DECA Lab

Part 5: ARM-lite

Department of Electrical and Electronic Engineering

Imperial College London

v1.1

Spring 2020

Contents

1. Introduction

The next one week lab (and one catchup week) will be used to extend your MU0 architecture from Lab 4. At the Instruction Set Architecture level the changes are to add a single (complex) new two-operand instruction called ARMish. At hardware level the changes are:

- Replace `Acc` by 4 registers `R0-R3`, where previous instructions using `Acc` will use `R0`.
- Add a new *two operand* instruction of the form `Ra := Ra op Rb`
- Add a `CARRY` status bit, implemented as a flip-flop. Implement instructions that add with carry in from `CARRY`, and that write `CARRY` based on the adder carry out.
- Implement fields in the new instruction word that write and use `CARRY`

In order to speed work up you are given the new hardware already implemented except for the control signals for CARRY.

2. Overview

The ARMish instruction you add illustrates many of the features found in modern ISAs, and specifically the ARM data processing instructions, in a RISC design which is relatively simple to implement. It is an interesting example of the strength of RISC design philosophy as discussed in the lectures. You will be able to see that typical programs run much, much quicker on the new instruction set, and the hardware to implement it is not complex because it is very uniform.

The ARMISH instruction word (bits 13:0) divides into 6 independent fields each of which controls separate hardware. For example, the `CIN` field controls the carry input to the ALU, the `S` field controls writing of the `CARRY` flip-flop, and the `COND` field controls the input to the `SKIP` flip-flop.

2.1. Differences from ARM data Processing

If you compare this instruction with the ARM data processing instruction as described in lectures there are differences as well as similarities:

- `S` field, and `CARRY` works in a similar way.
- instead of different opcodes (ADC/ADD) controlling ALU carry in, the `CIN` field does this. This is actually simpler than the `ARM` method, and also provides more options.
- The (not implemented except as a possible extension) `COND` field works in a way similar to the ARM `COND` field but with important differences. The ARM `COND` applies to the current instruction, and allows each instruction to make itself conditional. That would have been possible to implement in ARMish instructions, but then they could not control MU0 instructions, and particularly MU0 jumps. ARMish delayed skip thus fits better with the MU0 ISA even though it is less good just considering the ARMish instructions.

3. Work During Lab 5

Before the lab

In this lab you will replace the single `Acc` register by a *register file* containing 4 registers `R0 - R3`, and an ALU that implements two operand instructions. `CARRY`, `SKIP` and their associated logic can be implemented in the next session.
In order to make this work faster you are given, complete, a working block schematic and verilog design for the register file connected to a Verilog working ALU.
Before the lab your task is to review the information in this handout and understand how the register file hardware operates.

- Look in Figure 5, a schematic of the dual port RAM with 4 locations. Each port has its own set of (two) address lines. The RAM locations are implemented using 4 D registers `R0-R3`. One 4 output demultiplexer `DEMUX4` controls writing. Two 4 input multiplexers, `MUX4A, MUX4B` each allow an register to be read. This RAM has two independent ports, one which will read and (if required) write to a register, the other of which will read a register.
  - If `Port1Addr,Port2Addr` are correct during `EXEC1`, in which cycle is the corresponding RAM data out valid on `Port1Q` or `Port2Q`? See Appendix B for the answer.
  - If `Wen` is 1, can the `Port1Addr` register be read and written at the same time? If it is written in `EXEC1` when does its Q output change to be the newly written value? See Appendix B for the answer.
- Look here (or in Figure 6) at your register file schematic. It is similar to the dual port RAM, except that there are additional inputs (`R0wen,R0din`), outputs (`R0q`), and logic (`MUX2, G1`). The purpose of these is so that you can keep your old instructions using `R0` in place of `Acc`. The `R0wen,R0din,R0q` ports on regfile will replace the previous `Acc` ports `en, data, q`. Note that you do not have an `sload` input. The new R0 register operates as `lpm_shiftreg` with `sload` always high. You cannot implement the `LSR` instruction but that does not matter since the new instruction will do this and more.
- Trace through the logic implemented by `BUSMUX` and `G1` separately in the two cases `R0wen=1` and `R0wen=0`. Do you understand why this works? If not ask GTAs when you are in the lab.
- Look at Figure 3 to see how the register file connects with a Verilog combinational ALU to implement instructions of the form: `Rd := Rd op Rs`.
- Read Appendix A for an overview of how the dual port RAM is used.
- Read Appendix D for an overview of fields in the new `ARMih` instruction. Note you may ignore the `COND` field since implementing this is not required.

