Practical Module 2: Register Transfer Language

Towards The Basic Data Path

This document is designed to show how we can start with a very simple adding machine, and then by incrementally adding components we arrive at a general purpose processor architecture. We use the circuits and insights from part 1 as the starting point. The goal is the *Basic Data Path*, as described in chapter 4.1 of the book.

In this section, we have multiple questions. For example, asking you to add busses to smaller architectures or think about instructions and clock cycles. Inspiration for the answers can be gained from studying the Basic Data Path; though note that the busses from the Basic Data Path are not the only possible busses to add to architectures. Instead, think about what the busses in those questions are required to do.

Integrating Components

In Module 1, you designed a 1-bit logic unit (multiplexor), and later a 1-bit adder. The inputs were connected to toggle switches and the outputs to LEDs. These two circuits can be combined (Fig 2.7 in the textbook) to form an ALU. The exact operation to be performed on the ALU inputs is determined by the selector inputs. The ALU can be extended to multiple bits (Fig 2.8 in the textbook).

We replace the banks of switches and LEDs with simple registers that only allow for pre-loading and outputting values. Register OR (Operation Register) contains the operation for the ALU. The result is stored in the Result register.

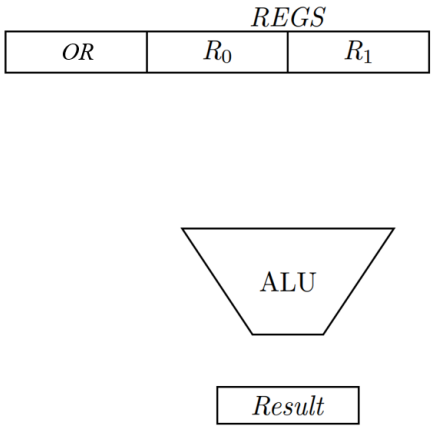

Figure 1 Four registers and a simple ALU

1. Add busses to connect the 3 registers to the ALU in order to allow for two register values to be processed by the ALU, and the ALU output to be stored in the Result register.
2. How many bits are required to encode all possible ALU operations if we have an ALU with an addition operation, a subtraction operation and the the operations of Question 1.4 in Part 1 of the logic module?
3. What additional circuitry is required to now allow the ALU to execute both of the following;
    i. Result = $R_0 - R_1$
    ii. Result = $R_1 - R_0$

Top

By using registers we can abstract away the actual source of the values and the ALU operation, and also use the same model regardless of the bit width of the registers. Instead we can focus on the fact that on the rising edge of a clock signal the values in the OR, R_0 and R_1 registers become fixed for at least the duration of one clock cycle and on the next clock cycle the output of the ALU is clocked into the Result register. We can express this using the following notation `destination <= source`.

Clock cycle 1 : `OR,R0,R1 <= ALU operation, value1, value2`
Clock cycle 2: `Result <= R0(bus) (ALU operation) R1(bus)`

4. Assuming the functionality of question 3 above has been implemented, why is it necessary to specify which register uses which bus when passing a value to the ALU?

Simple Instructions

The combination of ALU operation and data will now be referred to as an *instruction* and R_0, R_1 as the *source* registers. We assume the ALU is implemented with a bit width of 16 bits.

5. What is the total number of bits required for an instruction consisting of an operation and two values? i.e. to fully capture all possible operation codes and values?

Making Something Useful

Our instructions consist of an operation and two values, for example `ADD 55,46` (Result = 55 + 46); all of which must be manually entered into the registers. Whilst having a machine that can add two values is useful, it would be more useful to build programs, for example to sum a sequence of numerical values. However, in order to perform this simple task we need to build a circuit that is able to;

a) Stream a sequence of instructions and values to the circuit.
b) Somehow store and make use of the ALU output (e.g. to store a running total).

To stream a sequence of instructions to the circuit we introduce the notion of external memory. For this, we use RAM (Random Access Memory) to store a sequence of instructions (see also section 3.6 in the book). When our circuit is powered on we need to fetch the first instruction in RAM into the registers and process the instruction. We then increment (point) to next instruction, fetch it, and process it.

