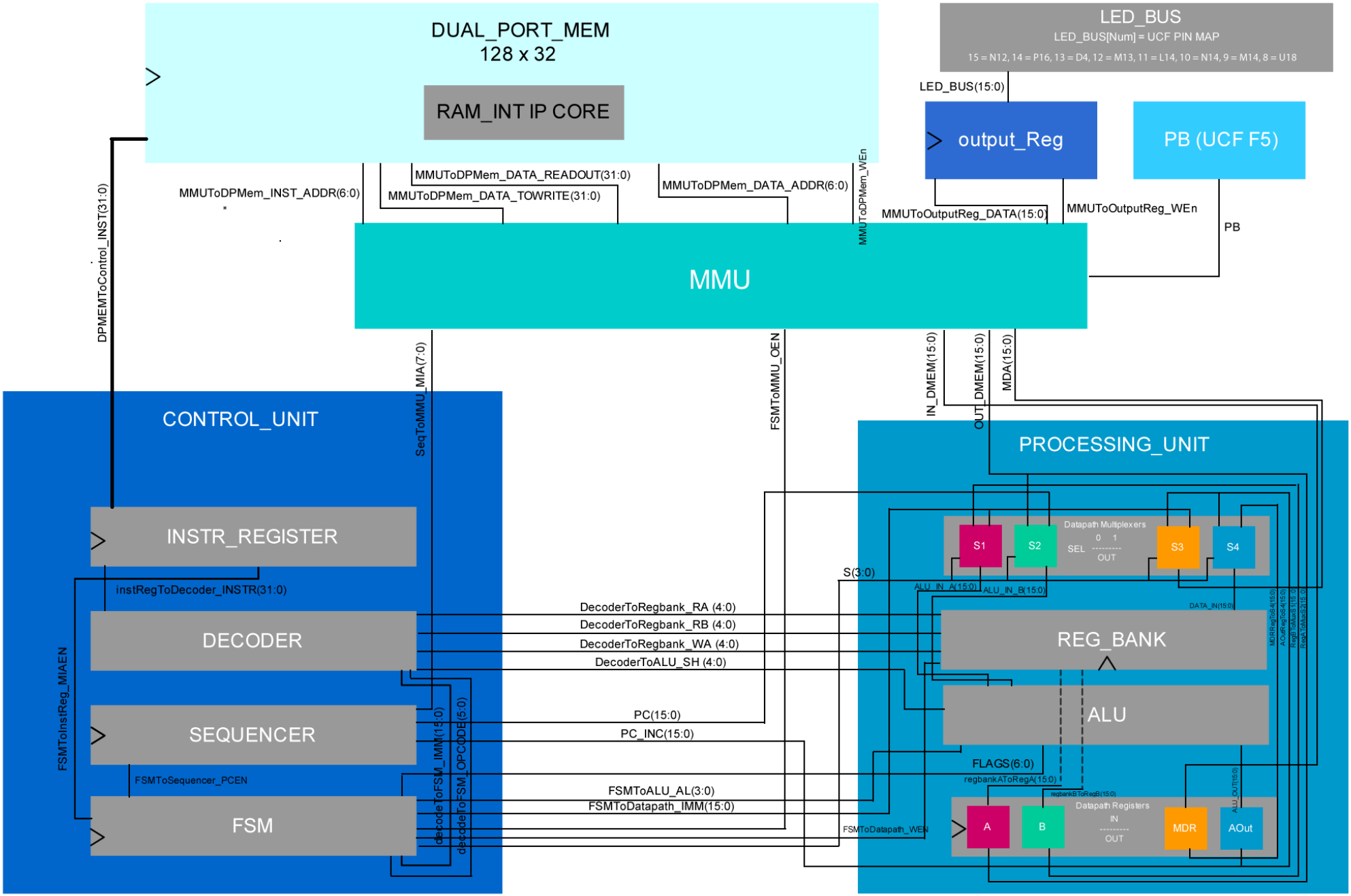
Contents

Multicycle Processor Design 1
CPU Block Diagram 1
Instruction coding and control signals 3
Explanation of OPCODES 4
Explanation of operand positions 5
Multicycle Control Signals 5
Machine language Equivalent 8
State Machine 9
VHDL Code 10
Synthesizable VHDL Code 10
Testbench Code 30
Simulation 31
HDL Synthesis 36
Synthesis Warnings 38
UCF File 38
FPGA Board Upload Output 39

Multicycle Processor Design

CPU Block Diagram

It should be noted that signals passing between units have the correct names, and internal signals within units. However, signals that connect units do not necessarily have the same name once they connect within the unit, as they connect to the input/output ports of that unit and then carry more relevant names. These are not placed on the diagram as this would cause undue clutter. Some signals are dashed, indicating they do not connect to the component they pass through, but there is no room to route it elsewhere.


