CSE 141L Lab 1. 9-Bit Instruction Set Architecture

Winter 2021

In this lab you shall design the instruction set and overall architecture for your own special-purpose (very) reduced instruction set (RISC) processor. You will design the hardware for your processor core in subsequent labs.

Your processor shall have **9-bit instructions** (machine code) and shall be optimized for three simple programs, described below. For this lab, you shall design the instruction set and instruction formats and code three programs to run on your instruction set. Given the tight limit on instruction bits, you need to consider the target programs and their needs carefully. The best design will come from an iterative process of designing an ISA, then coding the programs, redesigning the ISA, etc.

Your instruction set architecture shall feature fixed-length instructions 9 bits wide. Your instruction-set specification should describe:

- what operations it supports and what their respective opcodes are.
  For ideas, see MIPS or ARM instruction lists
- how many instruction formats it supports and what they are (in detail -- how many bits for each field, and where they're found in the instruction). Your instruction format description should be detailed enough that someone could write an assembler (a program that creates machine code from assembly code) for it. (Again, refer to ARM or MIPS.)
- number of registers, and how many general-purpose or specialized. All internal **data** paths and storage will be 8 bits wide.
- addressing modes supported (this applies to both memory instructions and branch instructions). That is, how are addresses constructed or calculated? Lookup tables? Sign extension? Direct addressing? Indirect, as used in linked lists or ARM or MIPs to address the data_memory from reg_file contents?

For this to fit in a 9-bit field, the memory demands of these programs will have to be small. For example, you will have to be clever to support a conventional main memory of 256 bytes (8-bit address pointer). You should consider how much data space you will need before you finalize your instruction format. Your instructions are stored in a separate memory, so that your data addresses need be only big enough to hold data. Your data memory is byte-wide, i.e., loads and stores read and write exactly 8 bits (one byte). Your instruction memory is 9 bits wide, to hold your 9-bit machine code, but it can be as wide as you need to hold all three programs.

You shall write and run three programs on your ISA. You may assume that the first starts at address 0, and the other two are found in memory after the end of the first program (at some nonoverlapping address of your choosing). The specification of your branch instructions may depend on where your programs reside in memory, so you should make sure they still work if the starting address changes a little (e.g., if you have to rewrite one of the programs and it causes the others to also shift). This approach will allow you to put all three programs in the same instruction memory later on in the quarter.

We will impose the following constraints on your design, which will make the design a bit simpler. You shall assume a single-address data memory (Verilog design provided). You shall also assume a register file (or whatever internal storage you support) that can write to only one register per instruction. You may also have a single ALU condition/flag register (e.g., carry out, or shift out, sign result, zero bit, etc., like ARM's Z, N, C, and V status bits) that can be written at the same time as an 8-bit register, if you want. You may read more than one register per cycle. Please restrict register file depth to no more than 16 registers. Also, manual loop unrolling of your code is not allowed.

In addition to these constraints, the following *suggestions* will either improve your performance or greatly simplify your design effort. In optimizing for performance, distinguish between what must be done in series vs. what can be done in parallel. An instruction that does an add and a subtract (but neither depends on the output of the other) takes no longer than a simple add instruction. Similarly, a branch instruction where the branch condition or target depends on a memory operation will make things more difficult later on.

Generic, general-purpose ISAs (that is, those that will execute other programs just as efficiently as those shown here) will be seriously frowned upon. We really want you to optimize for these programs only, as is common practice in consumer embedded systems with tight cost consstraints.

You shall turn in a lab report, plus program listings. The report will answer the questions posed below. In describing your architecture, keep in mind that the person grading it has much less experience with your ISA than you do. It is your responsibility to make everything clear -- one objective of this course is to help you improve your technical writing and reporting skills, which will benefit you richly in your career.

For each lab, there will be a set of requirements and questions that direct the format of the writeup and make it easier to grade, but strive to create a report you can be proud of. Sometimes that may require a little repetition, e.g. describing something where you think it belongs in the report, and then again in the "question" part, so the graders won't miss it.

Additional constraints:

1) Your ALU instructions will be comparable in complexity to those in ARM.

2) Your data memory will have only one address pointer input port, for both input and output.

3) Your register file will have no more than two output ports and one input port. You may use separate pointers for reads and writes, if you wish.

4) You may use lookup tables (LUTs) / decoders, but these are limited to 32 elements each (i.e., address pointer width up to 5 bits). We do not allow something like a 512-element LUT with 32-bit outputs which simply maps your restricted 9-bit machine code field to a 32-bit clone of ARM or MIPS instructions. (It was "cute" the first time a team did this.)

Components of Lab 1 report:

0. Team.
List the names of all members of your team.

1. Introduction.
This should include the name of your architecture, overall philosophy, specific goals strived for and achieved.