To do this we need an additional register for storing the address of the instruction to be fetched from RAM. We call this register the IP (Instruction Pointer) register. We extend this register with counting functionality. For now the IP register is only capable of counting from 0 to 2^n where n is the bit width of IP. A counting register can be designed by extending normal registers. See question 3.3 of the logic module for an example extension of a register of 4 bits.

To store and make use of ALU output, we generalize from input and output registers to a general purpose register bank, which can be used as both input and output. In the architecture, the register bank is shown as:

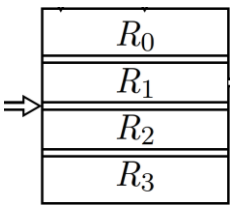

Figure 2 A register bank

6. What additional hardware is required such that a value from the ALU can be directed to any of the 4 registers?

Top

With the register bank now added to our simple architecture, instructions need the destination register specified, e.g. `ADD R0,55,46` (R_0= 55 + 46). However, note that this instruction uses more than 30 bits and is of limited use compared to an instruction such as `ADD R0,55` (R_0= R_0 + 55), as this kind of instruction can be used to for example calculate a running total and takes fewer bits. Therefore, we limit ourselves to using instructions of the second kind, like `ADD R0,55` (R_0= R_0 + 55) and `OR R0,R1` (R_0= R_0 v R_1).

These instructions can more generally be expressed as

```
RA <= RA(bus) (operation) RB(bus)     where RA is the destination and RB is the source
RA <= RA(bus) operation value         where RA is the destination
```

However, this can still lead to a disparity in the bit size of instructions and of values, which would need to be reflected in the bit width of the ALU and the circuitry of the RAM, which adds complexity. Therefore, we limit the instructions to the bit width of the ALU, 16 bits.

7. Consider storing instructions such as `ADD R0,55` and `SUB R1,128` in 16-bit RAM. Once all ALU operations (see question 2) and possible destination registers have been encoded, how many bits remain to encode the value? For the amount of bits necessary for destination registers, consider how many of registers we have in our current architecture. Also take into account that we need to distinguish between instructions like `ADD R0,55` and `OR R0,R1` in the instruction, where the source operand can be a value or a register.

Note that our ALU up till this point has negation functionality. Normally this would result in an instruction `NOT R0`, which only has a single argument. We do not add this instruction, such that in our architecture, an instruction always has two arguments. To bring our ALU in line with the one used in the basic data path, we additionally make sure that all ALU operations from 2.8 are possible in our ALU.

8. How many bits are available to encode the value now?

Examine the architectural components below in Figure 3. A new register in this figure is the IR, or Instruction Register. This register can hold the instruction after it is fetched from RAM. IR also has two small buses connecting to the register bank, named *IR.ra* and *IR.rb*.

9. What is the purpose of the connections *IR.ra*, *IR.rb*? What additional instructions are now possible?
10. Add to Figure 3 all the busses required to allow the processor to bring a stream of instructions one by one into the IR and allow the ALU result to be stored in any arbitrary register (R_0-R_3). Your busses should allow for instructions of the form `OP RA val` or `OP RA RB` to be executed, where `OP` is the operation to be executed. Note that the value or register references are encoded within the instruction held within IR. How can IP be incremented to point to the next instruction?

