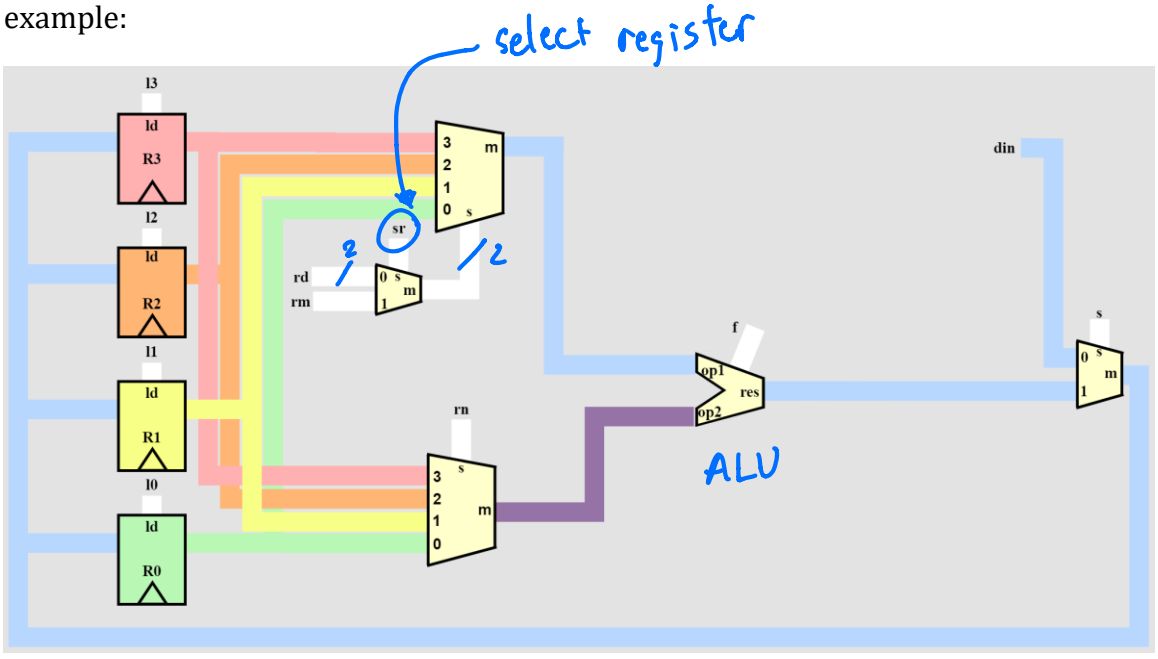
much simpler than the original processor

# The ARM0 processor

ARM0 designs by Craig Scratchley.
Some paragraph text derived from work by Tony Dixon and used with permission.


## A Simple Processor Datapath

A "Processor Datapath" is a multi-function sequential digital system that provides functions that can perform arithmetic and logic computations on the contents of registers and save the results. The following logic diagram provides a simple example:




The coloured busses are all 8-bits wide. The taller, rectangular white control busses (f, rm, and rn) are each 2 bits wide, and the white squares are single-bit signals.

Analyzing the behavior of the processor datapath will identify all the possible functions this system can perform with an assignment of values to the control inputs. With 11 bits of control inputs, there are 2^11 possible functions. However, the full functionality of a processor datapath is not always required. Instead, a subset of useful functions is identified and the processor datapath is used only to provide these functions.

Any function that a processor datapath can perform with a single assignment of values to its control inputs is called a μ-instruction. Thus μ-instructions correspond

to the rows of the function select table for the processor datapath. To determine if a processor datapath provides a μ-instruction for a given task, it is necessary to find an appropriate assignment of values to the control inputs. Such an assignment is called a "control word".

In logic diagrams, the control signals are grouped into a bus called a "control bus". The bits of a control bus are connected to the individual control input ports of the components of the processor datapath.

This system is able to input values and place them, one at a time, in any registers. It can also perform any of the 4 functions provided by the ALU with 1 or 2 registers, as appropriate, as the 1 or 2 needed operands.