DUAL_PORT_MEM
128 x 32
RAM_INT IP CORE
LED_BUS
LED_BUS[Num] = UCF PIN MAP
15 = N12, 14 = P16, 13 = D4, 12 = M13, 11 = L14, 10 = N14, 9 = M14, 8 = U18
LED_BUS(15:0)
output_Reg
PB (UCF F5)
MMUToDPMem_INST_ADDR(6:0)
MMUToDPMem_DATA_READOUT(31:0)
MMUToDPMem_DATA_TOWRITE(31:0)
MMUToDPMem_DATA_ADDR(6:0)
MMUToDPMem_WEn
MMUToOutputReg_DATA(15:0)
MMUToOutputReg_WEn
PB
MMU
DPMEMToControl_INST(31:0)
SeqToMMU_MIA(7:0)
FSMToMMU_OEN
IN_DMEM(15:0)
OUT_DMEM(15:0)
MDA(15:0)
CONTROL_UNIT
INSTR_REGISTER
instRegToDecoder_INSTR(31:0)
DECODER
SEQUENCER
FSM
FSMToSequencer_PCEN
FSMToInstReg_MIAEN
decodeToFSM_IMM(15:0)
decodeToFSM_OPCODE(5:0)
PROCESSING_UNIT
Datapath Multiplexers
S1
S2
SEL
OUT
S3
S4
S(3:0)
ALU_IN_A(15:0)
ALU_IN_B(15:0)
DATA_IN(15:0)
REG_BANK
ALU
FLAGS(6:0)
regbankAToRegA(15:0)
regbankBToRegB(15:0)
Datapath Registers
IN
OUT
A
B
MDR
AOut
ALU_OUT(15:0)
FSMToDatapath_WEN
DecoderToRegbank_RA (4:0)
DecoderToRegbank_RB (4:0)
DecoderToRegbank_WA (4:0)
DecoderToALU_SH (4:0)
PC(15:0)
PC_INC(15:0)
FSMToALU_AL(3:0)
FSMToDatapath_IMM(15:0)

Instruction coding and control signals

Instruction	Operation	Instruction Coding																														
		OPCODE					Operands																									
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARITHMETIC																																
nop	None	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
add Rt, Ra, Rb	Rt = Ra + Rb	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Rb					Ra						Rt		
sub Rt, Ra, Rb	Rt = Ra - Rb	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Rb					Ra						Rt		
addi Rt, Ra, Imm	Rt = Ra + Immediate Value	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
subi Rt, Ra, Imm	Rt = Ra - Immediate Value	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
inc Rt, Ra	Rt = Ra + 1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
dec Rt, Ra	Rt = Ra - 1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
LOGIC																																
not Rt, Ra	Rt = NOT Ra	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Ra		
or Rt, Ra, Rb	Rt = Ra OR Rb	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Rb											Ra		
and Rt, Ra, Rb	Rt = Ra AND Rb	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Rb											Ra		
xor Rt, Ra, Rb	Rt = Ra XOR Rb	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	Rb											Ra		
andi Rt, Ra, Imm	Rt = Ra AND Immediate Value	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
ori Rt, Ra, Imm	Rt = Ra OR Immediate Value	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
xori Rt, Ra, Imm	Rt = Ra XOR Immediate Value	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	IMM[15:0]											Ra		
shli Rt, Ra, n	Rt = Ra shifted left by n bits	0	1	1	1	0	0	1	0	n																					Ra		
shri Rt, Ra, n	Rt = Ra shifted right by n bits	0	1	1	1	0	1	1	0	n																					Ra		
roli Rt, Ra, n	Rt = Ra rotated left by n bits	0	1	1	1	0	1	0	1	n																					Ra		
rori Rt, Ra, n	Rt = Ra rotated right by n bits	0	1	1	1	1	1	0	0	n																					Ra		
TRANSFER																																
move Rt, Ra	Rt = Ra	1	0	0	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ra		
load Rt, Imm	Rt = DMEM[Imm] {direct addressing}	1	0	0	0	0	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	IMM[15:0]											Ø	Ø	Ø	Ø	Rt
load Rt, Ra	Rt = DMEM[Ra] {register indirect addressing}	1	0	0	0	0	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ra		
load Rt, Ra, Imm	Rt = DMEM[Ra + Imm] {base plus offset addressing}	1	0	0	0	1	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ra		
store Rb, Imm	DMEM[Imm] = Rb {direct addressing}	1	0	1	0	0	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	IMM[15:0]											Ø	Ø	Ø	Ø	Rb
store Rb, Ra	DMEM[Ra] = Rb {register indirect addressing}	1	0	1	0	0	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ra		
store Rb, Ra, Imm	DMEM[Ra + Imm] = Rb {base plus offset addressing}	1	0	1	1	1	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[9:0]											Ra		
CONTROL																																
jmp Off	Jump to IMEM[PC+off]	1	1	1	0	0	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ø	Ø	Ø	Ø	Ø
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra = 0)	1	1	0	0	0	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra ≠ 0)	1	1	0	0	1	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra = 1)	1	1	0	0	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra < 0)	1	1	0	1	0	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra > 0)	1	1	0	1	0	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra ≤ 0)	1	1	0	1	1	0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		
brca, cond, Off	If (condition is true, then) jump to IMEM[PC+off], else continue. (Condition: Ra ≥ 0)	1	1	0	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	OFF[8:0]											Ra		