Top

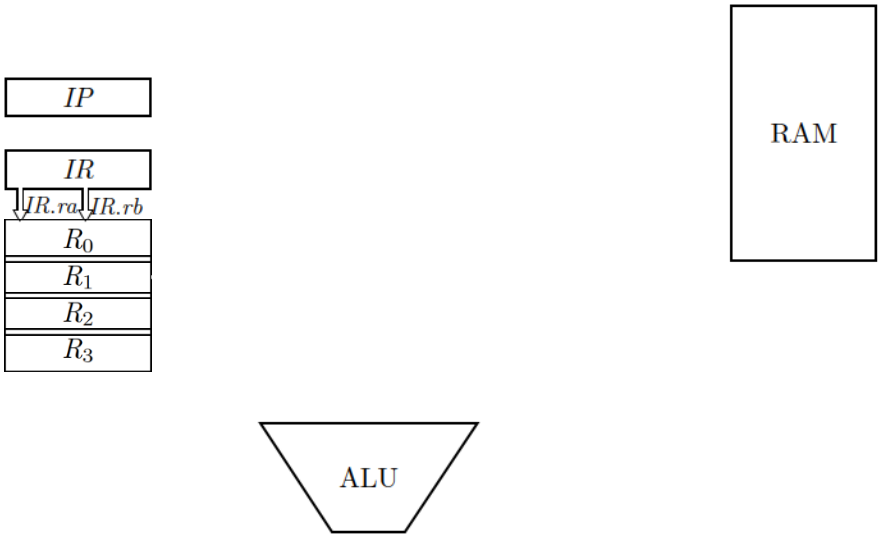

Figure 3 A simple architecture to allow sequences of instructions to be executed

Loading an instruction using the above architecture requires the following steps;

`RAM <= IP` (send the pointer value to RAM in order to access the memory location)
`IR <= RAM` (pass the instruction pointed to by IP, from RAM to IR)

Execution then follows, e.g.
`RA <= RA(bus) (operation) RB(bus)` (destination = destination <operation> source)

Expressing executions with RA and RB means we do not have to worry about which registers are involved, the process is the same regardless of which two registers are involved.

The above architecture allows for a sequential program such as the following

```
AND R0, 0              Sets R0 = zero
ADD R0, value1         Add a value to R0 (could also use OR)
ADD R0, value2         -"-
AND R2, 0              Sets R2 = zero
OR  R2, R1             Store R1 in R2 (subtotal)(could also use ADD)
```

11. The architecture developed so far is still limited to processing a sequential program in order. Is it possible to use such an architecture to perform multiplication or division?

Loading Values

Although the ALU can accommodate 16 bit values, the values we can actually use are limited in range due to the 16-bit instructions containing both opcodes and operands (4.1 in the textbook and question 8 above). To utilize the ALUs full bit width requires 16-bit values be loaded from memory. There are two general use cases for this. First where we wish to use values of up to 16 bits as part of an instruction, e.g. `ADD R0,64000`. Secondly, where an array of these values is to be used by a program e.g. summing a total. The first use case is addressed in question 4.2 of this session. If we allow RAM values to be brought into a register directly, this leads to two further instructions, noting that [ ] is used to denote that `[number]` is an address;

`LOAD R0,[125]` (R_0 is filled with the value stored in RAM at [address])
`LOAD R0,[R1]` (R_0 is filled with the value stored in RAM at the [address] held in R_1)

Top

If we wish to allow the RAM values to be passed straight to the ALU, this allows for instructions similar to the following;

`ADD R0, [125]` (R0 = R0 + value held in RAM at [address])

12. Is there a difference between instructions `LOAD R0,0x12A` and `LOAD R0,[0x12A]?` (0x denotes a hexadecimal number)
13. In what way is the instruction `LOAD R0,[address]` limited? Think about question 7.

In the architecture, we make use of an address register ∑A and a read register ∑R, instead of directly connecting other registers to the RAM. The inclusion of these registers stems from the fact that while our architecture model includes both processor components and the RAM side by side, the processor and the RAM are separate parts of a computer, which is reflected in the architecture by a set of buffer registers (starting with a ∑). These registers signify the processor sending (or receiving) something from the RAM, like sending an address to ∑A or receiving a value from ∑R.