Now consider the following register transfer statement:

Rd ← Rn - Rm

This is not among the functions that are provided by the ALU. However:

Rn – Rm = Rn + (NOT Rm + 1)

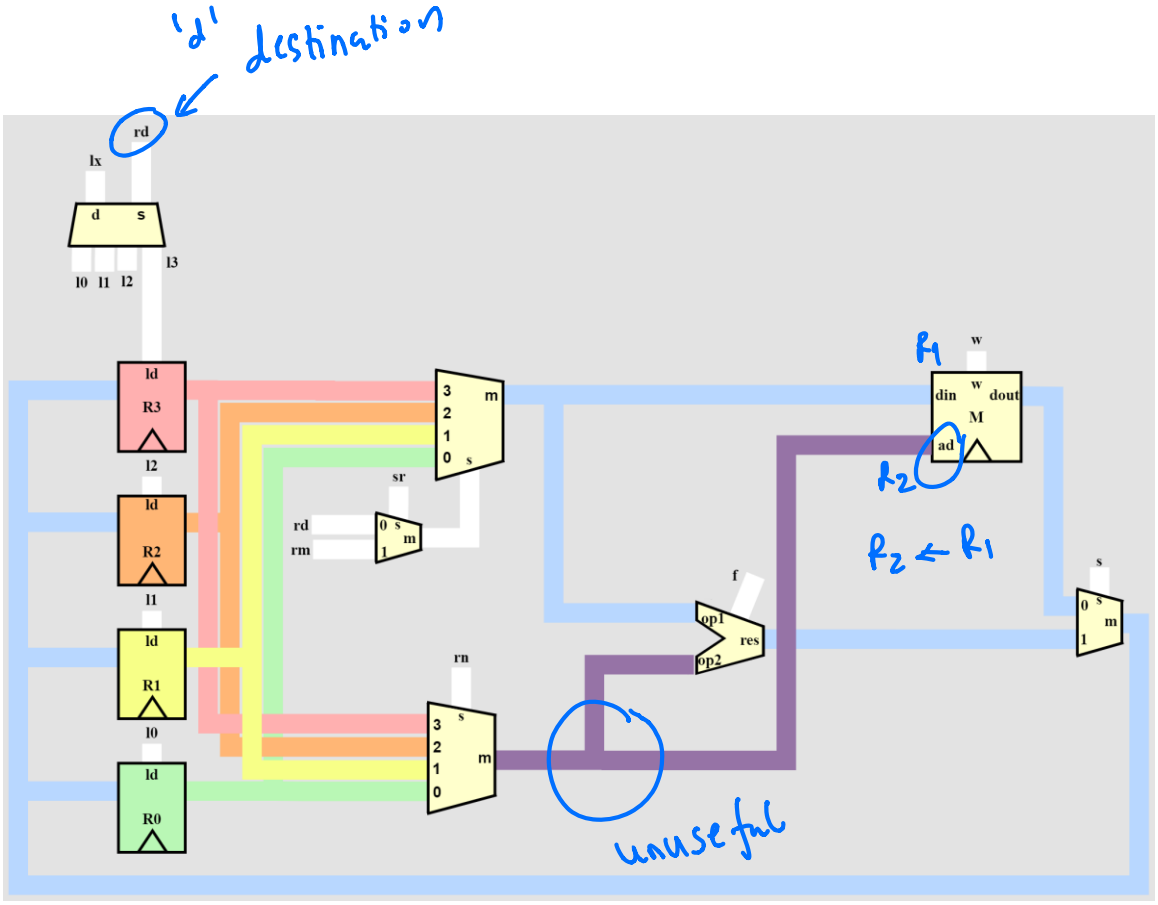
provided the numbers are encoded using 2's complement representation. Therefore, the following sequence of processor operations will permit the processor to perform the subtraction

R0 = R0 – R1:

1. R2 ← 1
2. R1 ← NOT R1
3. R1 ← R1 + R2
4. R0 ← R0 + R1

In other words, subtraction is defined by an algorithm that uses the operations provided by the processor.

Now let's extend the processor by providing a 256×8 memory from which values can be retrieved from any one of 256 locations and where the contents of the registers can be stored.




The introduction of the memory and also two additional multiplexors means that the number of control points that define the function select table for this digital system is increased over the previous one. Specifically, we must now account for different assignments of values to the 12 bits worth of control inputs f, rm, rn, rd, (2 bits each) and lx, s, sr, w, which could mean a table of 2^12 rows. Note that rd appears twice in the diagram. Once to control which of l0 through l3 gets the lx signal in the demultiplexor (lx becomes 1 when loading a register from memory), and once as an input bus to the *sr* ("select register") multiplexor to possibly become the selector (i.e. 's') bits for the higher multiplexor with 4 sets of inputs (sr selects rd when storing a register to memory).