Explanation of OPCODES

The reasoning behind the specific OPCODE layout was to allow the isolation of specific instructions of similar types, which in turn allows less “else” statements to be utilised in the decoder state (therefore a simpler, more efficient code base).

Arithmetic Instructions:

- For the arithmetic instructions the first two most significant bits of the OPCODE are always set to “00”.
- If an immediate is required for the instruction the fourth most significant bit will be the value of ‘1’. All other Arithmetic instructions without immediate values will be a ‘0’ at this bit.
- For a decrement or increment instruction the third most significant bit is a value of ‘1’.
- The NOP instruction is all zeroes following the normal convention.
- The last two lowest significant bits are set to differentiate the similar instructions.

Logic Instructions:

- For the Logic instructions the first two most significant bits of the OPCODE are always set to “01”.
- If an immediate is required for the instruction the fourth most significant bit will be the value of ‘1’ following the same principal as in arithmetic. In the case of Logic however the rotate instruction shares this bit, requiring the inclusion of the third most significant bit, in order to differentiate between a shift and a logic immediate instruction.
- All Logic shift instructions are distinguished from other logic by the value of ‘1’ in the third most significant bit of the OPCODE. They all share the “n” bus for number of bits to shift by.
- Rotate and Shift instructions - Shift - are further differentiated by the fourth most significant bit, rotate having a value of ‘1’, shift ‘0’.
- The last two lowest significant bits are set to differentiate the similar instructions.

Transfer Instruction:

- For the Transfer instructions the first two most significant bits of the OPCODE are always set to “10”.
- The move instruction is different enough from other instructions in how it is implemented in the program that the lowest three significant bits are set to “111”. (it functions more like an Arithmetic Instruction in the FSM)
- Store instruction are differentiated from other transfer Instructions via the third most significant bit being the value ‘1’ in the OPCODE.
- For any instruction using an offset in the transfer instruction the fourth and fifth most significant bit are the value ‘1’ (the Move instruction will be excluded from this grouping as it will already be differentiated by the decoder before this point).
- The last two lowest significant bits are set to differentiate the similar instructions.

Control Instructions:

- For the Control instructions the first two most significant bits of the OPCODE are always set to “11”.
- The Jump instruction is separated from the branch instruction via the third most significant bit of the OPCODE being the value ‘1’.
- For the Branch instruction the OPCODE for the three lowest significant bits is arbitrarily defined to increase linearly from “001” to “111” as the condition for the branch is based upon the flag generation from the ALU.