☐ **Task 1.** *If you have been working separately from your lab partner choose whichever Lab4 design works best and copy this for both of you. Follow the instructions in Figure 1 to create a new Lab5 project which is a copy of the lab4 work and add the register file and ALU block to the Lab5 project.*

☐ **Task 2.** *Connect the inputs on the* `top` *block.*

- *Connect* `EXEC1` *to* `EXEC1`*.*
- *Connect* `instr` *to* `IR'` *(16 bits).*

☐ **Task 3.** *Implement a boolean expression in the Verilog ALU to control write enable of the new registers* `wenout` *when used by ARMish instructions. The registers are written in the* `EXEC1` *cycle of every ARMish instruction - but must NOT be written during execution of normal MU0 instructions. Note that there is a separate enable* `r0wen` *that is used by the MU0 instructions to write to R0 (which takes the place of* `Acc`*).*

☐ **Task 4.** *Use the datapath test program from Lab 4 to check that your MU0 instructions still work with the register file taking the place of* `Acc`*. The results will not be identical because* `LSR` *will not work. You may change your existing decode logic for* `Acc enable` *so that* `Acc` *does not change during an* `LSR` *instruction.*

☐ **Task 5.** *Note Figure 9 which is a quick reference allowing you to compose ARMish instructions in hex easily. Test your new ALU instructions, without Carry, using the tests in Figure 2. To create a MIF file in hex you can use the Quartus* ***File → New file → Create MIF file****.*

1. In Quartus open your lab 4 project and archive it: *project* → *archive* will create a `*.qar` file from the entire project can be reconstructed.
2. Unarchive this file (separately if you want independent copies) to a new Lab5 quartus project in an empty directory. Download the provided block schematic register file and alu regfile.zip. Unzip this into 3 files: `regfile.bdf, top.bdf, alu.v`, and upload these using mobaXterm to the Lab5 project directory.
3. In Lab5 *Project→add/remove files→ add remove files → add all.* This will add top.bdf, regfile.bdf, alu.v to the project.
4. Open the `top.bdf` schematic sheet.
5. *File→create→create symbol files for current file*
6. Open the main MU0 schematic.
7. Delete your existing LPM_SHIFTREG `Acc` block.
8. Add the created symbol from `top.bdf`, found under *project* in the symbol tool, to your schematic, to the MU0 schematic. Call it (instance property) REGFILE.
9. Connect `R0din, R0q, R0wen` to the corresponding busses and signal in your old design where `Acc` was connected, using connection by name as necessary to keep the schematic readable. Note that the old `Acc sload` decoder signal is no longer used, and `LSR` MU0 instructions will no longer work.

Figure 1: Replacing Acc by Regfile and ALU on the MU0 schematic

Assembler	Hex
`LDI 0x123`	`0x8123`
`ARM MOV R2, R0`	`0xC028`
`LDI 0x111`	`0x8111`
`ARM MOV R3, R0`	`0xC02C`
`ARM ADD R3 R2`	`0xC00E`

Figure 2: Tests without CARRY

tool. When using this the tool to enter instruction values set ***View → memory radix → hexadecimal****.*
The new ARMish ***XSR*** *instruction, when used with* ***Rd = Rs = R0****, can be used to replace MU0* ***LSR****. Using Figure 7 and 9 determine the hex value of the ARMish instruction that will do this?*

□ **Task 6.** *Implement in the Verilog ALU the boolean expressions required to write* ***CARRY*** *as specified by the* ***S*** *bit in Figure 7. The necessary signals are:*

- ***cin****. The carry in to the ALU adder from* ***CIN*** *field.*
- ***carryen****. Controls writing CARRY, derived from* ***S****, the ARMish opcode bits, and timing info from* ***EXEC1****.*
- ***shiftin****. This determines the MSB of the result in the special case of* ***XSR****, as in Figure 7. Note that* ***shiftin*** *is just* ***cin*** *for the specified instructions.*
- ***carryout****. To make CARRY work properly in right shifts (XSR) carryout must equal* ***rsdata[0]*** *in XSR and ALU cout otherwise. You may wish to use Verilog conditional operator to simplify this: condition ? thenpart : elsepart.*

Which of these four signals are don't care for MU0 instructions, and outside ***EXEC1****, and which must have defined values at all times?*