Instead of enumerating all rows, only a subset of the rows would be worthwhile specifying. The rows chosen would represent a smaller "toolbox" of operations that are deemed useful in the construction of algorithms for more complex tasks. An example of something "unuseful" would be writing to memory when the address is also the second operand of a binary operation being performed in the ALU.

It is evident that tasks can be defined by one or possibly more µ-instructions. For any processor-datapath architecture, a subset of possible tasks can be chosen, called the "instruction set" of the processor. A set of bits which identify an instruction

together with any accompanying value or values that may be required by the type of instruction to execute form what is called a "machine instruction".

For RISC processors such as the 8-bit ARM0 processor that we are designing, all instructions are typically the same length (8 bits in our case), and the instructions have bits that identify the type of instruction and usually bits for operands of the instruction. In addition, RISC processors usually have only one μ-instruction per machine instruction.

In the table below, each row of the table is for a different type of processor instruction and has been assigned a unique label providing a "mnemonic" representation. The processor instruction performs the task described under the heading "task".

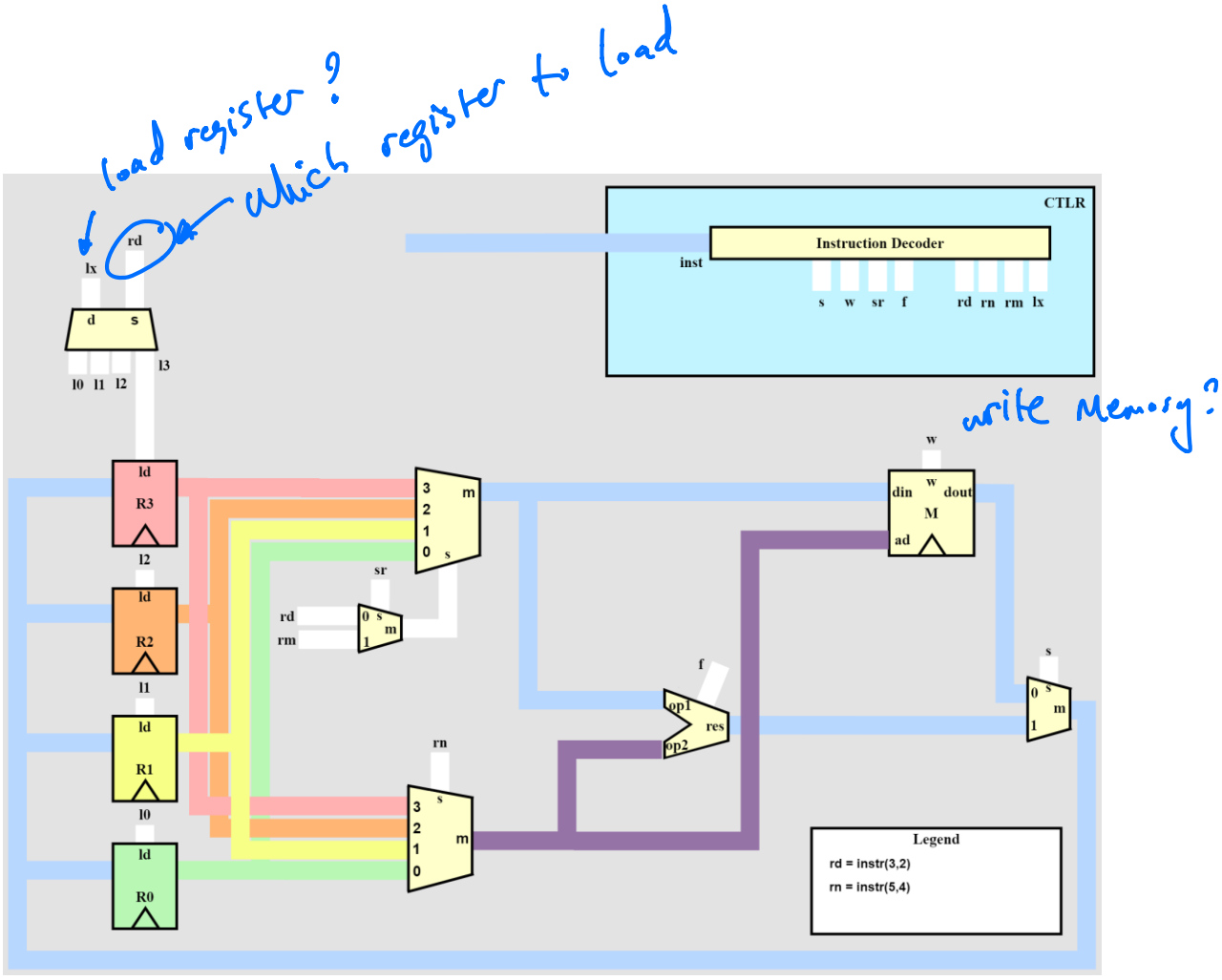
| instruction | | | | | | | | control signals | | | | | task | mnemonic |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i7 | i6 | i5 | i4 | i3 | i2 | i1 | i0 | s | f | lx | sr | w | | |
| 1 | 0 | \|-- Rn --\| | | \|-- Rd --\| | | 0 | 0 | X | XX | 0 | 0 | 1 | M[Rn]←Rd | STRB |
| 1 | 1 | \|-- Rn --\| | | \|-- Rd --\| | | 0 | 0 | 0 | XX | 1 | 0 | 0 | Rd←M[Rn] | LDRB |
| 0 | 0 | \|-- Rn --\| | | \|-- Rd --\| | | \|- Rm -\| | | 1 | 01 | 1 | 1 | 0 | Rd←Rn + Rm | ADD |
| 0 | 1 | 0 | 0 | \|-- Rd --\| | | \|- Rm -\| | | 1 | 11 | 1 | 1 | 0 | Rd←Rm | MOV |
| 0 | 1 | 1 | 0 | \|-- Rd --\| | | \|- Rm -\| | | 1 | 10 | 1 | 1 | 0 | Rd←NOT Rm | MVN |

