Designing a datapath

Mike Spivey,
Trinity Term, 2019

The plan now is to put together the elements we have studied into a simple datapath that can execute Thumb instructions. We'll do it in stages, adding at each stage just sufficient logic to implement a few more instructions. The design we make will be seriously unrealistic, in that all the work of executing an instruction will be performed inside a single clock cycle: this will lead to longer combinational paths than we would like, and so require a lower clock speed. A more practical design would use *pipelining* to overlap the execution of each instruction with preparation for the next one. You can study pipelining and the design questions it raises in next year's course on Computer Architecture. For now, we must be content with the observation that if we wanted a pipelined implementation of the Thumb architecture, then effort spent on this single-cycle implementation would not be wasted, because a pipelined design starts with a single-cycle design, drawing lines across the circuit to separate what happens for a particular instruction in this clock cycle from what happens in the next.

The design is shown in these notes by means of a sequence of circuit diagrams, each accompanied by a selection of settings for the control signals that correspond to instructions that the circuit is capable of executing. The final design is also represented by a register-level simulator written in C. The simulator contains tables that decode all the instructions it implements, and is capable of loading and executing binary images prepared using the standard Thumb tools.

1 Instruction fetch

The first stage (Figure 1) is to arrange to fetch a stream of instructions from memory, and decode them into control signals that will drive the rest of the datapath. For this, let's install a program counter *PC*, a register that will, on each clock cycle, feed the address of the current instruction to a memory unit *IMem* so that it fetches a 16-bit instruction. There's also a simplified adder that increments the PC value by 2 and feeds it back into the PC as the address of the next instruction. Some of the 16 bits of the instruction are fed into a combinational circuit that decodes it, producing a bundle of control signals that are fed to the functional units in the datapath. Since those functional units and their connections are yet to be added, we can't say precisely what

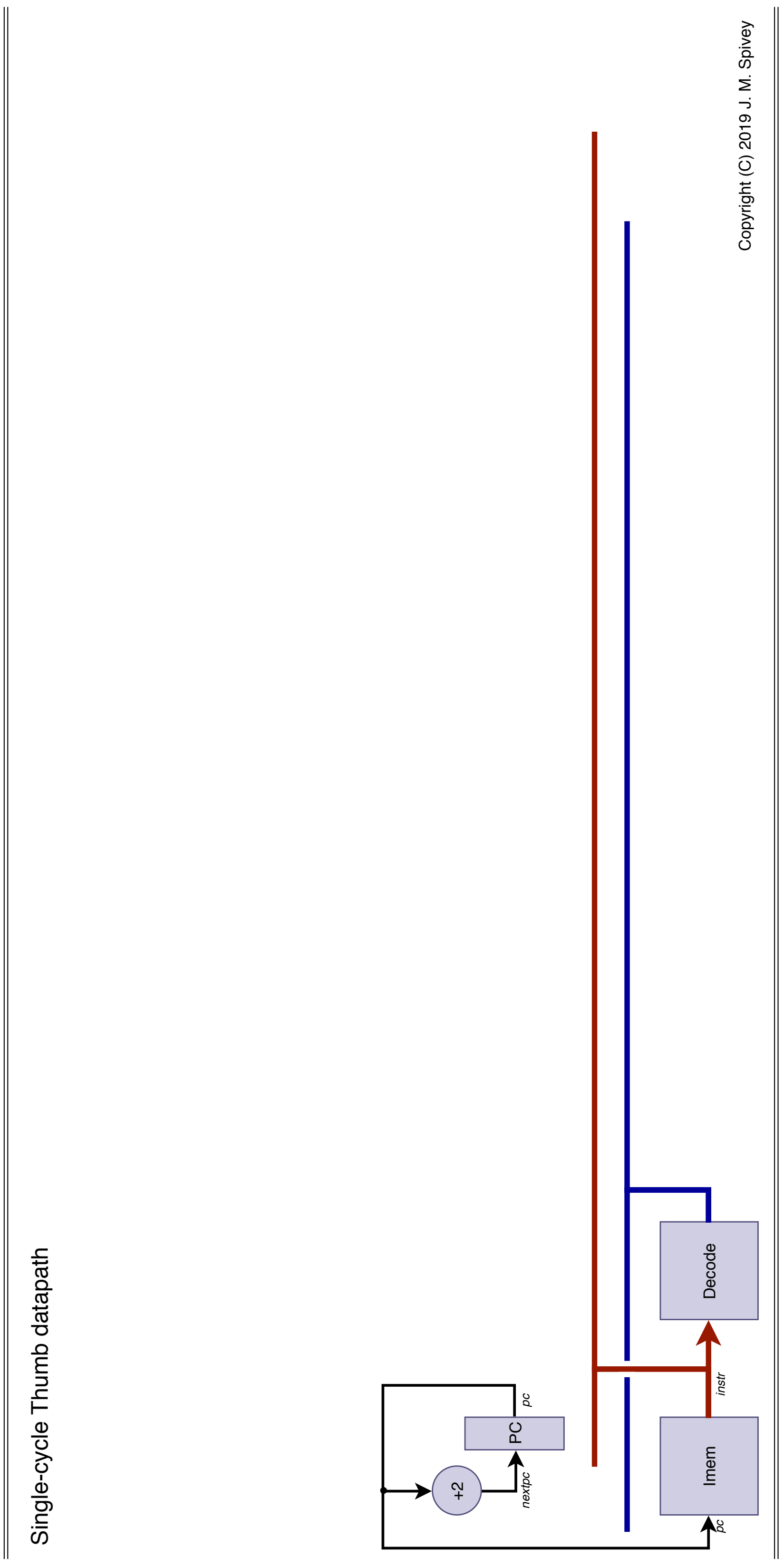

Figure 1: *Instruction fetch*

the control signals are at this stage; but let's add wiring that makes both the control signals and the remaining bits of the instruction (those that have not been accounted for in the encoding) available to each part of the datapath. We can imagine building the decoder from a ROM or (as we'll see later) several ROMs that decode different parts of the instruction.

This design is capable of implementing only straight-line programs with no branching, because there is no way to avoid a sequence of PC values that goes 0, 2, 4, 6, ... Also, this design doesn't reflect the fact that instructions can access the PC alongside the other 15 registers. We'll adjust the design later to correct both of these problems.

2 ALU operations

Now let's add some datapath components: a simple register file and an ALU (Figure 2). The twin-port register file is capable of reading the values of two registers and writing a result to a third register, all in the same clock cycle. We can imagine for now that the three registers are selected by fixed fields in the instruction, as they are in the add reg instruction:

adds ⟨Rx⟩,⟨Ry⟩,⟨Rz⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rz			Ry			Rx		

In executing this instruction, the two registers that are read are selected by fields *instr*⟨5:3⟩ and *instr*⟨8:6⟩ of the instruction. The control unit must ask the ALU to add its two inputs, producing a result that is fed back to the register file. The control unit also tells the register file to write the result back into the register selected by *instr*⟨2:0⟩.