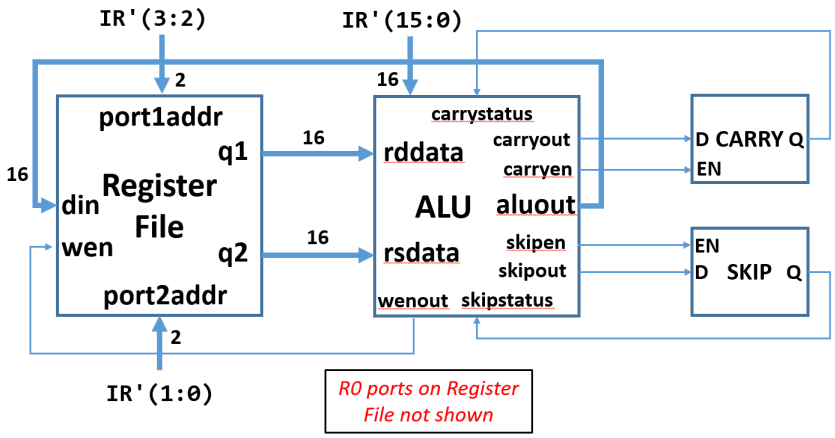

Figure 3: ARMish components connected

□ **Task 7.** *Test your ARMish instructions, with CARRY, as in lecture 11. Check that you can use the new instructions as suggested in TBL Class 6.*

Challenge

There are two ways, to extend this work, it is not likely you will have time for both.
Easy, but requires creativity

- Add one or more new ARMish opcodes using the 4 unused `OP` values. There are lots of options, and implementation can be quick.
- Show a could be useful code fragment that your additions make faster.

More difficult

□ **Task 8.** *Implement in the ALU logic as detailed in Figure 8 that writes `SKIP` to 1 when the next instruction must be skipped, as specified by the `COND` field. `SKIP` should be written only during `EXEC1`, and if `SKIP` is 1 (for an instruction being skipped), it should always be written 0 regardless of `COND`.*

□ **Task 9.** *Add logic (a few `AND` gates) to ensure that when SKIP is high nothing happens:*

- *`wen` and `rOwen` are 0.*
- *`PC sload` is 0*
- *RAM `wren` is 0*
- *`CARRY` cannot change its value.*

□ **Task 10.** *For instructions with $h_2 = 0$[a] `SKIP` will always be 0 and have no effect. Using such instructions, and a test program which implements `R0:R1 := R0:R1 + R2:R3` as suggested in the TBL classes to test your new instructions with `CARRY, CIN` functionality!*

□ **Task 11.** *Optional. Work out instructions (perhaps based on TBL Class questions) to test `COND` and `SKIP`.*

[a] h_2 is the hex digit from $IR' = h_3h_2h_1h_0$

Wen	Port1 addr	Port2 addr	Port1 Out	Port2 Out	op
0	a	b	Ra(15:0)	Rb(15:0)	n/a
1	a	b	Ra(15:0)	Rb(15:0)	Ra := din

Figure 4: Dual Port Register File Operation

A. Dual Port RAM and Register File

The new instructions require 4 registers, which are implemented as dual port RAM with 4 `LPM_FF` blocks, together with logic to read two registers and write one register all in one cycle.

The read and write operations are all independent, except that one of the reads must be from the register that is being written. Hence only two sets of register select (address) lines, and two ports, one of which is read and write. Note that two address lines are needed to select one of four registers. The logic to implement this is shown in Figure 5. Each read operation requires a multiplexer `MUX4` which selects one of the 4 register outputs. The write operation is implemented via a `DEMUX4` block which outputs one of its 4 outputs high, corresponding to the current value of its address inputs, if `wen` is high.

Together this logic implements a *two port RAM* with the logic function shown in Figure 4. Port1 has 2 address inputs that determine the register written to, and the port 1 read output. Port 2 has a separate two address lines that determine a register which can independently be read. The `Wen` line determines whether the register addressed by Port1 is written in each cycle.