Explanation of operand positions

For the operand positions the main effort was consistency. This was to reduce the amount of combinational logic required to implement the instruction set. As a result, the positions of all operands remain constant aside from “Rb”, which cannot remain in the same position when an “IMM[15:0]” is needed due to overlap. To counter this “Rb” was placed in the usual position of “Rt” as these three operands never coincide in any instruction.

Another design feature of the instruction set is that “IMM” “OFF” and “n” all overlap in the central 16 bits.

Multicycle Control Signals

The step field of the following tables correspond to states in our FSM, matching that which is defined by the standard multicycle FSM:

ARITHMETIC					
	STEP	S[1,2,3,4]	AL	OEN	WEN
nop	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
add rt, ra, rb	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	00ØØ	1010	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
sub rt, ra, rb	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	00ØØ	1011	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
addi rt, ra, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	10ØØ	1010	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
subi rt, ra, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	10ØØ	1011	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
inc rt, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1000	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
dec rt, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1001	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1

LOGIC					
	STEP	S[1,2,3,4]	AL	OEN	WEN
not rt, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	0111	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
or rt, ra, rb	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	00ØØ	0101	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
and rt, ra,rb	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	00ØØ	0100	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
xor rt, ra, rb	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	00ØØ	0110	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
andi rt, ra, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	10ØØ	0100	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
ori rt, ra, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	10ØØ	0101	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
xori rt, ra, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	10ØØ	0110	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
shl rt, ra, n	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1100	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
shr rt, ra, n	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1101	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
rol rt, ra, n	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1110	0	0
	REGWR S3	ØØØØ	ØØØØ	0	1
ror rt, ra, n	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	Ø0ØØ	1111	0	0
	REGWR S3	ØØØØ	ØØØØ	0	1

TRANSFER					
	STEP	S[1,2,3,4]	AL	OEN	WEN
move rt, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S2	0ØØØ	1010	0	0
	REGWR S3	ØØØ0	ØØØØ	0	1
loadi rt, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S6	ØØ0Ø	Ø	0	0
	REGWR S7	ØØØ1	ØØØØ	0	1
loadr rt, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S6	ØØ0Ø	Ø	0	0
	REGWR S7	ØØØ1	ØØØØ	0	1
loado rt, ra, off	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S6	ØØ0Ø	ØØØØ	0	0
	REGWR S7	ØØØ1	ØØØØ	0	1
stori rb, imm	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S5	ØØ0Ø	ØØØØ	1	0
storr rb, ra	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S5	ØØ0Ø	ØØØØ	1	0
storo rb, ra, off	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	ØØØØ	ØØØØ	0	0
	ALU S4	10ØØ	1010	0	0
	MEMRW S5	ØØ0Ø	ØØØØ	1	0

CONTROL					
	STEP	S[1,2,3,4]	AL	OEN	WEN
jmp off	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	11ØØ	1010	0	0
	ALU S8	ØØØØ	ØØØØ	0	0
brc ra, cond, off	FETCH S0	Ø1ØØ	1000	0	0
	REGRD S1	11ØØ	1010	0	0
	ALU S8	ØØØØ	ØØØØ	0	0