The same datapath could be used to implement other instructions that perform arithmetic on registers – the three-register form of sub, certainly:

subs ⟨Rx⟩,⟨Ry⟩,⟨Rz⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	Rz			Ry			Rx		

For this instruction, we need to tell the ALU to subtract rather than add. But we can also implement instructions like ands that specify two registers and overwrite one of their operands:

ands ⟨Rx⟩,⟨Ry⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Ry			Rx		

This instruction is a bit different, because the ALU must do a bitwise AND operation, but also because the three registers are selected by different fields of the instruction, with *instr*⟨2:0⟩ used to select both one of the inputs and the output of the instruction. Let's leave aside for a while these issues of

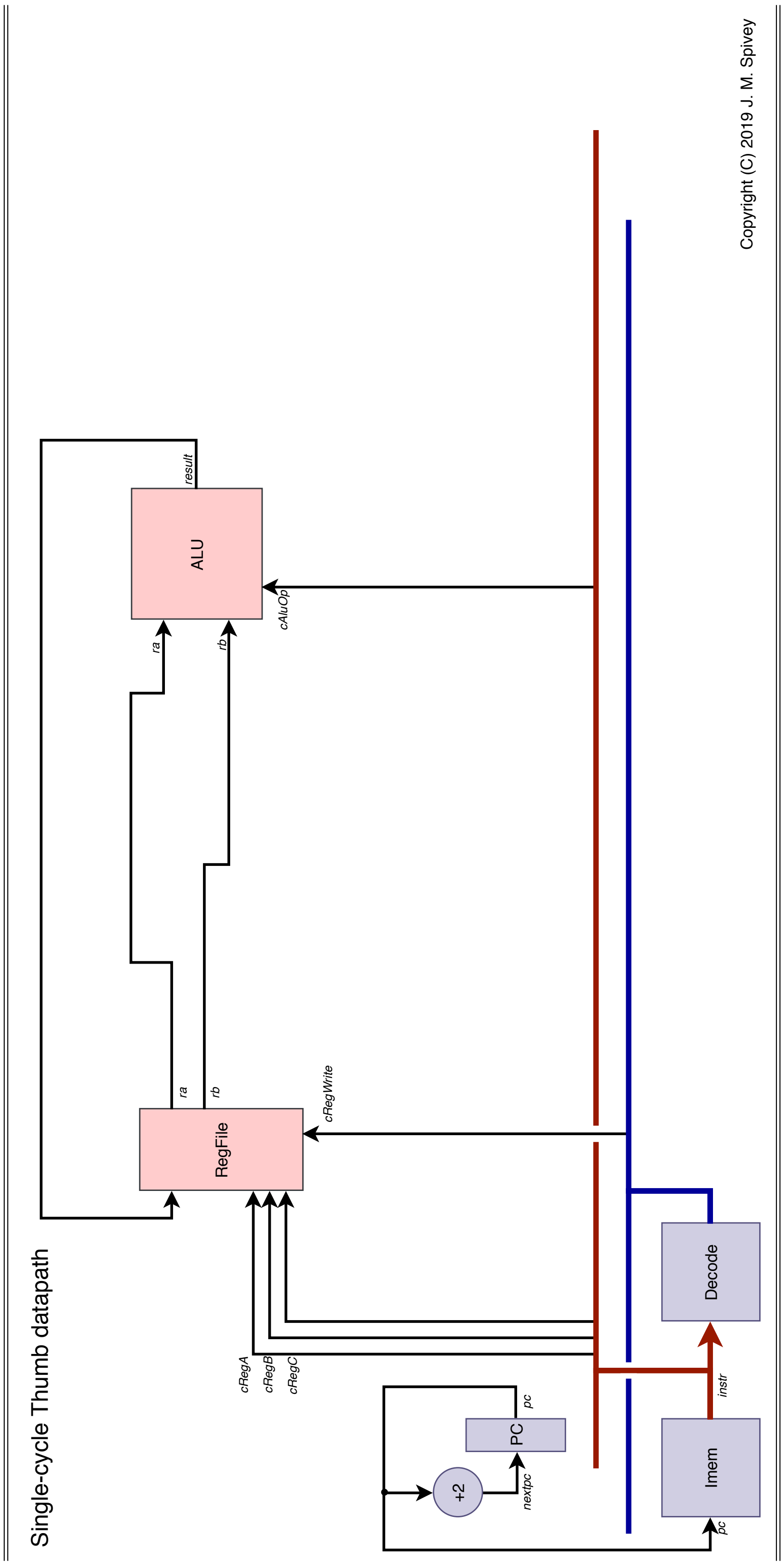

Figure 2: *ALU operations*

detail in decoding, and concentrate instead on what features are needed in the datapath itself.

3 Immediate operands

In addition to instructions that perform ALU operations with the operands taken from registers, there are also instructions that take the *second* operand from a field in the instruction. Examples of this are two forms of the add instruction:

adds ⟨Rx⟩,⟨Ry⟩,#⟨imm3⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Ry			Rx		

adds ⟨Rw⟩,#⟨imm8⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rw			imm8							

We can also cover the immediate form of the mov instruction, if we allow an ALU operation that simply copies the second operand and ignores the first.

movs ⟨Rw⟩,#⟨imm8⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rw			imm8							

To accommodate these, we can introduce a multiplexer on the second input of the ALU, fed both from the second register value *rb* and from appropriate fields of the instruction (Figure 3). The examples above show that it must provide the option of selecting both *instr*⟨8:6⟩ and *instr*⟨7:0⟩, and there are other possibilities we will discover as we proceed.

Now we have a few control signals, we can start to make a table (Figure 4) showing how they should be set to carry out various instructions. Each control signal takes a selection of discrete values that can be enumerated by small integers, and would be represented by bit patterns in a hardware representation, but we need not be concerned with which bit pattern corresponds to which function. For example, the multiplexer that feeds the second input of the ALU is capable of selecting between the register value *rb*, the 8-bit immediate field *instr*⟨7:0⟩ and the 3-bit immediate field *instr*⟨8:6⟩, as well as others to be introduced later. We will denote these initial three possibilities by *RegB*, *Imm8* and *Imm3*, hoping that these names are self-explanatory, and not caring what bit patterns are chosen to represent them. What's more, we will need a range of functions for the ALU, including the ones we denote here by *Add*, *Sub*, *And* and *Mov* – with *Mov* meaning that the output of the ALU is the same as its second input. We will add more functions as we need them.

This table will expand as we get further, adding more rows to provide more instructions, but also more columns to control the extra hardware we will need to implement them. There will always be settings for any added control signals that make the new hardware do nothing, so that we can still