2. Instruction formats.
List all formats and an example of each. (ARM has R, I, and B type instructions, for example.)

3. Operations.
List all instructions supported and their opcodes/formats.

4. Internal operands.
How many registers are supported? Is there anything special about any of the registers, or all of them general purpose?

5. Control flow (branches).
What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported?

6. Addressing modes.
What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

Additionally, please explicitly answer the following questions.

7. Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, devise a name for your class of machine.

8. Give an example of an "assembly language" instruction in your machine, then translate it into machine code.

For 9-11, give assembly instructions. Make sure your assembly format is either very obvious or well described, and that the code is well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We shall not correct/grade the machine code. State any assumptions you make. If you need initial nonzero values in registers or memory, you need to put them there. (Exception -- the test bench will load the incoming operands for you.)

9. **Program 1** (encrypter):

A message will be implanted by the test bench into your data memory, in locations [0:53], one ASCII character each):

Example: *Mr. Watson, come here. I want to see you.*

(Spaces and punctuation marks count.) Start with this message, but you should be able to handle <u>any</u> 1- to 54-byte message that uses ASCII characters corresponding to 0x20 through 0x7f.

1. (See the 8-bit ASCII table at the end of this writeup to convert this sequence of characters into numerical bytes. Example: M = 77 decimal = 0x4d)

<u>My Verilog test bench will handle this conversion for you.</u>

2. The number of required initial space characters will be stored in data memory at location [61]. Prepend a preamble of ASCII space characters (space = 32 decimal = 0x20). After encryption, you shall put these into data_memory starting at [64]. Since you will need at least 10 known data preamble values for proper decoding, *you will need to inject 10 space characters if the value in memory[61] is less than 10. Similarly, you shall insert only 26 spaces if memory[61]>26.*

3. Write a 7-bit maximal length linear feedback shift register (LFSR) using the tap sequence number stored in data memory location 62. For example, if the tap for the corresponding stored number is 7543, then:

```
x'[0]   = x[7]^x[5]^x[4]^x[3];
x'[6:1] = x[5:0];
```

where ^ denotes XOR and x[6] is the most significant bit of x and x[0] is the least. x'[6:0] is the "next state," and x[6:0] is the "present state."

See 8-tap LFSR schematic at the end of this writeup to help you visualize what is going on. The logic symbols are XORs. (Our 7-tap works the same way, with one less bit in the shift register.)

There are 9 different LFSR feedback patterns that result in a maximal length sequence of all 127 nonzero states. Your encryptor should be programmable to any randomly selected LFSR feedback pattern. The testbench will load the *index of* the LFSR pattern into data memory location [62] and the starting LFSR state into data_memory location [63]. *If the content of memory[62]>8, your design should use a starting value of memory[62][2:0], i.e., just the 3 LSBs, instead.* You will use only the 7 LSBs of the starting state, and substitute 8'h01 if the starting state is specified as 8'h80 or 8'h00, because all 0s is a disallowed state for LFSRs. (It works, but it just sits there in that state. You might find this useful for testing of all of the other functions of your design.)

4. For a count equal to the value in memory[61], compute the bitwise XOR of ASCII space = 0x20 with each successive value of your LFSR and write the results into data_memory[64:64+(mem[61])][6:0], with the constraint that there will be no fewer than 10 preamble space characters and no more than 24. Thus, if data_memory[61]<10, insert 10 spaces. If data_memory[61]>26, insert only 26 spaces. (Rationale: we need a known 10-character sequence to synchronize our Lab 2 decoder. You can always get around the maximum constraint by sending a message body that starts with space characters.)

5. Now read each value of the message out of memory, starting at location [0], XOR it with the next state of the LFSR, and write the result back into memory locations starting where step 4 left off and continuing to [127]. (*This includes the prepended and appended spaces characters, so just pad the end of the message with encrypted space characters to fill the space.*)

6. Finally (or as you go), set bit [7] of each value in data_mem[64:127] to the reduction XOR (parity) of that location's bits[6:0]. This is precisely the P[0] or P0 augmented parity bit we saw in CSE140L, for

those who took that class with me. These are commonly used in omputer memories to detect single-bit errors.

7. The test bench will write out the resulting message by looking up the stored values in the ASCII table.

Your ISA needs to be able to accomplish the above, starting with some way to bring in data from data_memory[0:63], generate a 7-bit LFSR, bitwise XOR each preamble (ASCII space) and data byte with a different LFSR state, insert parity in each MSB, and store the result back into data_memory[64:127]. See in-class demonstration.