Examine the architecture below in Figure 4;

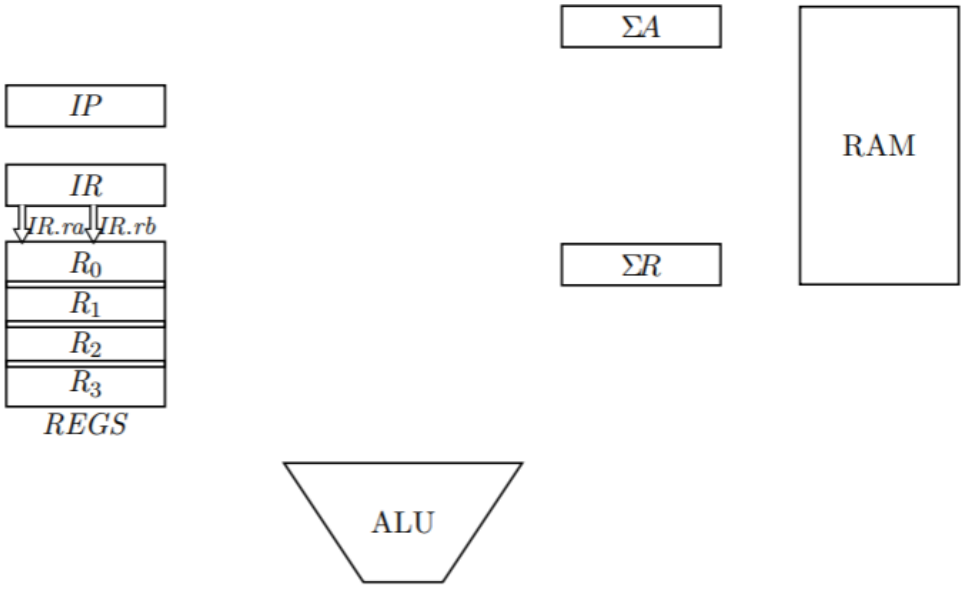

Figure 4 An architecture that allows for loading of values from RAM to registers

14. Add in the busses to Figure 4. Make sure the ALU can write to the register bank and the address register can be written to by IR, IP and the contents of a Register. Also make sure instructions can be loaded in from RAM to IR. Use as few buses as possible.
15. ∑A holds the address of the RAM location to either be written to or read from. If this register has a bit width of 16, what is the maximum number of addressable locations?

An instruction such as `LOAD R0,[value]` can be brought in from RAM and executed with the current components. If we leave out the buses, we can express this execution as a sequence of register transfers:

Clock cycle 1: `∑A, IP <= IP,IP+1` (Fetch - place address of instruction in ∑A)
Clock cycle 2: `∑R <= RAM[∑A]` (Fetch - copy the instruction from RAM into ∑R)
Clock cycle 3: `IR <= ∑R` (Execute - copy the instruction from ∑R into IR)

Top

Clock cycle 4: `∑A <= IR.val` (Execute - copy the instruction value into ∑A)
Clock cycle 5: `∑R <= RAM[∑A]` (Execute - copy the value from RAM into ∑R)
Clock cycle 6: `RA <= ∑R` (Execute – copy the value from ∑R to RA)

The register transfers are based on implementing the instruction on the Basic Data Path, but that does not mean that we can have the same transfers on the data path you made in question 15.

16. Adapt the above sequence for your data path and add in which transfers use which buses if multiple buses can be used for the transfer.

Storing Values

Storing values in RAM requires us to use the address register ∑A, and also include an additional register ∑W to store the value to be written. This enables further instructions such as;

`STOR R0,[0x12A]` (the value in R_0 is store at the address given as a numerical value)
`STOR R0,[R1]` (the value in R_0 is stored at the address given in register R1)

17. Using the architecture below, implement the busses required to pass the [address] value, or the register contents into ∑W.