store a byte in memory

load a byte mem

So for our design, bit i7 of the machine instruction distinguishes between memory instructions and data processing instructions. Bit i6 distinguishes between the "store register to a byte" and "load register with a byte" memory instructions and the "add" and the two "move" data processing instructions. Bit i5 distinguishes between the normal version (MOV) and ones complement ("negated") version (MVN) of the "move" instructions.

An instruction decoder can be added to our design to decode the instructions and generate the required control signals. Note that some of the control signals (Rn, Rm, and Rd) fall right out of the instructions.




## MACHINE LANGUAGE PROGRAMS

Definition: A machine language program is a sequence of machine instructions supplied to a processor that, when performed in the order given, performs a more complex task. For example, a machine program to perform the calculation

M[1] = 1 – 2

is shown below. This will calculate 1 – 2 and put the result in memory location 1. It requires memory location 0 to initially hold the value 1. The right-hand side of the table defines the machine language program for the algorithm to perform a subtraction.

| step | RTL statement | instruction |
|---|---|---|
| 0 | R2 ← M[R3] | 11111000 |
| 1 | R1 ← R2 + R2 | 00101010 |
| 2 | R0 ← R2 | 01000010 |
| 3 | R1 ← NOT R1 | 01100101 |
| 4 | R1 ← R1 + R2 | 00010110 |
| 5 | R0 ← R0 + R1 | 00000001 |
| 6 | M[R2] ← R0 | 10100000 |

It is assumed that R3 will hold the value 0 at the beginning of the program (more on that later). If we can initialize the memory so that memory location 0 is holding the data value 1, then the program should work properly. The data value of 1 at location 0 is loaded into R2. 1 is needed to perform any subtraction. Then the value 1 is added to itself and stored in R0 to form the minuend. The value 1 is copied to R1 to form the subtrahend. Then the subtraction begins. The first step is to calculate the 2's complement of the minuend stored in R0. The 2's complement is put in R0. With the 2's complement calculated, the difference can be calculated by doing a final addition. At the end of the program, the difference is stored at memory location 1.