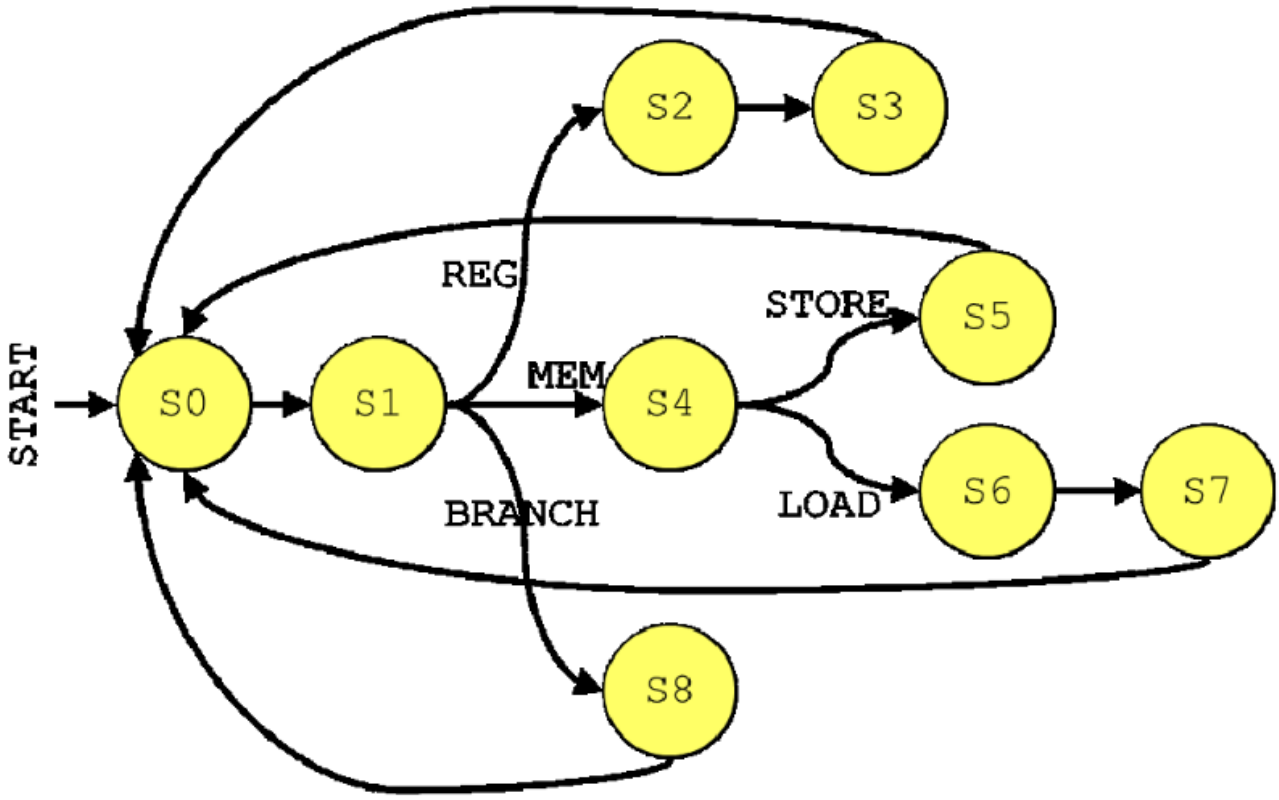
Machine language Equivalent

Machine language is coded as binary and then converted to hexadecimal for input into '.coe' file, using an excel spreadsheet:

Instruction	Binary Instruction Coding	HEX Equivalent
nop	00000000000000000000000000000000	00000000
loadi r15, h01F0	10000000000001111100000000001111	8007C00F
br r15,=0,-1	11000100000001111111100111100000	C407F9E0
addi r31, r0, h001F	00010100000000000111111111111111	14007FFF
add r1, r0, r0	00000100000000000000000000000001	04000001
inc r1, r1	00100100000000000000000000100001	24000021
br r31, <0, +5 (L2)	11010000000000000010111111100000	D00017E0
storr r1, r31	10100100000000000000001111100001	A40003E1
inc r1, r1	00100100000000000000000000100001	24000021
dec r31, r31	00101000000000000000001111111111	280003FF
jmp -4 (L1)	11100000000001111111000000000000	E007F000
loadi r2, h0010	10000000000000000100000000000010	80004002
ori r30, r0, h0004	01011000000000000001000000011110	5800101E
loadr r3, r30	10000100000000000000001111000011	840003C3
loado r4, r30, 3	10011000000000000000111111000100	98000FC4
xori r29, r0, hFFFF	01011111111111111111110000011101	5FFFFC1D
move r7, r0	10011100000000000000000000000111	9C000007
andi r5, r29, h0010	01010100000000000100001110100101	540043A5
andi r6, r4, h0001	01010100000000000000010010000110	54000486
br r6, =0, +2 (L4)	11000100000000000000100011000000	C40008C0
add r7, r3, r7	00000100000001110000000001100111	04070067
shr r4, r4, 1	01101000010000000000000010000100	68400084
shl r3, r3, 1	01100100010000000000000001100011	64400063
dec r5, r5	00101000000000000000000010100101	280000A5
br r5, ≠0, -6 (L3)	11001000000001111111010001010000	C807E8A0
stori r7, 0	10100000000000000000000000000111	A0000007
rol r7, r7, 9	01110110010000000000000011100111	764000E7
ror r7, r7, 3	01111000110000000000000011100111	78C000E7
not r8, r7	01000000000000000000000011101000	400000E8
xor r8, r29, r8	01001100000010000000001110101000	4C0803A8
sub r9, r8, r7	00001000000001110000000100001001	08070109
subi r10, r9, h0001	00011000000000000000010100101010	1800052A
br r10, >0, +9 (Lx)	11010100000000000010010101000000	D4002540
br r10, ≥0, +8 (Lx)	11011100000000000010000101000000	DC002140
addi r11, r10, h0002	00010100000000000000100101001011	1400094B
br r11, ≤0, +6 (Lx)	11011000000000000011100101100000	D8001960
or r12, r7, r11	01000100000010110000000011101100	440B00EC
storo r12, r0, 1	10111000000000000000010000001100	B800040C
and r12, r12, r11	01001000000010110000000110001100	480B018C
br r12, =1, +3 (Lok)	11001100000000000000110110000000	CC000D80
jmp +0	11100000000000000000000000000000	E0000000
jmp +0	11100000000000000000000000000000	E0000000
stori r7,h01F8	10100000000001111110000000000111	A007E007
jmp +0	11100000000000000000000000000000	E0000000