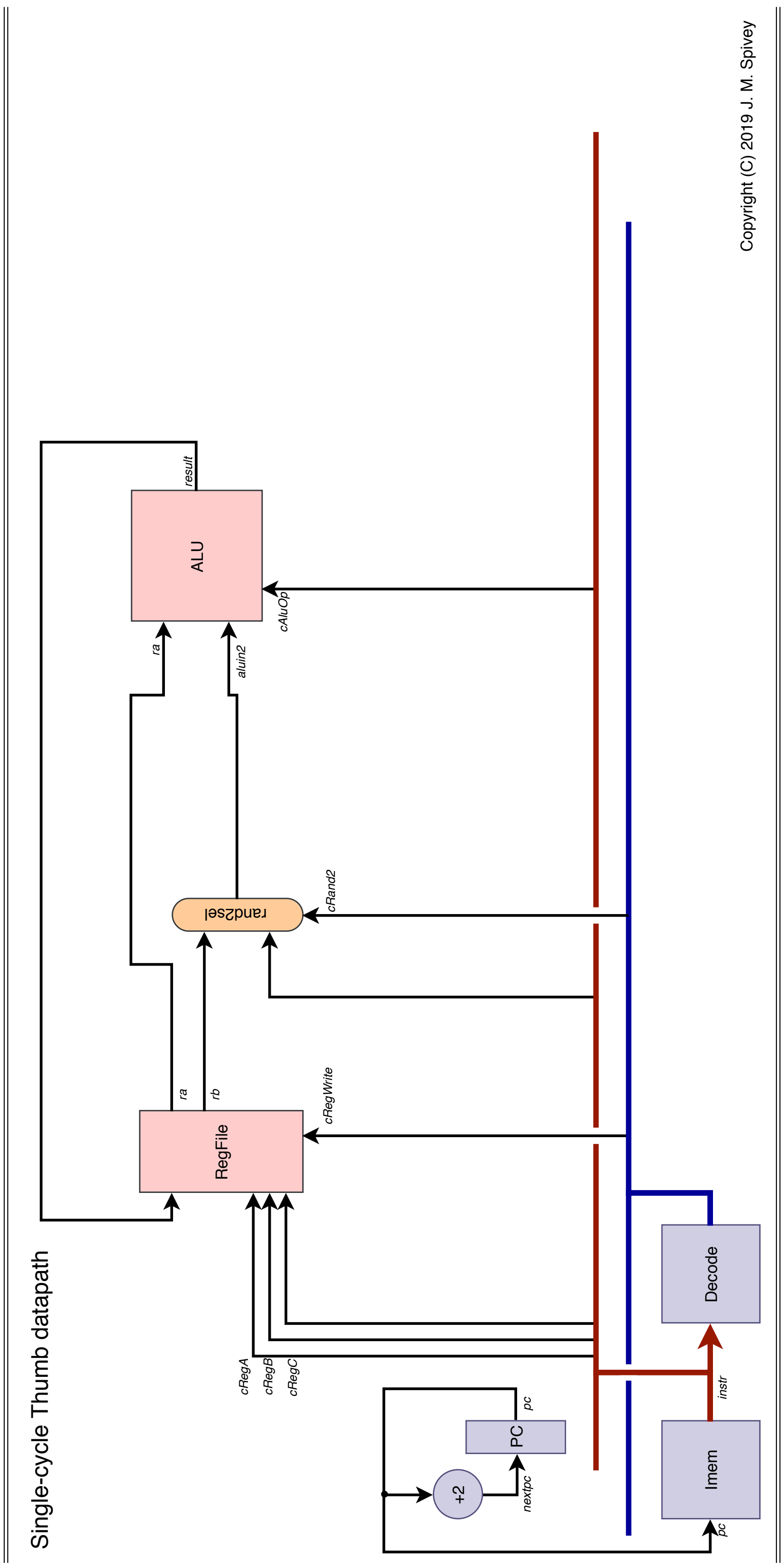

Figure 3: *Immediate operands*

Instruction	*cRand2*	*cAluOp*	*cRegWrite*
adds r	*RegB*	*Add*	*T*
adds i3	*Imm3*	*Add*	*T*
adds i8	*Imm8*	*Add*	*T*
subs r	*RegB*	*Sub*	*T*
subs i3	*Imm3*	*Sub*	*T*
subs i8	*Imm8*	*Sub*	*T*
ands r	*RegB*	*And*	*T*
movs r	*RegB*	*Mov*	*T*
movs i8	*Imm8*	*Mov*	*T*

Figure 4: *Decoding table with immediate operands*

get the effect of these early instructions unchanged. We've still to deal with the issue of what registers are selected to take part in the instructions, and we have also yet to provide for the fact these instructions all set the condition codes. And so far, all instructions write their result into a register: that will change too.

4 Data memory

Now let's add a second interface to memory, so that we can implement load and store instructions. We'll use what's called a *modified Harvard architecture*, meaning that the data memory will be treated separately from the instruction memory. They could be separate memories, as on some microcontrollers like the PIC, or we could imagine having two independent caches in front of the same memory, and modelling just the things that happen when there is a cache hit all the time, not the periods of suspended animation when the processor core is waiting for a memory transaction to complete. Either way, this is different from the *von Neumann architecture* of ARM's Cortex-M0 implementation, where there is one memory interface, and loads and stores are executed in an extra cycle between instruction fetches.

The Thumb instruction set provides instructions like ldr r0, [r1, r2] and str r0, [r1, r2] that form an address by adding the contents of two registers r1 and r2, and either load a memory word and save it in a third register r0, or take a value from r0 and store it.

ldr ⟨Rx⟩,[⟨Ry⟩,⟨Rz⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rz			Ry			Rx		

str ⟨Rx⟩,[⟨Ry⟩,⟨Rz⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rz			Ry			Rx		

We should notice two requirements for the datapath: first, that we need to form addresses by adding, and second, that the str instruction here reads

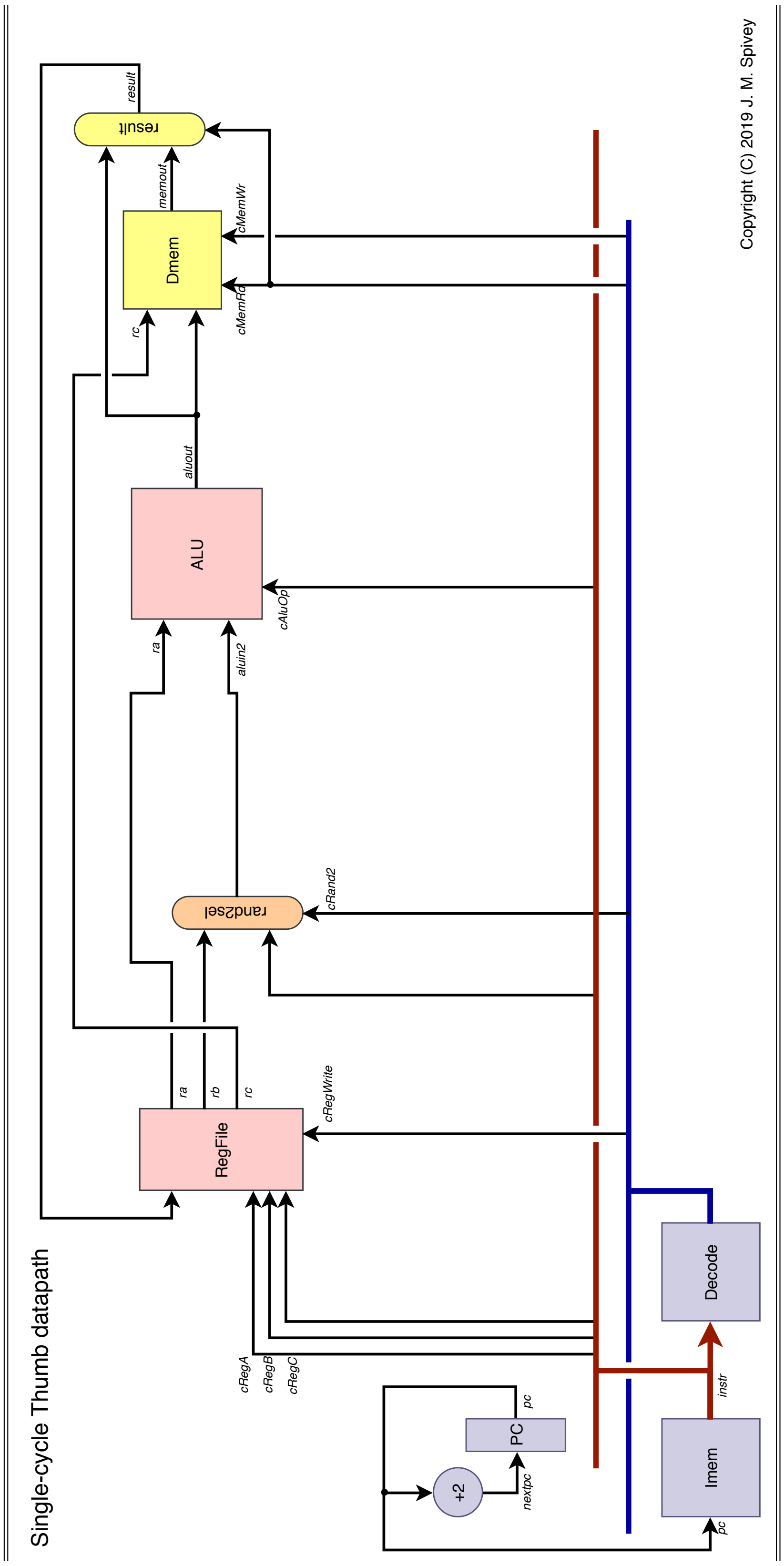

Figure 5: *Data memory*

Instruction	*cRand2*	*cAluOp*	*cMemRd*	*cMemWr*	*cRegWrite*
adds r	*RegB*	*Add*	*F*	*F*	*T*
movs i8	*Imm8*	*Mov*	*F*	*F*	*T*
ldr r	*RegB*	*Add*	*T*	*F*	*T*
str r	*RegB*	*Add*	*F*	*T*	*F*

Figure 6: *Decoding table for data memory*

not two but all three of the registers named in the instruction. We can use the existing ALU to do the address calculation, and we can easily enhance the register file with an extra mux so that it outputs the current value of all three named registers. If the third register value isn't needed (as in all instructions except a store) then it costs nothing to compute it anyway (Figure 5).