A "symbolic representation" of the machine language program can now be expressed as follows:

Symbolic Program

```
LDRB R2, [R3]    ;@ 1: needed for 2's comp.
ADD R1, R2, R2   ;@ init R0 to 2
MOV R0, R2       ;@ copy 1 to R1

;@ calculate 1 – 2

                 ;@ calc. 2's comp of R0
MVN R1, R1       ;@ a) take 1's comp of R0
ADD R1, R1, R2   ;@ b) add 1

ADD R0, R0, R1   ;@ complete subtraction
STRB R0, [R2]    ;@ store result
```

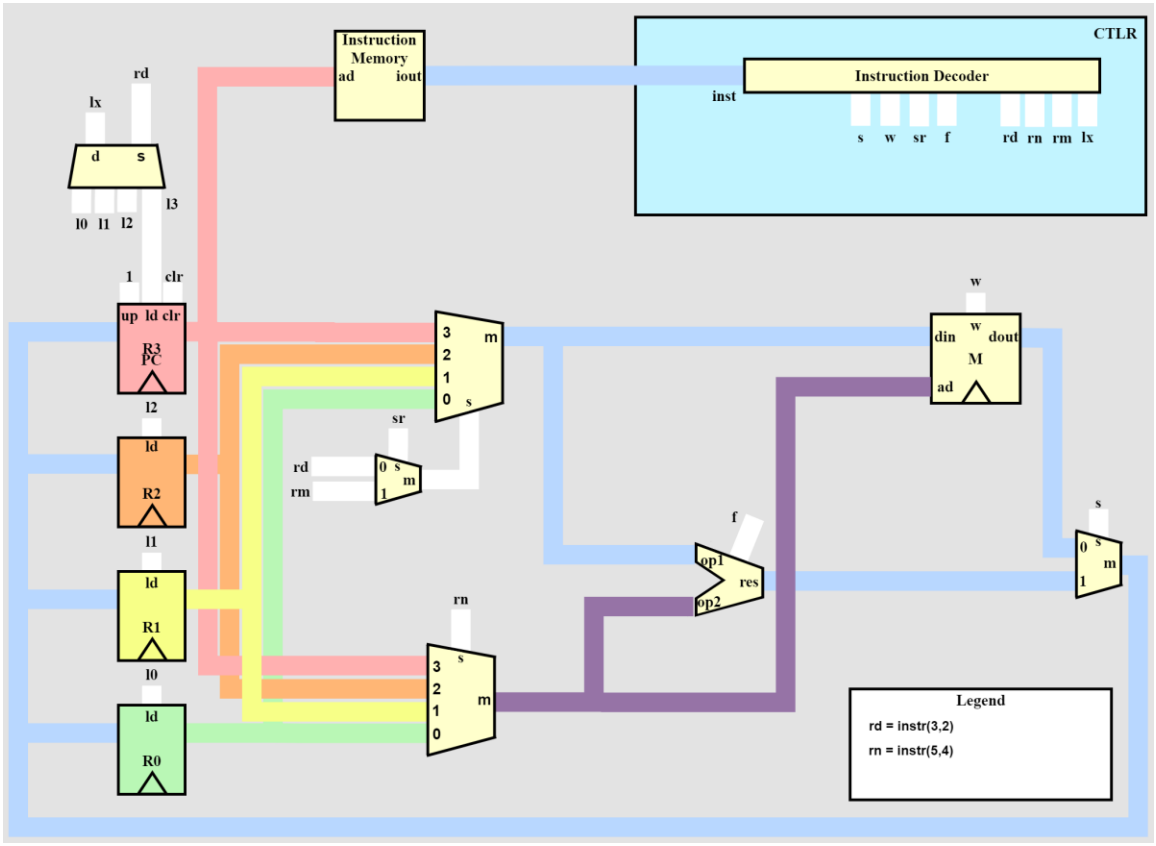
While the symbolic representation is more readable to humans, it must be translated into machine language instructions before it can be executed.

## STORED PROGRAMS

Rather that physically entering each machine language instruction of a program individually in the order needed, they can be stored in a memory because they are binary sequences. The execution of the machine language program then consists of retrieving the instructions, one at a time, from the memory and applying them as we have been doing manually in the previous section.

When a program and the data that it will access are stored in memory units accessed by different buses, the processor architecture is called “Harvard” architecture.

To implement the Harvard architecture for the processor under development, the logic must be extended with an instruction memory.