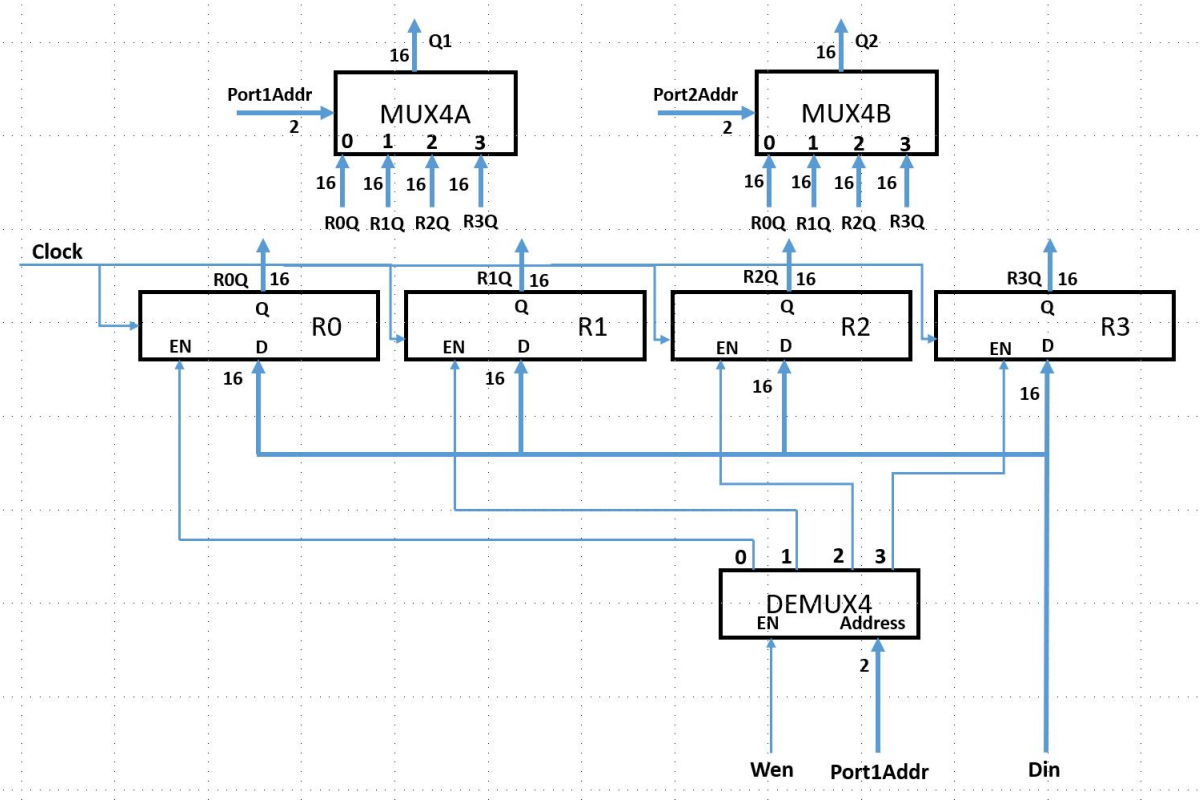

Figure 5: Two Port RAM

B. Answers

- The two port RAM data outputs q1, q2 come from multiplexers with inputs the register q outputs, and select `port1Addr, port2Addr`. Multiplexers are combinational logic therefore there is no delay and the outputs are also available for use in `EXEC1`.