In addition to the third register value *rc*, the diagram shows two further architectural elements. There's the data memory *DMem*, with two data inputs, one data output and two control inputs, both single bits. The two data inputs are an address, taken from the ALU output, and a value to be stored, taken from *rc*. The data output is a value *memout* that has been loaded from memory, and this together with *aluout* feeds a new mux that determines the result of the instruction. The two control inputs for the memory are *cMemRd* and *cMemWr*, telling it whether to conduct a read cycle, a write cycle, or (if the instruction is not a load or store) neither. Writing when we don't want to write is obviously harmful, and reading unnecessarily might also be harmful if it causes a cache miss, or an exception for an invalid address. The result mux can be controlled by the same *cMemRd* signal, so that the result of the instruction is *memout* for load instructions and *aluout* for everything else.

Let's enhance the decoding table to cover these two new instructions, as shown in Figure 6. I've kept just a few of the existing instructions, extending the lines for them to include values for *cMemRd* = *cMemWr* = *F* that maintain the same function as before; the other instructions can be extended in the same way. I've added entries for the ldr and str with the reg+reg addressing mode. Note that str is the first instruction that *doesn't* write its result back to a register. There will be others, so while it was tempting before to suppose *cRegWrite* = *T* always, and it's tempting now to suppose *cRegWrite* = ¬*cMemWr*, we will see later that neither of these are true.

5 Barrel shifter

As the next step, let's add a barrel shifter to the datapath, so that we can implement shift instructions like the following.

lsls ⟨Rx⟩,⟨Ry⟩,#⟨imm5⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Ry			Rx		

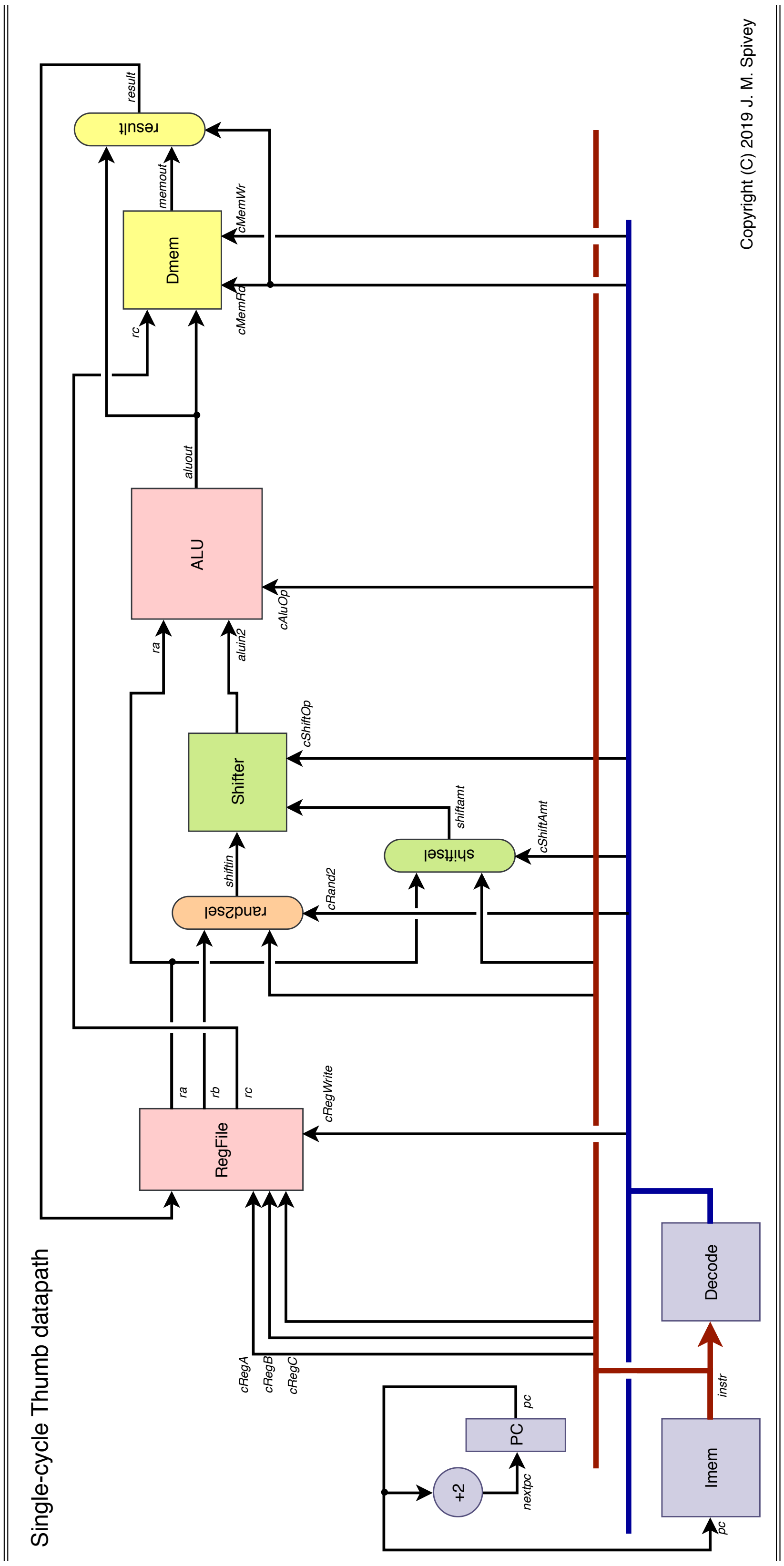

Figure 7: *Barrel shifter*

Instruction	*cRand2*	*cShiftOp*	*cShiftAmt*	*cAluOp*	*cMemRd*	*cMemWr*	*cRegWrite*
adds r	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*F*	*T*
movs i8	*Imm8*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*T*
ldr r	*RegB*	*Lsl*	*Sh0*	*Add*	*T*	*F*	*T*
str r	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*T*	*F*
lsls i5	*RegB*	*Lsl*	*ShImm*	*Mov*	*F*	*F*	*T*
rors r	*RegB*	*Ror*	*ShReg*	*Mov*	*F*	*F*	*T*
ldr i5	*Imm5*	*Lsl*	*Sh2*	*Add*	*T*	*F*	*T*
str i5	*Imm5*	*Lsl*	*Sh2*	*Add*	*F*	*T*	*F*

Figure 8: *Decoding table for barrel shifter*

rors ⟨Rx⟩,⟨Ry⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Ry			Rx		

We could make a barrel shifter part of the ALU, so that left and right shifts were added to the list of ALU operations; or failing that, we could put the shifter 'in parallel' with the ALU, feeding its output together with those of the ALU and the data memory into the result mux. That would allow us to implement the shift instructions OK, but it would be less versatile. We can take a glance at big-ARM instructions like

```
ldr r0, [r1, r2, LSL #2].
```

This shifts left by 2 bits the value of r2, adds that to the value of r1 to form an address, loads from that address, and puts the result in r0, all in one instruction. This is really useful if r1 contains the base address of an array and r2 contains an index into the array. Sadly, Thumb code doesn't have a way to encode all that in one instruction. We can nevertheless provide for such operations by adding a barrel shifter in front of the ALU, operating on the value that will become the ALU's second input (Figure 7). There is a control signal to set the operation – *Lsl*, *Lsr*, *Asr* or *Ror* – to be performed by the shifter. There's also a mux that lets us choose the shift amount, either a constant like 0 or 2 (*Sh0* or *Sh2*), or an immediate field of the instruction (*ShImm*), or a value taken from the first register *ra* read by the instruction (*ShReg*).

There are two new control signals, *cShiftOp* and *cShiftAmt*. Existing instructions will continue to work if we set *cShiftOp* = *Lsl* and *cShiftAmt* = *Sh0*, representing the constant 0.

ldr ⟨Rx⟩,[⟨Ry⟩,#⟨imm5⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Ry			Rx		

str ⟨Rx⟩,[⟨Ry⟩,#⟨imm5⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Ry			Rx		

We can make good use of the shifter in implementing load and store instructions with the reg+imm addressing mode, because it is specified that the offset should be multiplied by 4 in such instructions, and we can get the shifter to do the multiplication. Figure 8 shows the control signals for some existing instructions, plus the two shift instructions and the reg+imm forms of ldr and str.

The disadvantage of putting the barrel shifter 'in series' with the ALU is that it lengthens the combinational paths, one of which now stretches from the register file, through shifter, ALU and data memory, back to the register file. The long path will slow the maximum clock rate that can be supported by the machine.

