 14](#_Toc86101079)

[Jump and link 15](#_Toc86101080)

[Branch operations 15](#_Toc86101081)

[Branch on carry 15](#_Toc86101082)

[Branch on negative 15](#_Toc86101083)

[Branch on zero 15](#_Toc86101084)

[Branch on overflow 15](#_Toc86101085)

[Branch on carry and link 15](#_Toc86101086)

[Branch on negative and link 16](#_Toc86101087)

[Branch on zero and link 16](#_Toc86101088)

[Branch on overflow and link 16](#_Toc86101089)

[Memory operations 16](#_Toc86101090)

[Load byte 16](#_Toc86101091)

[Program Counter (PC) 20](#_Toc86101092)

[High-level diagram 20](#_Toc86101093)

[Circuit diagram 21](#_Toc86101094)

[Board diagram and 3D view 21](#_Toc86101095)

[Arithmetic and Logic Unit (ALU) 22](#_Toc86101096)

[High-level diagram 24](#_Toc86101097)

[Circuit Diagram 25](#_Toc86101098)

#### 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
* Little Endian
* 15 GP 8-bit registers in addition to Stack Pointer, Status Register, and program counter
* 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 21 separate registers:

|  |  |  |  |
| --- | --- | --- | --- |
| Register | ID | Width (bits) | notes |
| GP0 | 0 | 8 | Normal general-purpose architectural register |
| GP1 | 1 | 8 |
| GP2 | 2 | 8 |
| GP3 | 3 | 8 |
| GP4 | 4 | 8 |
| GP5 | 5 | 8 |
| GP6 | 6 | 8 |
| GP7 | 7 | 8 |
| GP8 | 8 | 8 |
| GP9 | 9 | 8 |
| GP10 | 10 | 8 |
| GP11 | 11 | 8 |
| GP12 | 12 | 8 |
| GP13 | 13 | 8 |
| GP14 | 14 | 8 |
| AUX | 15 | N/A | INVALID as a GP register; acts as an hub for auxiliary (PC, SP, STAT) registers |
|  |  |  |  |
| Stack Pointer (low) | *var* = 0, *Ri* = 15 | 8+8 | Contains address of top of stack (full) |
| Stack Pointer (high) | *var* = 1, *Ri* = 15 | Contains address of top of stack (full) |
|  |  |  |  |
| Status Register | *var* = 2, *Ri* = 15 | 8 | Contains status bits (Carry, Negative, Overflow, Zero, Interrupt enable) |
|  |  |  |  |
| Program Counter (low) | *var* = 3, *Ri* = 15 | 8+8 | Contains address of next instruction to execute |
| Program Counter (high) | *var* = 4, *Ri* = 15 |
|  |  |  |  |
| Interrupt ID | *var* = 7, *Ri* = 15 | 8 | Contains ID of Interrupt request |

## General Purpose Registers

TinyRISC defines 15 General Purpose Registers, each 8 bits wide, with IDs 0 to 14.

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.

Register ID 15 [1111]2 is an invalid register ID and does not point to a general-purpose register. It instead points to the entire set of auxiliary registers.

## 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**: Instructions which query the PC will treat the value in PC as being their **own** address.

The PC itself is broken down into two sub-sections, each 8 bits wide and addressable in instructions, named *PC\_high*and PC\_low*,* being the high and low bytes of the program counter, respectively.

A programmer can only reference the program counter through these smaller sections, and not as a whole. Access to PC subsections is only possible via R-type instructions (see section on instructions). The programmer must address these sections by using register ID 15 and by setting bits in the **var** field to either [011]2 (low) or [100]2 (high).

**Note**: The Program Counter and its registers are **read-only**. Attempting a write will yield an exception. Modifying the PC must be done through jump/branch instructions.

### 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 15 and passing [7]10 = [111]2 to the **var** instruction field.

**Writing** to the SR can be done using a register-register instructions by pointing to register ID 15 and passing [7]10 = [111]2 to the **var** instruction field. Additionally, individual SR bits can be modified using a “Status Register Set” instruction (see section on instructions for additional detail).

### 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.

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

Note: Proper operation of the stack requires that the SP be initialized at startup.