10. **Program 2** (decrypter):

An encrypted message from Program 1 will now occupy data memory locations [64:127]. The MSB of each data word is the parity of the other 7

1. By examining the first 10 bytes of this message, figure out the seed (inital) value for the LFSR and its feedback pattern. You will probably need to perform a correlation to accomplish this. (You, of course, know your own seed, but assume you received someone else's encrypted message, with an unknown seed and pattern. How would you crack the code, given that there are 127 possible initial seeds and 9 maximal length feedback patterns?)

2. Proceed as in Program 1, except that we'll be reading from memory locations [64:127] and writing back to [0:63]. (This decrypted message will start with 10 to 24 ASCII space charcters, followed by the message itself, and ending in ASCII space characters as needed to fill the space.)

3. If you have properly seeded your decrypter's LFSR and can use the same feedback pattern as in the encrypter, you should be able to recover the original message.

Your ISA needs to be able to accomplish everything it can already do for Program 1, but in addition, it needs to be able to search through LFSR states and identify the correct one feedback and initial state. This is the heart of the assignment.

11. **Program 3** (error detection and remove initial space characters)

1. This program shall detect the location of the first non-space character in the message, since there will be leading preamble padding bits of 0x20. (The message itself may also have started with additional space characters. We shall remove these, as well, because we have no way of distinguishing padding preambles from starting spaces within the message body.) It will copy this character into memory location[0], and successive non-zero characters into memory [1, ...]. At the end of the message, it shall pad the remaining memory address values up to [63] with ASCII spaces.

2. The other difference from Program 2 is that a few of the encrypted message characters may have one bad (flipped) bit. (Remember bit [7], our global parity? This is for error detection.) As you load each inoming message character, check its lower 7 bits for consistency with its highest bit ([7]). Half of the 256 possible 8-bit values will be wrong, whereas the others will be correct. If you detect a corrupt character, insert an error flag, 0x80, into the corresponding output character stream.

3. Note that your device shall run all three programs in succession, controlled by a 4-bit interface with the test bench:

clk: digital clock (generated in test bench, input to your device)

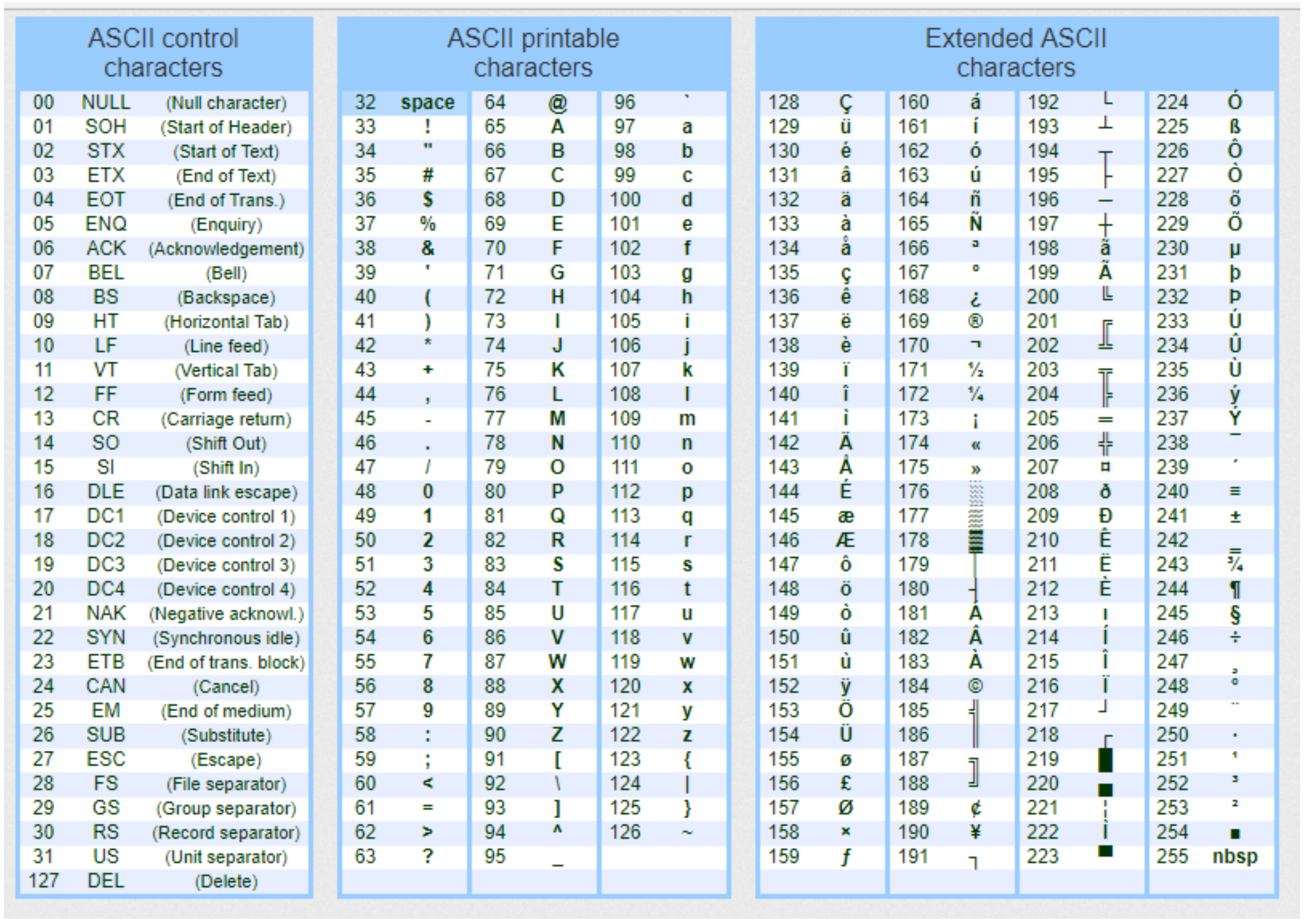
rst: master reset (generated in test bench, input to your device -- in particular, puts your program counter at starting instruction address, most likely 0)