6 PC as a register

Up to now, we have kept the PC separate from the general register files, as indeed it is on some architectures. But on the ARM, the PC can be accessed like other registers, and used in PC-relative addressing. So our next step is to merge the PC into the register file, using a design for a 'turbocharged' register file we prepared earlier (Figure 9). In addition to the three selectable outputs that can output the value of any register (including the PC), there is now a special-purpose output that always carries the (current) PC value. There is also a separate input that receives the value of PC+2, which will be written to the PC if it is not specifically selected for the writing of a different value: this allows for the implementation of branch instructions at a later point. We'll assume that the design of the register file includes an adjustment so that, when the PC is read as an ordinary register, the specified value PC+4 is output.

7 Instruction decoding

We've got quite a good table of control signals now, so it's time to fill in more details of the decoding process (Figure 10). Each instruction uses up to three registers, sometimes selected by fields in the instruction, and sometimes fixed as SP or LR or PC. To sort this out, we can add three identical multiplexers, driven by control signals *cRegSelA*, *cRegSelB*, *cRegSelC*, that either select one of a list of fields (*Rx*, *Ry*, *Rz*, *Rw*) from the instruction or give a fixed value (*Rsp*, *Rlr*, *Rpc*). The other addition in this figure is a unit called *alusel*. This tidies up a couple of pairs instructions where the ALU operation could be add or subtract, but the decision is not determined by the first few bits of the instruction. One such pair are the forms add/sub r/i3 shown earlier that use bit *instr*⟨10⟩ to decide whether the second operand is a register or a 3-bit immediate field, and use bit *instr*⟨9⟩ to decide between adding and subtracting. The alusel decoder sorts out the details, and the details can be found in the code of the simulator.

Let's refresh the table of control signals, adding the three register selec-

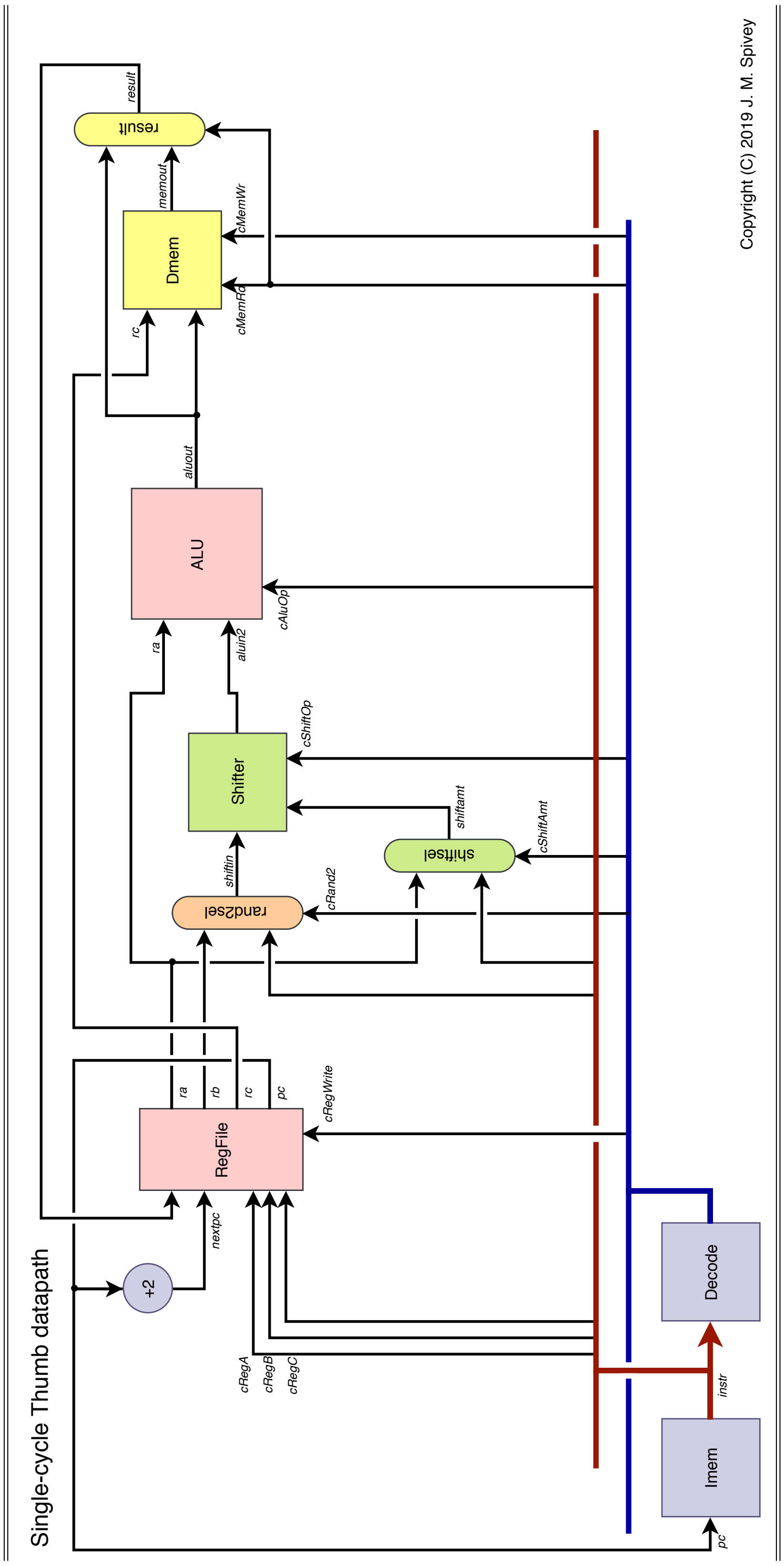

Figure 9: *Turbocharged register file*

tors (see Figure 11. We'll do that first for our existing selection of example instructions first, a fair selection of instructions for the machine, especially if we let ands stand as an example of register-to-register ALU operations and rors stand as an example of shifts with the shift amount taken from a register. Most instructions can be identified from their first five bits and set out in a table of 32 possibilities. Among those implemented in the simulator, most of the rest start with 010000 or 010001, and can be identified using two further, smaller tables. All these tables could become ROMs in a hardware implementation. This part of a Thumb implementation is more complicated than is typical for RISC machines because of the variety of instruction formats. For comparison, the MIPS has just three instruction formats, all 32 bits long: one that names three registers, one that has two registers and a 16-bit immediate field, and a third format with a large offset for subroutine calls.

The datapath as it stands also contains the resources to implement a number of other instructions. For example, there is are several instructions that implicitly involve the stack pointer, including ones that add or subtract a constant from the stack pointer, using bit *instr*⟨7⟩ to decide which.

add sp,sp,#⟨imm7⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

sub sp,sp,#⟨imm7⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

There are also an instruction that forms an address by adding a constant and the stack pointer, and instructions that load and store from that address.

add ⟨Rw⟩,sp,#⟨imm8⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rw			imm8							

ldr ⟨Rw⟩,[sp,#⟨imm8⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rw			imm8							

str ⟨Rw⟩,[sp,#⟨imm8⟩]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rw			imm8							

All of these can be implemented using the register selector *Rsp*, as shown in the next few lines of Figure 11.