A fancy register called a Program Counter (PC) must be used to provide addresses to the instruction memory. The PC is a register that indicates where in memory the

next instruction to be executed can be found. In our design, we will use R3 as the PC, and so those names are aliases for each other in our design. While a typical PC has the ability to be loaded with a specific address in order to branch to a different part of the program, a PC must have the ability to be incremented. Unless an instruction explicitly alters the PC, on the appropriate clock edge (we are assuming the rising edge) the PC must be incremented to the next instruction. Our PC has a control input called "up" that can be used to increment it. If both *up* and the PC's *ld* (i.e. *l3*) signals are active, we assume that *l3* takes precedence over input *up* and only a load results. We have simply wired input "up" to 1 in this design because we want the PC to be incremented on every rising clock edge unless overridden by the *l3* signal.

Earlier we wrote that R3, now also known as PC, would hold the value 0 at the beginning of the program. We can go further by stating that, when the processor is powered on, the PC must come up holding the value 0. Thus, the first instruction that the processor will execute will be at address 0 at the beginning of the instruction memory.

Notice that the PC also has a clr control input. This will clear the PC back to 0. So at any point as instructions are being executed, the PC can be cleared and the processor will start at address 0 all over again.

### Immediate Data and Offsets

For the program discussed above, I wrote that location 0 of the data memory needed to be initialized to 1. How can we do that? In practice such a thing is often difficult because the data memory is often volatile RAM, and doesn't come up with the value 1 in location 0 when it is powered on. In contrast, instruction memory is often ROM, non-volatile, and the entire program starting at instruction address 0 will be available when the processor is started. It is practical to put some constant data in instruction memory where it is easily available when the processor is powered on. One way to do this is using so-called immediate data, which is data buried within an instruction.

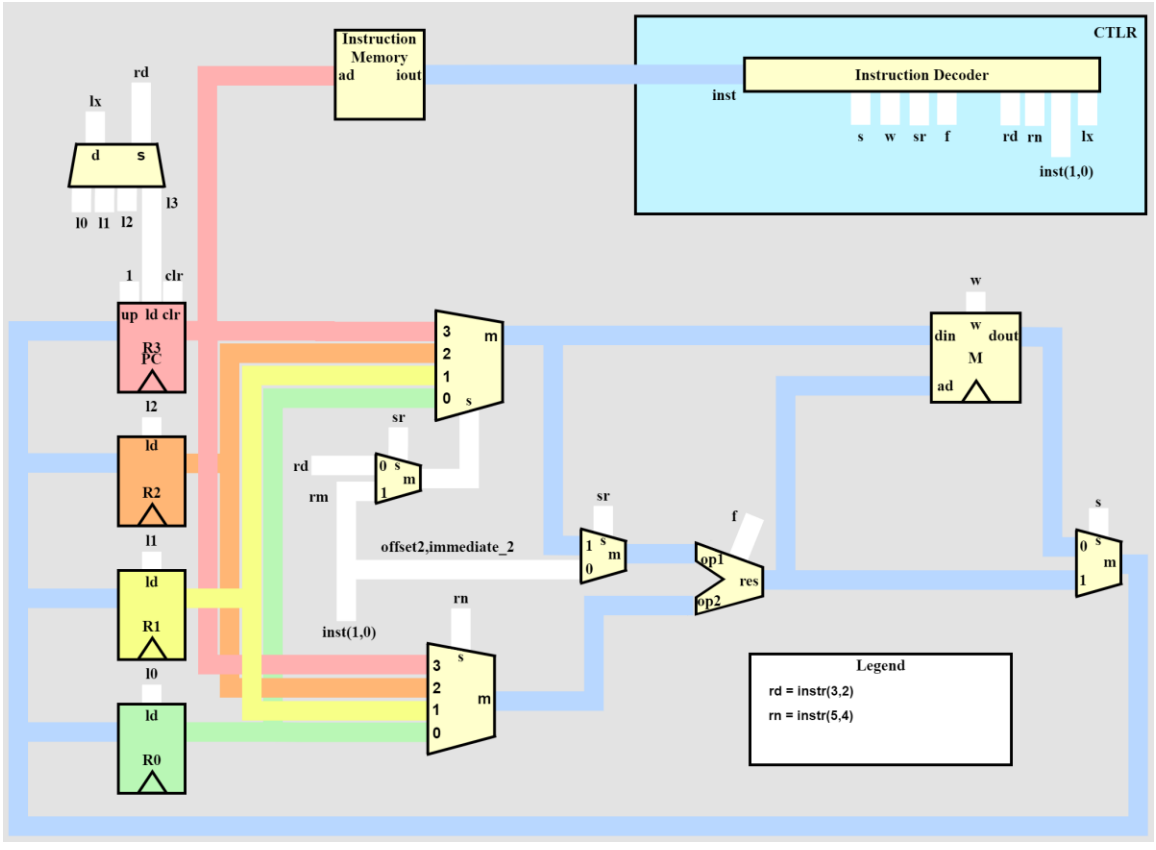
Notice in the instruction table shown earlier that some bit patterns were not being used. For example, instruction bit i4 was always 0 for the MOV and MVN instructions. We could use bit i4 to choose between a register and immediate data as the source of the data being moved. With i4 == 0, Rm in bits 0 and 1 would select a register as the source of the data. With i4 == 1, bits 0 and 1 can be interpreted as unsigned immediate data to be further processed (in the case of the MVN