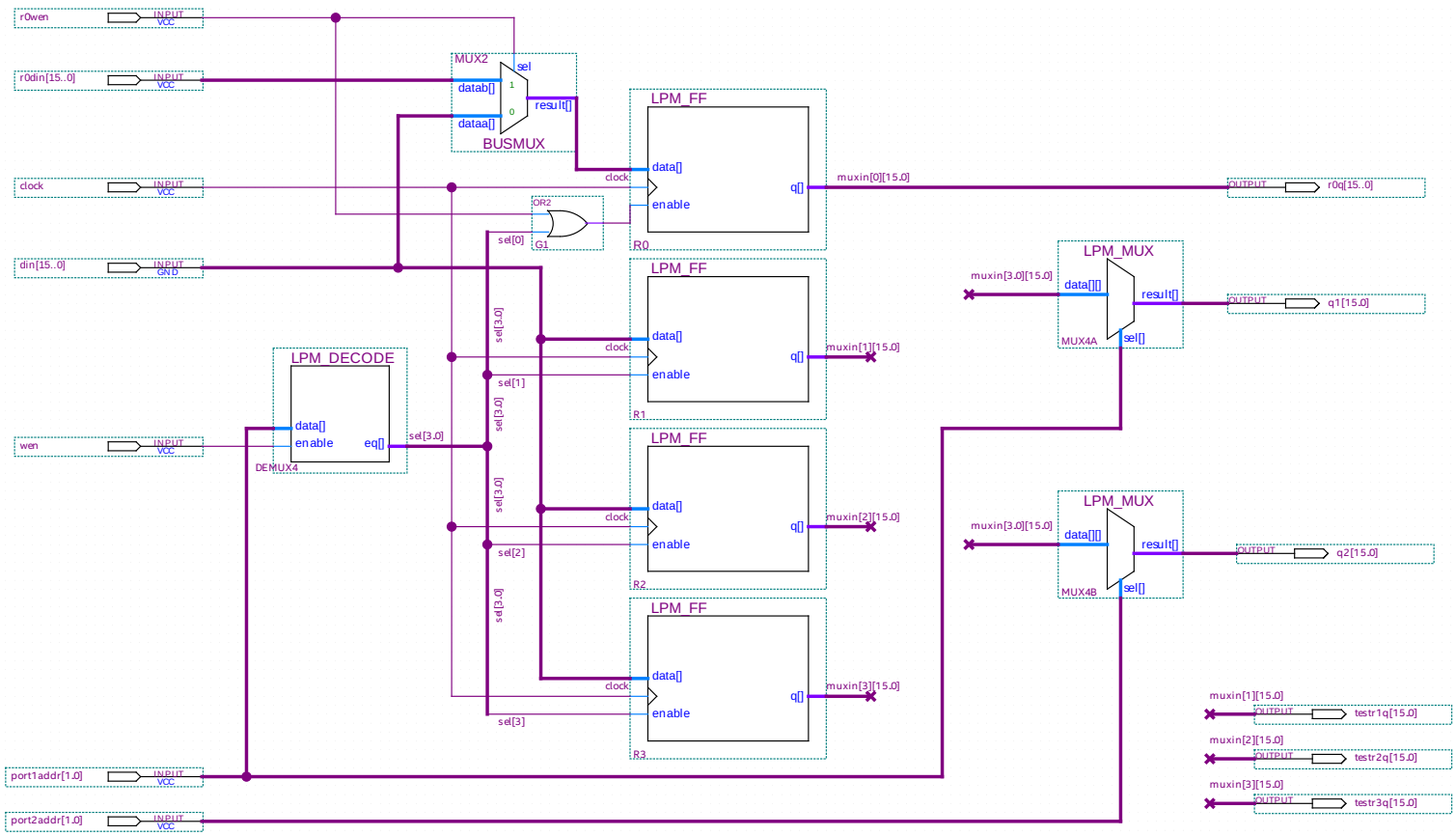

Figure 6: Register File

- Each register is implemented by D FFs so they can be read and written in the same cycle. The newly written data will appear on the outputs in the *next* cycle. This is just what we want in order to implement a two operand instruction with combinational logic in one cycle, with two registers read, and one of them also written!

C. Verilog ALU

This design correctly implements the 4 operations specified in the `OP` field of an ARMish instruction. Another 4 operations are possible. The logic to drive `CARRY` and `SKIP` flip-flops is not complete.

```
module alu ( instruction, rddata, rsdata, carrystatus, skipstatus, exec1,
    aluout, carryout, skipout, carryen, skipen, wenout) ;

// v1.01

input [15:0] instruction; // from IR'
input exec1; // timing signal: when things happen
input [15:0] rddata; // Rd register data outputs
input [15:0] rsdata; // Rs register data outputs
input carrystatus; // the Q output from CARRY
input skipstatus; // the Q output from SKIP

output [15:0] aluout; // the ALU block output, written into Rd
output carryout; // the CARRY out, D for CARRY flip flop
output skipout; // the SKIP output, D for SKIP flip flop
output carryen; // the enable signal for CARRY flip-flop
output skipen; // the enable signal for SKIP flip-flop
output wenout; // the enable for writing Rd in the register file

// these wires are for convenience to make logic easier to see
wire [2:0] opinstr = instruction [6:4]; // OP field from IR'
wire cwinstr = instruction [7]; // 1 => write CARRY: CW from IR'
wire [3:0] condinstr = instruction [11:8]; // COND field from IR'
wire [1:0] cininstr = instruction [13:12]; // CIN field from IR'
wire [1:0] code = instruction [15:14]; // bits from IR': must be 11 for ARM instruction

reg [16:0] alusum; // the 17 bit sum, 1 extra bit so ALU carry out can be extracted
wire cin; // The ALU carry input, determined from instruction as in ISA spec
wire shiftin; // value shifted into bit 15 on LSR, determined as in ISA spec

// do not change
assign alucout = alusum [16]; // carry bit from sum, or shift if OP = 011
assign aluout = alusum [15:0]; // 16 normal bits from sum

// change as needed
assign wenout = exec1; // correct timing, to do: add enable condition
assign carryen = exec1; // correct timing, to do: add enable condition
assign carryout = alucout; // this is correct except for XSR
                        // note the special case of rsdata[0] when OP=011 (XSR)
assign cin = 0; // dummy, to do: replace with correct logic
assign shiftin = 0; // dummy, to do: set equal to cin for correct XSR functionality

assign skipout = 0; // dummy, to do: replace with correct logic
assign skipen = exec1; // correct timing, to do: add enable condition

always @(*) // do not change this line -it makes sure we have combinational logic
  begin
    case (opinstr)
      // alusum is 17 bit so we must extend the two operands to 17 bits using 0
      // otherwise Verilog default extension will sign-extend these inputs
      // that create a subtle (not always obvious) error in carry out
      // note that ~ is bit inversion operator.
      3'b000 : alusum = {1'b0,rddata} + {1'b0,rsdata} + cin; // if OP = 000
      3'b001 : alusum = {1'b0,rddata} + {1'b0,~rsdata} + cin; // if OP = 001
      3'b010 : alusum = {1'b0,rsdata} + cin; // if OP = 010
      3'b011 : alusum = {rsdata[0], shiftin, rsdata[15:1]}; // if OP = 011
      // to do (optional): add additional instructions as cases here
      // available cases: 3'b100,3'b101,3'b110, 3'b111
      default : alusum = 0;// default output for unimplemented OP values, do not change
    endcase;
  end

endmodule
```