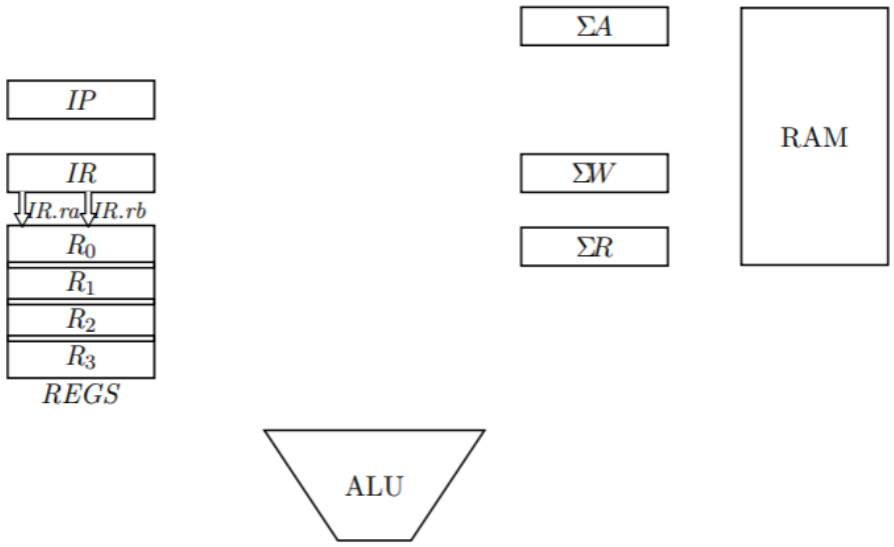

Figure 5 An architecture that allows for loading and storing of values

18. The final basic datapath in the textbook *does not* allow for instructions to specify an address as the destination, for example `ADD [0x124],R0`. What would be the effect on the number of bits required to encode operands and opcodes in instructions if this were allowed?

Our architecture is now capable of sequentially executing a sequence of instructions. We can also use RAM to store and load values. This would allow us to write a program that could sum a list of data values (providing we can halt execution at the last instruction), but it does not allow for a sequence of instructions to be repeatedly executed or for a sequence of instructions to be skipped over conditionally. To implement this functionality requires that we can unconditionally, and conditionally, break the sequential execution of instructions by pointing IP at any instruction in memory.

Top

For the following sections it is sufficient to refer directly to the basic datapath as there are only a few small additions to make.

Unconditional Branching

In order to execute an instruction out of sequence with the main program we must be able to place the address of that instruction into the IP. For example, consider a program that infinitely loops and so must execute the first instruction after the last. This requires that the last instruction in the program is one that changes IP to the address of the first instruction in the program (note that IP at present is only a counter register that outputs a value).

This also requires that we know the exact address of the first instruction in the program. This is complicated by the fact that programs held in external storage are loaded into RAM by an operating system, and the exact memory address of instructions is unknown (textbook, p79). An alternative solution is to use relative displacements, i.e. jump 102 instructions forward/backwards or jump to a particular label in the code. This is covered in textbook on, p79.

If we wish to implement a branching instruction `BRA address`, then the destination address is encoded in the instruction and must be passed into the IP register. If we wish to implement an instruction of the type `BRA displacement` then the displacement value must be added to the current IP value before placing it in the IP register i.e. the IP must also be connected to the ALU. Note that displacements can be either positive, or negative.

19. Which connections in the Basic datapath are used for positive and negative displacements in branching instructions?

Conditional Branching

A conditional branch allows computers to break the sequence of execution and jump to another memory location if and only if a predetermined condition evaluates to true. Evaluating the condition is done by the ALU and results in a number of flags being set that are indicative of the comparison. The flags are stored in a condition code, or status, register.

20. Identify the flags used in the basic datapath.

When describing data transfers using the basic datapath, we generally assume that the values of the flags are known to the processor. We therefore do not need to read those values from CC. However, we do need to take care to describe that the flags are set during an instruction, which we do in Register Transfer Language through the register transfer `CC <= ALU.cc`.

Subroutines

Unconditional and conditional branching allow all higher level language programming structures to be implemented; Do-While, While-Do, If-Then, If-Then-Else. However, in order to implement subroutines (re-useable blocks of code) the architecture must be capable of storing a return address.

21. Read the first 3 paragraphs of section 5.6 and identify the bus used by the Basic Data Path to allow for return addresses taken from IP to be stored in the RAM.

Conclusion

Now that you have an understanding of the basic datapath you can work on the RTL exercises. These exercises are designed to assess your understanding as to how a processor executes different types of instructions based on the architectural constraints. This understanding is expressed via RTL sequences.

Top