State Machine

The original state machine was used with no modifications.



VHDL Code

Synthesizable VHDL Code

During development of the processor, U's were used for don't care values rather than '0's , in order to better flag design issues in the processor on a testbench. For the synthesizable version that was placed onto the FPGA, these were all replaced with '0's, keeping latches out of the system.

Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
USE ieee.std_logic_unsigned.all;
use work.DigEng.ALL;

entity top_level is
     generic (dataSize : natural := 16;
                numRegisters : natural := 32;
                numRegistersBase : natural := 5;
                busSize : natural := 4);
    Port ( clk: in  STD_LOGIC;
           rst: in  STD_LOGIC;
           LED_BUS : out STD_LOGIC_VECTOR(15 downto 0);
           PB : in STD_LOGIC);
end top_level;

architecture Behavioral of top_level is
    ------- CONTROL UNIT TO PROCESSING UNIT -----------
    signal DecoderToRegbank_RA : STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
    signal DecoderToRegbank_RB : STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
    signal DecoderToRegbank_WA : STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);

    signal FSMToRegBank_WEN : STD_LOGIC;
    signal FSMToDatapath_IMM : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal FSMToDatapath_WEN : STD_LOGIC;
    signal FSMToDatapath_S : STD_LOGIC_VECTOR (3 downto 0);

    signal FSMToALU_AL : STD_LOGIC_VECTOR (3 downto 0);
    signal DecoderToALU_SH : STD_LOGIC_VECTOR (busSize-1 downto 0);

    signal PC : STD_LOGIC_VECTOR (dataSize-1 downto 0);

    ---------- PROCESSING UNIT TO CONTROL UNIT -----------------
    signal PC_INC : STD_LOGIC_VECTOR (dataSize-1 downto 0);
    signal FLAGS : STD_LOGIC_VECTOR (6 downto 0);

    ---------- CONTROL UNIT TO MMU -------------------
    signal SeqToMMU_MIA : STD_LOGIC_VECTOR(7 downto 0);
    signal FSMToMMU_OEN : STD_LOGIC;

    ------------ PROCESSING UNIT TO MMU -----------------
    signal IN_DMEM : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal OUT_DMEM : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal MDA : STD_LOGIC_VECTOR(dataSize-1 downto 0);

    -------------- MMU TO DPMEM ------------
    signal MMUToDPMem_INST_ADDR : STD_LOGIC_VECTOR(6 downto 0);
    signal MMUToDPMem_DATA_TOWRITE : STD_LOGIC_VECTOR(31 downto 0);
    signal MMUToDPMem_DATA_READOUT : STD_LOGIC_VECTOR(31 downto 0);
    signal MMUToDPMem_DATA_ADDR : STD_LOGIC_VECTOR(6 downto 0);
    signal MMUToDPMem_WEn : STD_LOGIC;

    ---- MMU TO MMIO -----
    signal MMUToOutputReg_DATA : STD_LOGIC_VECTOR(15 downto 0);
    signal MMUToOutputReg_WEn : STD_LOGIC;

    ---- DPMEM TO CONTROL -----
    signal DPMEMToControl_INSTR : STD_LOGIC_VECTOR(31 downto 0);
```