We can also implement unconditional branches that add a signed 11-bit constant to the PC.

b ⟨disp11⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	disp11										

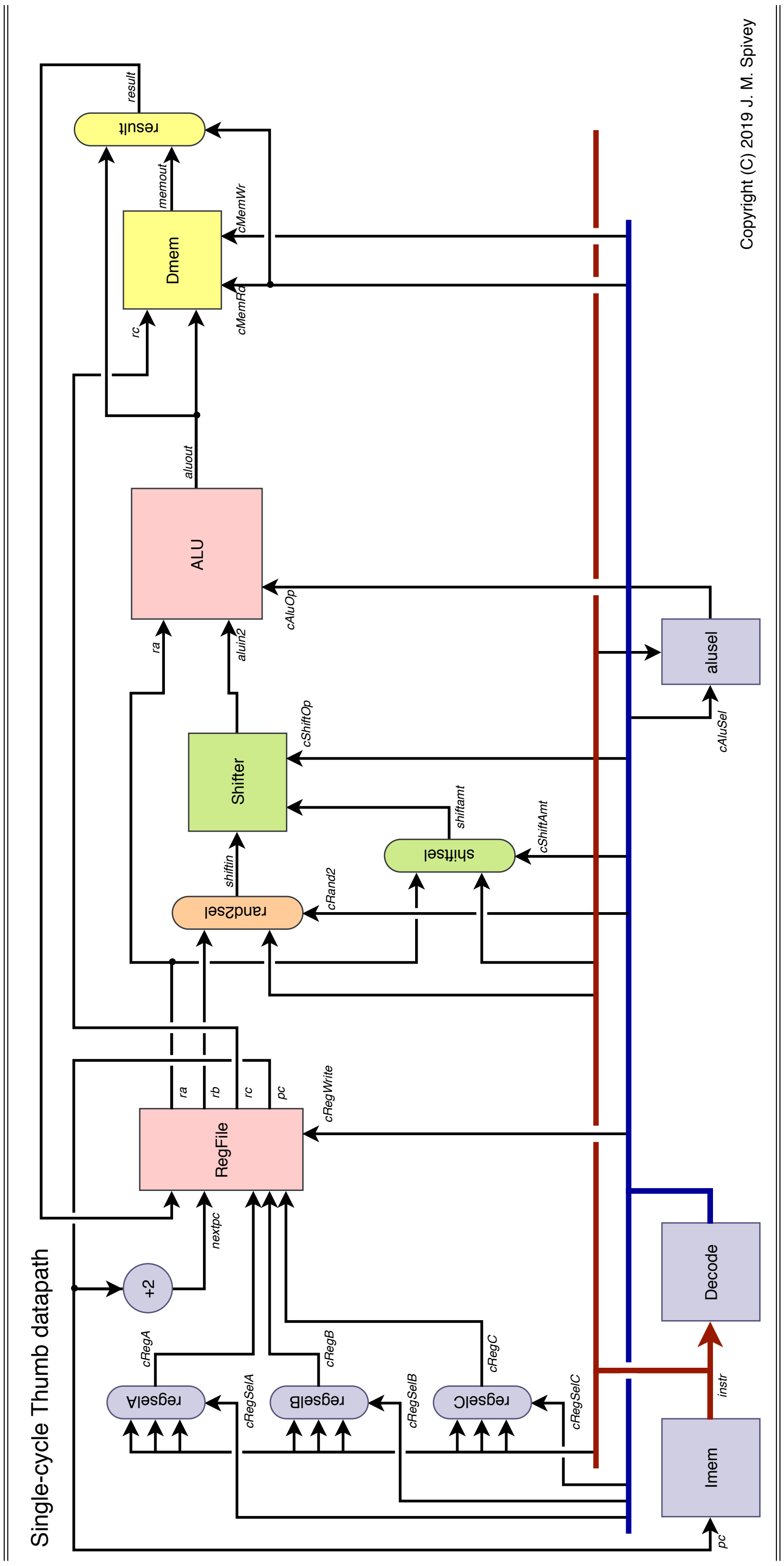

Figure 10: *Instruction decoding*

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cRegWrite*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*	
adds/subs r/i3	*Ry*	*Rz*	*Rx*	*RImm3*	*Lsl*	*Sh0*	*Sg9*	*F*	*F*	*T*
movs i8	–	–	*Rw*	*Imm8*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*T*
ldr r	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Add*	*T*	*F*	*T*
str r	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*T*	*F*
lsls i5	–	*Ry*	*Rx*	*RegB*	*Lsl*	*ShImm*	*Mov*	*F*	*F*	*T*
ands r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*And*	*F*	*F*	*T*
rors r	*Ry*	*Rx*	*Rx*	*RegB*	*Ror*	*ShReg*	*Mov*	*F*	*F*	*T*
ldr i5	*Ry*	–	*Rx*	*Imm5*	*Lsl*	*Sh2*	*Add*	*T*	*F*	*T*
str i5	*Ry*	–	*Rx*	*Imm5*	*Lsl*	*Sh2*	*Add*	*F*	*T*	*F*
add/sub sp	*Rsp*	–	*Rsp*	*Imm7*	*Lsl*	*Sh2*	*Sg7*	*F*	*F*	*T*
add rsp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*F*	*F*	*T*
ldr sp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*T*	*F*	*T*
str sp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*F*	*T*	*F*
b	*Rpc*	–	*Rpc*	*SImm11*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*T*

Figure 11: *Decoding table for register selection*

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cRegWrite*	*cWLink*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*		
bx/blx r	–	*Ryy*	*Rpc*	*RegB*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*T*	*C*
bl1	*Rpc*	–	*Rlr*	*SImm11*	*Lsl*	*Sh12*	*Add*	*F*	*F*	*T*	*N*
bl2	*Rlr*	–	*Rpc*	*Imm11*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*T*	*Y*

Figure 12: *Decoding table for subroutine calls*

The last line of the table shows the control signals that are needed. As you can see, the displacement is multiplied by 2 before adding it to the PC. This implementation depends on two features of the register file: that the PC reads as PC+4 when accessed as a numbered register, and that writing the PC explicitly takes precedence over the usual update with *nextpc* = *pc* + 2.

8 Subroutine calls

In order to implement the branch-and-link instructions bl and blx, we need one extra feature of the register file, and one small enhancement to the datapath. The register file has one further control input *cLink* that, when active, causes the *nextpc* value to be written to the link register LR, in addition to the normal updating of registers (Figure 13). This will permit us to implement an instruction that simultaneously sets the link register to a return address while loading the entry point of a subroutine into the PC.

The bx r and blx r instructions contain a 4-bit register field $instr\langle 6{:}3\rangle$ that I will denote by *Ryy*: it permits the naming of the high registers r8 ... r15