instruction) and moved.

Notice how we saved the difference at location 1 in the above program. What if we wanted to save the difference at a different location, like 2. How could we easily do that?

The real ARM processor allows one to specify an offset from *Rn* when loading or storing a register. We can do that with our ARM0 processor as well by putting an offset in bits 1 and 0 of the instruction.

The version of the processor shown below includes the ability to have immediate data for MOV/MVN instructions and offsets for LDRB/STRB instructions.




Note that control signal sr now controls two multiplexors. It chooses both between Rd and Rm for the selector of the output of the higher multiplexor with 4 sets of inputs, and also between the output of that multiplexor and bits 1 and 0 of the instruction (those two bits zero-extended out to 8 bits).

Here is the new table of processor instructions:

| instruction | | | | | | | | control signals | | | | | task | mnemonic |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i7 | i6 | i5 | i4 | i3 | i2 | i1 | i0 | s | f | lx | sr | w | | |
| 1 | 0 | \|-- Rn --\| | | \|-- Rd --\| | | offset2 | X | 01 | 0 | 0 | 1 | M[Rn]←Rd | STRB |
| 1 | 1 | \|-- Rn --\| | | \|-- Rd --\| | | offset2 | 0 | 01 | 1 | 0 | 0 | Rd←M[Rn] | LDRB |
| 0 | 0 | \|-- Rn --\| | | \|-- Rd --\| | | \|- Rm -\| | 1 | 01 | 1 | 1 | 0 | Rd←Rn + Rm | ADD |
| 0 | 1 | 0 | 0 | \|-- Rd --\| | | \|- Rm -\| | 1 | 11 | 1 | 1 | 0 | Rd←Rm | MOV Rd,Rm |
| 0 | 1 | 0 | 1 | \|-- Rd --\| | | immed2 | | 1 | 11 | 1 | 0 | 0 | Rd←iout(1,0) | MOV Rd, #i |
| 0 | 1 | 1 | 0 | \|-- Rd --\| | | \|- Rm -\| | | 1 | 10 | 1 | 1 | 0 | Rd←NOT Rm | MVN Rd,Rm |
| 0 | 1 | 1 | 1 | \|-- Rd --\| | | immed2 | | 1 | 10 | 1 | 0 | 0 | Rd←NOT iout(1,0) | MVN Rd, #i |

And here is an updated version of a subtraction program:

| step | RTL statement | instruction | Symbolic Program |
|---|---|---|---|
| 0 | R2 ← iout(1,0) | 01011001 | MOV R2, #1 ;@ needed for 2's comp. |
| 1 | R0 ← NOT iout(1,0) | 01110000 | MVN R0, #0 ;@ load 255 or -1 |
| | | | ;@ (1's comp of 0) |
| | | | ;@ calculate 255 – 3 or -1 – 3 |
| | | | ;@ take 2's comp of 3 |
| 2 | R1 ← NOT ir(1,0) | 01110111 | MVN R1, #3 ;@ a) take 1's comp of 3 |
| 3 | R1 ← R1 + R2 | 00010110 | ADD R1, R1, R2 ;@ b) add 1 |
| 4 | R0 ← R0 + R1 | 00000001 | ADD R0, R0, R1 ;@ complete subtraction |
| 5 | M[R3 + ir(1,0)] ← R1 | 10100100 | STRB R0, [R2, #0] |
| 6 | R3 ← iout(1,0) | 01011110 | MOV PC, #2 ;@ PC is ARM0 alias for R3 |