As is the case with the Program Counter, the Stack Pointer can be divided into 2 8-bit subsections (SP\_high and SP\_low respectively) which can be addressed individually by passing register ID 15 through the appropriate field and setting **var** to [0]2 (low) and [1]2 (high) respectively.

### Interrupt ID

The Interrupt ID Register, or IID for short, is an 8-bit register containing a unique identifier pertaining to the interrupt or exception which was last triggered. The value of IID can then be used to infer what type of exception/interrupt was triggered and ultimately where to execute the appropriate handler. Its length allows for 256 unique exceptions and interrupts.

The IID register is **read only**. Its contents can be read by passing RID = [15]10 and var = [111]2.

# The TinyRISC machine model: Instruction syntax

TinyRISC uses fixed length, 16bit instructions to encode operations. The TinyRISC ISA itself defines 28 distinct instructions, with capacity for up to 32 separate instructions (5-bit fixed-length opcode field).

The available instructions can be loosely classed into two distinct categories: Register-Register types (“R-type”) and Immediate-Register types (“I -types”).

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] | | | | | Rs/Rd [4] | | | | Ro [4] | | | | var [3] | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **I-type** | opcode [5] | | | | | Rs/Rd [3] | | | Immediate [8] | | | | | | | |  |

## 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). As is plainly visible here, the **Rs** operator is here is destroyed, the result of the operation will later be found in its place once the operation is completed.

*An example: Register R1 and R2 contain [55]10 and [10]10 respectively. We wish to add them together.*

*We would use the following implied instruction (pseudocode):*

*The previous contents of R1 (55 in decimal) will have been replaced by the result of the above operation, which is (hopefully) [65]10.*

*Instruction fields:*

* **Opcode:** <*5 bits>* contains the unique identifier of the instruction
* **Rs/Rd:** <4 bits> contains the Register ID of the register acting as source operand and destination
* **Ro:** <4 bits> contains the Register ID of the register acting as second operand
* **Var:** <3 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.

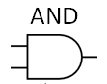
# Instruction listing:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Instruction name | Instruction class | opcode (dec) | opcode(hex) | Description | 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) |  |
|  |  |  |  |  |  |  |
| 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 |  |
|  |  |  |  |  |  |  |
| Status | Set Status register | I-type | 11 | B | Sets bits in status register | See section on status register operation |
|  |  |  |  |  |  |  |
| 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 |

## 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

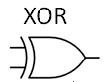
Description générée automatiquementBitwise OR operation. Compares bits of same rank in both operand and returns 1 if either (or both) operands are 1, 0 if otherwise.

***R-type instruction.***

***Opcode .***

### Logical XOR

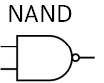
Une image contenant table

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***

## 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.

#### 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

## 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 as a result 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 | B1000 |
| Shift left | B0001 | OR | B1001 |
| Rotate Right | B0010 | XOR | B1010 |
| Shift Right | B0011 | NAND | B1011 |
| Add with C | B1100 |  |  |
| Subtract with C | B1101 |  |  |

The ALU is capable of executing 10 distinct instructions, hence the use of a 4-bit ALU code to communicate every instruction type.

Similar instructions only differ in a single bit; this is to make decoding easier. Let the ALU command codes be under format ABCD, where A is the MSB and D is the LSB.

Note: the transceivers controlling access from our subunits are ACTIVE-LOW. We wish them to be 0 ONLY if we want that unit to output its data.