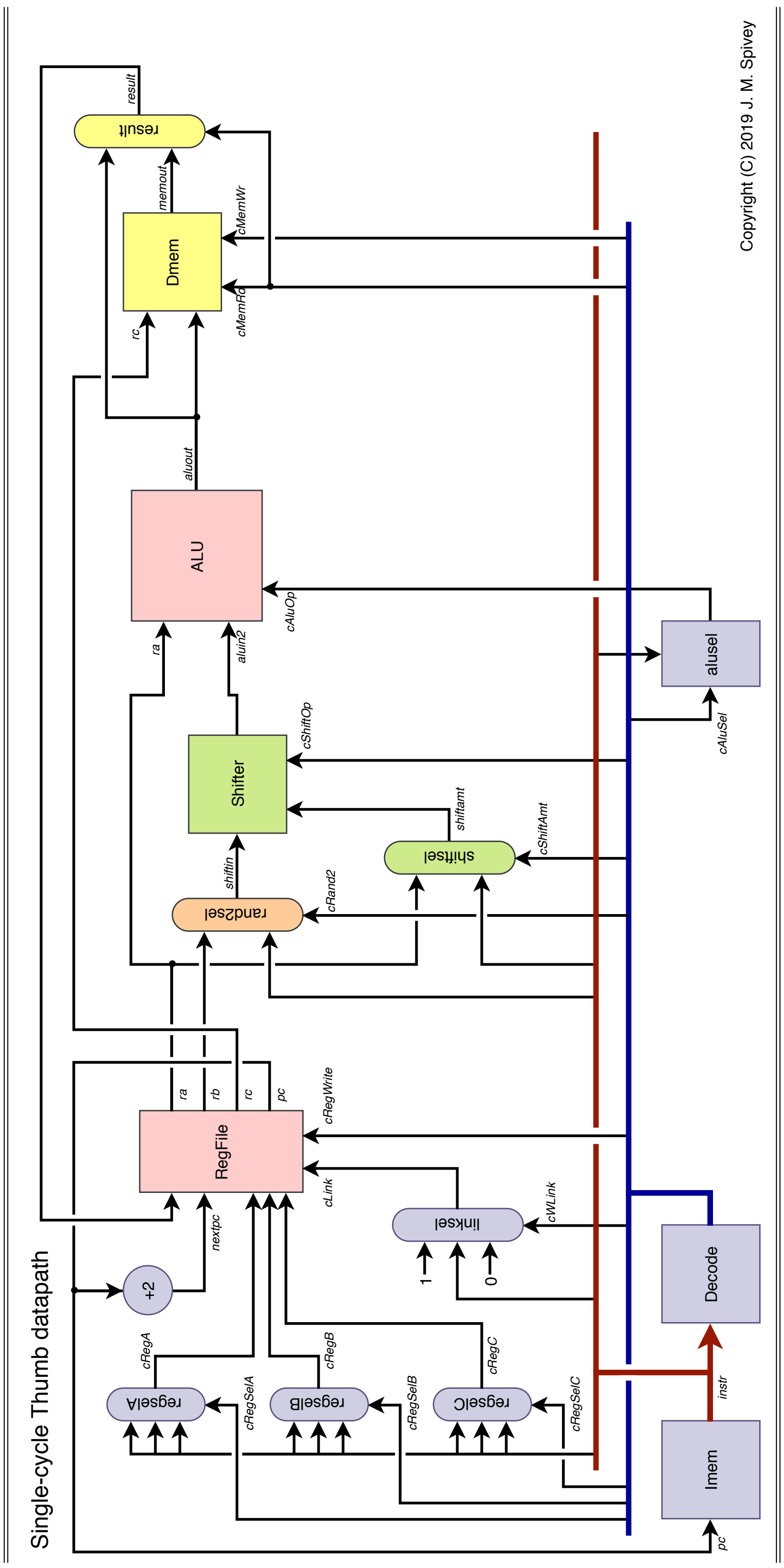

Figure 13: *Subroutine calls*

in addition to the low registers r0 . . . r7: in particular, we can use LR as the branch target as part of the action of returning from a subroutine.

bx ⟨Ryy⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Ryy				0	0	0

blx ⟨Ryy⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Ryy				0	0	0

Because the bx r and blx r instructions differ in only one bit, we need an extra multiplexer to derive the *cLink* control signal, with three settings – 0 for most instructions, 1 for the bl2 instruction (see below), and a copy of *instr*⟨7⟩ for the bx and blx instructions. The three values of the *cWLink* control signal for the multiplexer are denoted N, Y and C in Figure 12. Existing instructions are extended with *cWLink* = N, and the rule shown in the table covers the two branch-to-register instructions.

As hinted in an earlier problem sheet, the 32-bit bl instruction can, in simple cases, be executed in two halves that we shall call bl1 and bl2.

bl1, ⟨simm10⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10									

bl2, ⟨imm11⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	J1	1	J2	imm11										

The first half starts with bits 11110, and the second half starts with 11111, provided we assume bits J1 and J2 are both 1, as traditionally they were: only very long branches will make them anything else. We will implement the bl1 instruction by adding the offset, scaled appropriately, to the PC and putting the result in LR; then we implement bl2 by adding the second half of the offset to the LR, and putting the result in the PC, simultaneously setting LR to the return address.

9 Conditional execution

There are several features of the Thumb instruction set that we're not going to implement, but one remains that is essential to writing working programs, and that is the mechanism for conditional branches: arithmetic instructions set the status bits NZCV, and there is a form of branch instruction that contains one of 14 conditions defined as logical combinations of the status bits.

b⟨c⟩ ⟨imm8⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

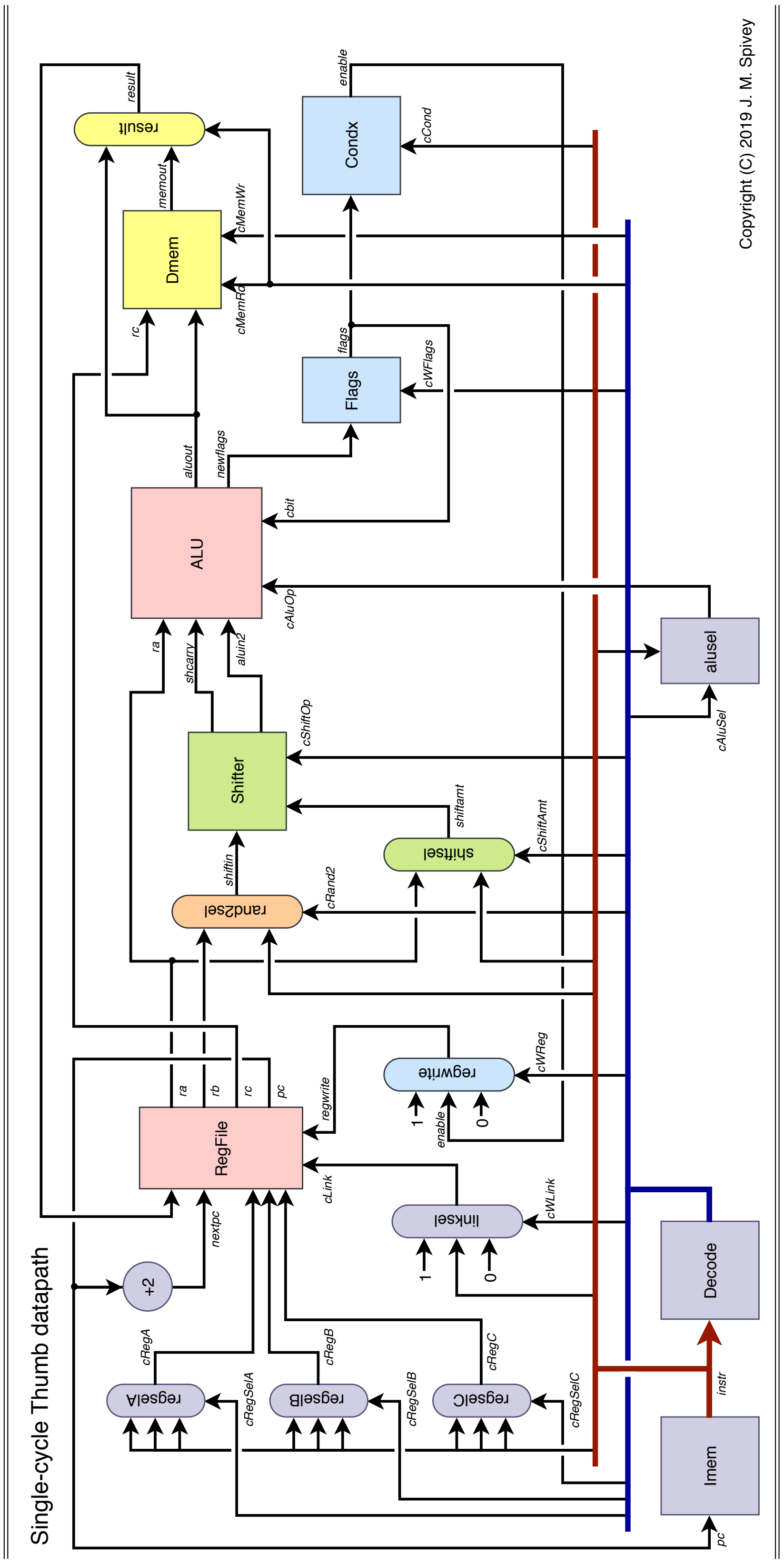

Figure 14: *Conditional execution*

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cWFlags*		*cWLink*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*		*cWReg*	
b⟨c⟩	*Rpc*	-	*Rpc*	*SImm8*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*F*	*C*	*N*
subs ri	*Rw*	-	*Rw*	*Imm8*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*Y*	*N*
cmp ri	*Rw*	-	-	*Imm8*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*N*	*N*

Figure 15: *Decoding table for conditional branches*

Our approach to implementing conditional branches will be to use the ALU to compute the target address of the branch from the PC value and the displacement, but to make the writing of the result into the PC conditional on the test being passed. Three new datapath elements and two new control signals will be needed. There is a small, 4-bit register to hold the NZCV flags, and a control signal that determines whether the flags are updated by each instruction. A separate, combinational circuit takes the register contents and the four-bit condition field shown in the instruction format, and computes a signal *enable* that indicates whether the condition is satisfied (Figure 14). As usual, this circuit functions in every instruction, whether it is a conditional branch or not, and produces a nonsense output except when a conditional branch is in progress. The decision whether to write the result of an instruction back to a register becomes a dynamic one: in place of the signal *cRegWrite* appearing in the decoding table, there is a signal *cWReg* that takes values *Y*, *N* and *C*, with *C* denoting that the *cRegWrite* signal is taken from *enabled*.