D. Two Operand Instruction Format

IR'	Field	Meaning
15:14	11	ARMish Opcode
13:12	CIN	Choose carry in
11:8	COND	Condition to set SKIP
7	S	1 = Write CARRY
6:4	OP	ALU operation
3:2	Rd	Register number
1:0	Rs	Register number

OP		ALU operation
000	ADD	Rd := Rd + Rs + cin
001	SUB	Rd := Rd + $\overline{\text{Rs}}$ + cin
010	MOV	Rd := Rs + cin[a]
011	XSR	Rd := Rs XSR[b] 1

CIN	Name	cin
00	C0	0
01	C1	1
10	CC	CARRY
11	CMSB	Rs(15)

Figure 7: ARMish ISA specification and instruction encoding

[a] The adder carry in is determined from bits 8:7.
[b] XSR shifts right (by 1), with bit 0 written into CARRY if S = 1, and cin shifted into bit 15. This combines ARM LSR, ASR, and RRX functionality.

The required implementation is specified in Figure 7. The new instruction is encoded in previously unused MU0 opcodes 12-15, and therefore has IR'(15:14)=11. The ALU adder carry in cin, and the bit shifted in to bit 15 in an ARMish XSR shift, is specified by the CIN field in the instruction as shown in Figure 7.

D.1. SKIP and COND

This part of the ISA may optionally be implemented.

Figure 8 shows how the COND filed in these instructions optionally implement a *conditional skip* of the next instruction.

Dependent on the current ALU outputs, the next instruction may be skipped or executed. If skipped then the next instruction to be executed will be at PC + 2.

For simplicity, this is implemented using a SKIP flip-flop to store the results of the condition whenever an ARMish instruction is executed. If SKIP is 1 in any instruction all changes to registers or RAM will be inhibited (implementing the skip) and SKIP will be reset to 0.

COND	Name	Meaning
0000	AL	Execute always
0001	NV	Skip always
0010	CS	Execute if adder Cout = 1
0011	CC	Execute if adder Cout = 0

Figure 8: SKIP conditions in ARMish instructions: NB other conditions are unspecified

E. Instruction Fields in Hex

For convenience, these tables summarise the hex digits $h_3h_2h_1h_0$ of the instruction word of an ARMish instruction.

h_3	CIN	meaning
C	C0	cin = 0
D	C1	cin = 1
E	CC	cin = CARRY
F	CMSB	cin = Rs(15)

h_2	COND
0	AL
1	NV
2	CS
3	CC

h_1	S=0	S=1
ADD	0	8
SUB	1	9
MOV	2	A
XSR	3	B

h_0	Rs=0	Rs=1	Rs=2	Rs=3
Rd=0	0	1	2	3
Rd=1	4	5	6	7
Rd=2	8	9	A	B
Rd=3	C	D	E	F

Figure 9: Hex digits $h_3h_2h_1h_0$ of instruction word