```
begin

-------- INSTANTIATION OF ENTITIES REQUIRED FOR PROCESSOR --------

-- Instantiation of processing unit containing ALU and register bank
-- along with S muxes.
PROCESSING_UNIT: entity work.PROCESSING_UNIT
    generic map(numRegisters => numRegisters,
                numRegistersBase => numRegistersBase,
                dataSize => dataSize,
                shiftSize => busSize)
    port map(clk=> clk,
             rst=> rst,
             RA => DecoderToRegbank_RA,
             RB => DecoderToRegbank_RB,
             WA => DecoderToRegbank_WA,
             IMM => FSMToDatapath_IMM,
             WEN => FSMToDatapath_WEN,
             S => FSMToDatapath_S,
             AL => FSMToALU_AL,
             SH => DecoderToALU_SH,
             FLAGS => FLAGS,
             PC => PC,
             PC_INC=> PC_INC,
             IN_DMEM => IN_DMEM,
             OUT_DMEM => OUT_DMEM,
             MDA => MDA);

-- Instantiation of control logic (including FSM, instruction register, instruction decoder and sequencer)
CONTROL_UNIT: entity work.CONTROL_UNIT
    generic map(numRegisters => numRegisters,
                numRegistersBase => numRegistersBase,
                dataSize => dataSize,
                busSize => busSize)
    port map(clk  => clk,
             rst  => rst,
             INSTRUCTION => DPMEMToControl_INSTR,
             MIA  => SeqToMMU_MIA,
             RA => DecoderToRegbank_RA,
             RB => DecoderToRegbank_RB,
             WA => DecoderToRegbank_WA,
             IMM => FSMToDatapath_IMM,
             OEN => FSMToMMU_OEN,
             WEN => FSMToDatapath_WEN,
             S => FSMToDatapath_S,
             AL => FSMToALU_AL,
             SH => DecoderToALU_SH,
             FLAGS => FLAGS,
             PC => PC,
             PC_INC  => PC_INC);

-- Instantiation of IP core memory wrapper for DUAL PORT MEMORY
DUAL_PORT_MEM: entity work.DP_MEM
    port map(clk => clk,
             INST_ADDRESS => MMUToDPMem_INST_ADDR,
             DATA_ADDRESS => MMUToDPMem_DATA_ADDR,
             DATA_In => MMUToDPMem_DATA_TOWRITE,
             WEn => MMUToDPMem_WEn,
             Data_Out  => MMUToDPMem_DATA_READOUT,
             INSTR_DATA => DPMEMToControl_INSTR);

-- Instantiation of MMU to write correctly into 128x32 distributed ip core
MMU: entity work.MMU
    port map(MIA  => SeqToMMU_MIA,
             DMEM_RW_ADDRESS => MDA,
             DMEM_OUT_TOPROC => IN_DMEM,
             DMEM_IN_FROMPROC => OUT_DMEM,
             OEn => FSMToMMU_OEN,

             INST_ADDRESS => MMUToDPMem_INST_ADDR ,

             DMEM_DATA_TOWRITE => MMUToDPMem_DATA_TOWRITE,
             DMEM_DATA_READOUT => MMUToDPMem_DATA_READOUT,

             DATA_ADDRESS => MMUToDPMem_DATA_ADDR,
             DMEM_WEn => MMUToDPMem_WEn ,
```

```
                MMIO_DATA  => MMUToOutputReg_DATA,
                MMIO_WEn => MMUToOutputReg_WEn,
                PB_IN => PB);

-- Instantiation of output register connected at memory address h01F8
output_Reg: entity work.register_block
        generic map(dataSize => 16)
        port map(D => MMUToOutputReg_DATA,
                    Q => LED_BUS,
                    clk => clk, rst=> rst,
                    En => MMUToOutputReg_WEn);