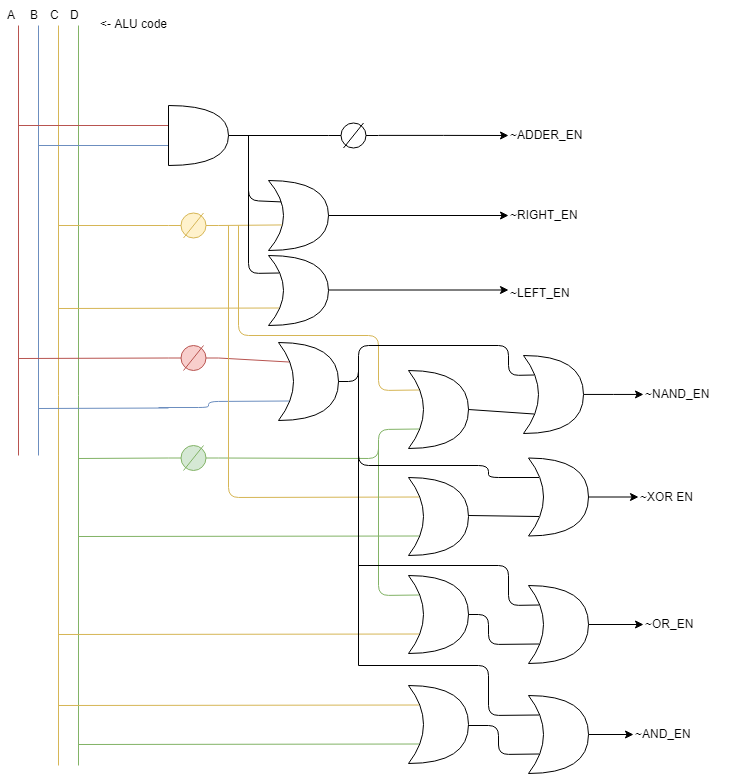
The activation equation for the adder/subtracter is simple, as there is only one actual unit inside. Bit D is instead used to negate (or not) the second operand. Looking at bits A and B, the adder is only active if both A and B are 1, hence:

The Shifter circuitry is comprised of 2 sub-units, one for each shift direction. Bit D is simply used to tell the shifter whether to reroute the shifted-out bit back into the bit array or not.

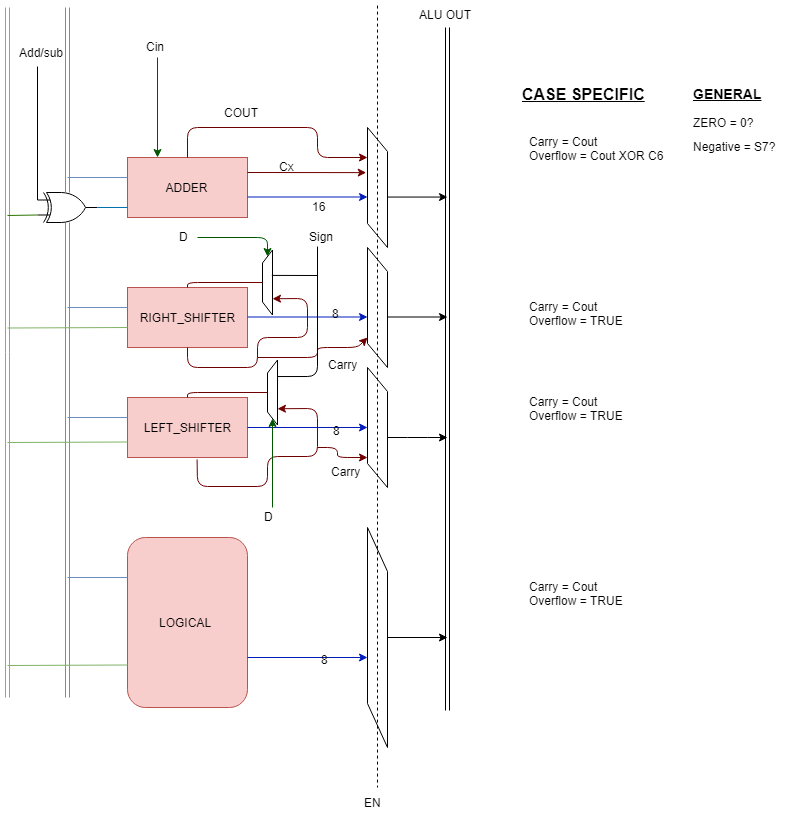
In either case the enable line will only be held low if both A and B bits are also low.

The logic circuitry is harder still, being comprised of 4 sub-units, each with a distinct function.