We can add the new signals to the table for existing instructions (Figure 15): *cWFlags* indicates whether the instruction should write the flags or not: *T* for adds and lsls and all the other arithmetic and logical instructions, *F* for loads and stores and branches. The values *T* and *F* for *cRegWrite* are replaced by values *Y* and *N* for *cWReg*, with *C* used only in the following rule for conditional branch instructions.

For comparison, the rules for subs ri and cmp ri are also given in the table: the only difference between them is that after the subtraction is performed, the subs instruction writes a register as well as updating the flags, and the cmp instruction just updates the flags. It's important to note that conditional branches read but don't destroy the flags: that makes it possible to have one compare instruction followed by several branches conditional on its result.

Context: *None of the detail here really matters, except as an illustration of the challenges faced by a datapath designer. The hardware, once designed, is fixed, and must be designed so that, with appropriate settings of the control signals, every machine language instruction can be implemented. For a single-cycle design like this one, where each instruction takes exactly one clock cycle, the instruction decoder essentially expands each instruction into a long string of control bits, reversing a kind of compression that makes some useful operations expressible in the limited number of instruction bits, and others not expressible at all.*

10 Summary of control signals

Figure 16 shows a summary of the named signals in the datapath. We can distinguish between *decoded* signals, which are determined by the opcode as looked up in the ROM, and therefore the same for all instances of an instruction, *derived* signals, which are the same for every execution of a particular instruction, and *dynamic* signals, which will differ from one execution of an instruction to the next. Figure 17 shows the values of the decoded signals for each instruction implemented in the simulator.

Decoded	Derived	Dynamic	Description
		pc	Program counter value
	instr		The 16-bit instruction
cRegSelA, cRegSelB, cRegSelC			Three rules for selecting registers
	cRegA, cRegB, cRegC		The three register numbers
		ra, rb, rc	The contents of the three registers
		nextpc	Address of the next instruction
cRand2			Rule for selecting shifter input
		shiftin	Input to the shifter
cShiftOp			Shift operation
cShiftAmt			Rule for determining shift amount
		shiftamt	Amount of shift
		aluin2, shcarry	Outputs from the shifter
cAluSel			Rule for determining ALU operations
	cAluOp		The ALU operation
		aluout, newflags	Outputs from the ALU
cMemRd, cMemWr			Whether to read or write the memory
		memout	Result of memory read
		result	Result for write-back
cWFlags			Whether to update the flags
	cCond		Condition to test
		enable	Whether the condition is satisfied
cWReg			Rule for writing result register
		regwrite	Whether result will be written
cWLink			Rule for updating link register
	cLink		Whether link register will updated

Figure 16: *Signals in the datapath*

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cWFlags*		*cWLink*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*		*cWReg*	
lsls i5	–	*Ry*	*Rx*	*RegB*	*Lsl*	*ShImm*	*Mov*	*F*	*F*	*T*	*Y*	*N*
lsrs i5	–	*Ry*	*Rx*	*RegB*	*Lsr*	*ShImm*	*Mov*	*F*	*F*	*T*	*Y*	*N*
asrs i5	–	*Ry*	*Rx*	*RegB*	*Asr*	*ShImm*	*Mov*	*F*	*F*	*T*	*Y*	*N*
adds/subs i3	*Ry*	*Rz*	*Rx*	*RImm3*	*Lsl*	*Sh0*	*Bit9*	*F*	*F*	*T*	*Y*	*N*
movs i8	–	–	*Rw*	*Imm8*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*T*	*Y*	*N*
cmp i8	*Rw*	–	–	*Imm8*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*N*	*N*
adds i8	*Rw*	–	*Rw*	*Imm8*	*Lsl*	*Sh0*	*Add*	*F*	*F*	*T*	*Y*	*N*
subs i8	*Rw*	–	*Rw*	*Imm8*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*Y*	*N*
ands r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*And*	*F*	*F*	*T*	*Y*	*N*
eors r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Eor*	*F*	*F*	*T*	*Y*	*N*
lsls r	*Ry*	*Rx*	*Rx*	*RegB*	*Lsl*	*ShReg*	*Mov*	*F*	*F*	*T*	*Y*	*N*
lsrs r	*Ry*	*Rx*	*Rx*	*RegB*	*Lsr*	*ShReg*	*Mov*	*F*	*F*	*T*	*Y*	*N*
asrs r	*Ry*	*Rx*	*Rx*	*RegB*	*Asr*	*ShReg*	*Mov*	*F*	*F*	*T*	*Y*	*N*
adcs r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Adc*	*F*	*F*	*T*	*Y*	*N*
sbcs r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Sbc*	*F*	*F*	*T*	*Y*	*N*
rors r	*Ry*	*Rx*	*Rx*	*RegB*	*Ror*	*ShReg*	*Mov*	*F*	*F*	*T*	*Y*	*N*
tst r	*Rx*	*Ry*	–	*RegB*	*Lsl*	*Sh0*	*And*	*F*	*F*	*T*	*N*	*N*
negs r	–	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Neg*	*F*	*F*	*T*	*Y*	*N*
cmp r	*Rx*	*Ry*	–	*RegB*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*N*	*N*
cmn r	*Rx*	*Ry*	–	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*F*	*T*	*N*	*N*
orrs r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Orr*	*F*	*F*	*T*	*Y*	*N*
muls r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Mul*	*F*	*F*	*T*	*Y*	*N*
bics r	*Rx*	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Bic*	*F*	*F*	*T*	*Y*	*N*
mvns r	–	*Ry*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Mvn*	*F*	*F*	*T*	*Y*	*N*
add hi	*Rxx*	*Ryy*	*Rxx*	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*F*	*F*	*Y*	*N*
cmp hi	*Rxx*	*Ryy*	–	*RegB*	*Lsl*	*Sh0*	*Sub*	*F*	*F*	*T*	*N*	*N*
mov hi	–	*Ryy*	*Rxx*	*RegB*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*F*	*Y*	*N*
bx/blx r	–	*Ryy*	*Rpc*	*RegB*	*Lsl*	*Sh0*	*Mov*	*F*	*F*	*F*	*Y*	*C*
ldr pc	*Rpc*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Adr*	*T*	*F*	*F*	*Y*	*N*
str r	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Add*	*F*	*T*	*F*	*N*	*N*
ldr r	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*Sh0*	*Add*	*T*	*F*	*F*	*Y*	*N*
str i5	*Ry*	–	*Rx*	*Imm5*	*Lsl*	*Sh2*	*Add*	*F*	*T*	*F*	*N*	*N*
ldr i5	*Ry*	–	*Rx*	*Imm5*	*Lsl*	*Sh2*	*Add*	*T*	*F*	*F*	*Y*	*N*
str sp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*F*	*T*	*F*	*N*	*N*
ldr sp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*T*	*F*	*F*	*Y*	*N*
add pc	*Rpc*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Adr*	*F*	*F*	*F*	*Y*	*N*
add sp	*Rsp*	–	*Rw*	*Imm8*	*Lsl*	*Sh2*	*Add*	*F*	*F*	*F*	*Y*	*N*
add/sub sp	*Rsp*	–	*Rsp*	*Imm7*	*Lsl*	*Sh2*	*Bit7*	*F*	*F*	*F*	*Y*	*N*
b⟨c⟩	*Rpc*	–	*Rpc*	*SImm8*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*F*	*C*	*N*
b	*Rpc*	–	*Rpc*	*SImm11*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*F*	*Y*	*N*
bl1	*Rpc*	–	*Rlr*	*SImm11*	*Lsl*	*Sh12*	*Add*	*F*	*F*	*F*	*Y*	*N*
bl2	*Rlr*	–	*Rpc*	*Imm11*	*Lsl*	*Sh1*	*Add*	*F*	*F*	*F*	*Y*	*Y*

Figure 17: *Decoding table for all instructions*