req: request device to start next program (generated in test bench, input to your device)

ack: "program done" flag (output generated by your device, tells test bench "check my work and then ask for the next program")

Note in particular that your device needs to bring the acknowledge / ack signal high right after it completes each program

Appendix 1. ASCII table



ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Appendix 2. 8-tap LFSR schematic -- for feedback pattern 7, 5, 4, 3 (numbering 0 to 7 instead of 1 to 8)

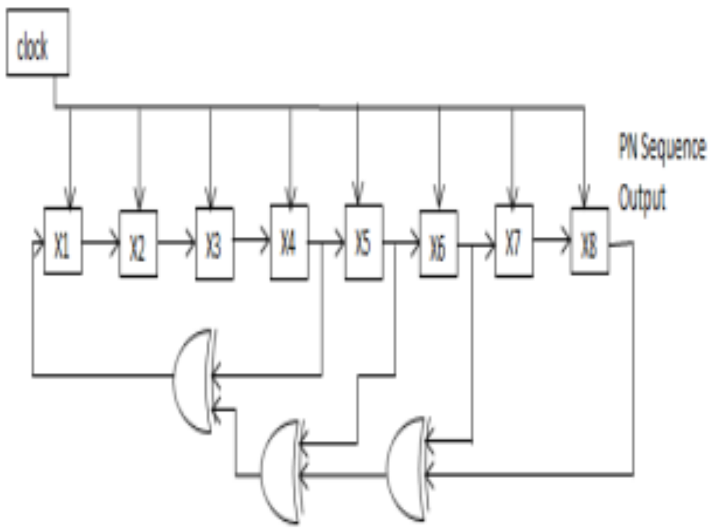


Figure 2: Block diagram of 8 bit LFSR.

This pattern can be represented as 0xb8 (why?). The full list of patterns for an 8-bit LFSR is:
(0x) e1, d4, c6, b8, b4, b2, fa, f3

For a 7-bit, 0-indexed LFSR, the maximal length feedback positions are [6,5] = 0x60, [6,3], [6,5,4,3], [6,5,4,1], [6,5,3,1], [6,5,3,0], [6,4,3,2], [6,5,4,3,2,1], and [6,5,4,3,1,0]

The testbench will randomly assign both a maximal length feedback pattern and a nonzero starting state to your encrypter. Your decrypter then needs to decode the message properly for any given starting state and feedback pattern.

As discussed in class, there are 9 possible feedback tap patterns for a maximal-length (all 127 nonzero states covered) length-7 LFSR with either 2, 4, or 6 active taps, and we can ask you to encrypt using any one of the 9 and to decrypt a message that was encrypted with an (unknown) one of the 9 LFSR patterns.

<u>Lab 1 flow, showing loads and stores:</u>

LFSR:
1) **LOAD** LFSR(0) = dat_mem[63]
2) **LOAD** taps = dat_mem[62]
3) LFSR(i+1) = LFSR_function(LFSR(i))

where LFSR function = (input<<1)|(^(input & taps))

Source data:
1) **LOAD** N = dat_mem[61]
2) *for first N cycles*:
source(i) = 0x20

3) *for remaining cycles, starting w/ i=N:*
source(i) = dat_mem[i-N]

Destination data:
1) dest(i)[6:0] = source(i) ^ LFSR(i)
2) dest(i)[7] = ^dest(i)[6:0]

3) **STORE**: dat_mem[64+i] = dest(i)