Implementation of the decoder: Inverters, OR gates, AND gates



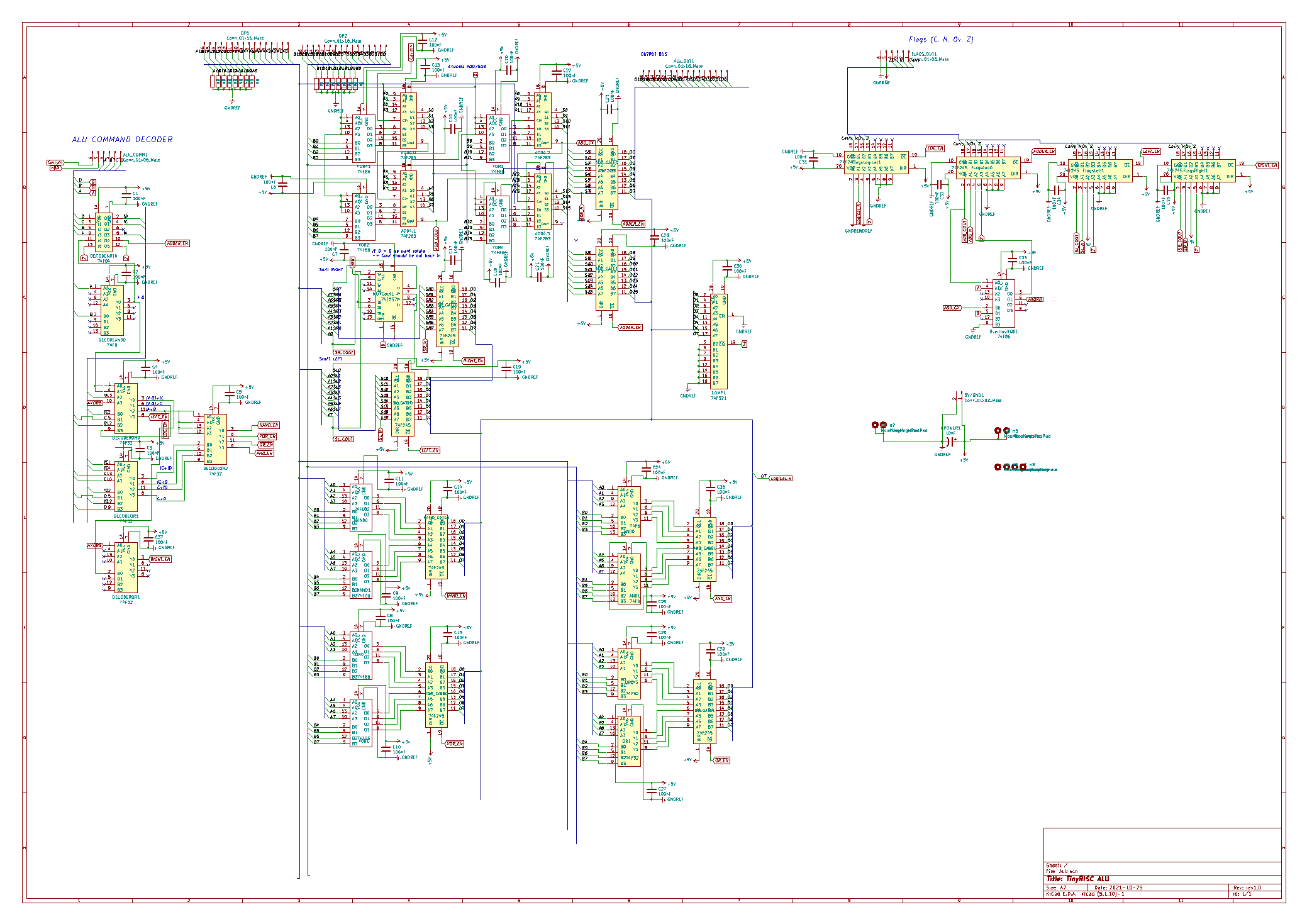
### High-level diagram



Chip candidates:

|  |  |
| --- | --- |
| Adder | 74F283 (x4) |
| Transceiver | 74F245 (x8) |
| wide MUX | 74F257 (x6) |
| 1b MUX | 74F157 |
| AND | 74F08 (x3) |
| OR | 74F32 (x5) |
| XOR | 74F86 (x3) |
| NAND | 74F00 (x2) |
| Invert | 74F04 |
| Comparator | 74F521 (x2) |

### Circuit Diagram



### 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:

* 15 8-bit registers, named 0 through 14.
* 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.