end Behavioral;
```

Processing Unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

---------------------------------- PROCESSING UNIT ---------------------------------

-- The processing unit connects the ALU to the internal processor register bank.
-- The ALU is connected to multiplexers to pipe data appropriately based upon the
-- instruction that is decoded. Similarly, the ALU output is multiplexed to Data
-- Memory to allow for memory address calculation using the ALU. Stages of the
-- processing unit are separated by registers to enable stages to be bypassed,
-- storing intermediate results. Output from ALU is multiplexed with output from
-- DATA memory, allowing selection between alu and dmem value for store to processor
-- registers based upon opcode.

-----------------------------------------------------------------------------------
entity PROCESSING_UNIT is
     generic (dataSize : natural := 16; -- Size of data in processor
                 numRegisters : natural := 32; -- Number of Registers
                 numRegistersBase : natural := 5; -- Log2 number of registers
                 shiftSize : natural := 4); -- Log2 size of data to give size of bus required to shift that
data size max amount
    Port ( clk : in STD_LOGIC;
              rst : in STD_LOGIC;
              -------- DECODED CONTROL SIGNALS --------
              -- These signals are decoded from the instruction/
              -- generated by the FSM.
              RA    : in STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
           RB    : in STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
           WA    : in STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
           IMM  : in STD_LOGIC_VECTOR(dataSize -1 downto 0);
           WEN  : in STD_LOGIC;
           S    : in STD_LOGIC_VECTOR (3 downto 0);
           AL : in STD_LOGIC_VECTOR (3 downto 0);
           SH : in STD_LOGIC_VECTOR (shiftSize-1 downto 0);
           FLAGS : out  STD_LOGIC_VECTOR (6 downto 0);

              ------ PROGRAM COUNTER SIGNALS -------
              -- PC signal from sequencer gets modified by ALU
              -- on fetch/branch by generating a PC inc signal of
              -- correct value.
           PC : in  STD_LOGIC_VECTOR (15 downto 0);
           PC_INC : out  STD_LOGIC_VECTOR (15 downto 0);

              ------- DATA Memory -----------
              -- Connection of stages of processing unit
              -- to data memory
              IN_DMEM : in STD_LOGIC_VECTOR(dataSize-1 downto 0);
              OUT_DMEM : out STD_LOGIC_VECTOR(dataSize-1 downto 0);
              MDA : out STD_LOGIC_VECTOR(dataSize-1 downto 0));

end PROCESSING_UNIT;

architecture Behavioral of PROCESSING_UNIT is
    --- Connection of registers A and B to ALU stage registers
    signal regbankAToRegA : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal regbankBToRegB : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    -- Connection of ALU stage registers to ALU input Multiplexers S1, S2
    signal regAtoMuxS2 : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal regBtoMuxS1 : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    -- Connection of S1 S2 multiplexer outputs to ALU inputs
    signal alu_in_A : STD_LOGIC_VECTOR (dataSize-1 downto 0);
    signal alu_in_B : STD_LOGIC_VECTOR (dataSize-1 downto 0);
    -- Connection of ALU output (after shifter) to AOut register
    signal alu_out : STD_LOGIC_VECTOR (dataSize-1 downto 0);

    -- Connection of S4 mux to data input of Register bank
    signal data_in : STD_LOGIC_VECTOR (dataSize-1 downto 0);

    -- Connection of registers to S4 for register write multiplexer
    signal AOutRegToS4 : STD_LOGIC_VECTOR(dataSize-1 downto 0);
    signal MDRRegToS4 : STD_LOGIC_VECTOR(dataSize-1 downto 0);
```

```
    -- Internal signals used to control multiplexers
    -- set to values from S bus input
    signal S1 : STD_LOGIC;
   signal S2 : STD_LOGIC;
   signal S3 : STD_LOGIC;
   signal S4 : STD_LOGIC;

begin

---------------------------- DATAPATH ----------------------------

-- S Bus values present in reverse order as bus is used to carry S values
-- We therefore set STD LOGIC signals (with more appropriate names to control
-- the multiplexers) to the correct values from S Bus input to proc unit
    S1 <= S(3);
    S2 <= S(2);
    S3 <= S(1);
    S4 <= S(0);

-- Register A to store values read in from Regbank A input for input
-- to ALU after S2 mux.
reg_A: entity work.register_block
        generic map(dataSize => dataSize)
        port map(D => RegbankAToRegA,
                    Q => regAtoMuxS2,
                    clk => clk,
                    rst=> rst,
                    En => '1');

-- Register B to store values read in from Regbank B input for input
-- to ALU after S1 mux.
reg_B: entity work.register_block
        generic map(dataSize => dataSize)
        port map(D => RegbankBToRegB,
                    Q => regBtoMuxS1,
                    clk => clk,
                    rst=> rst,
                    En => '1');

-- Register AOut to store values read in from ALU output for input
-- to Registerbank after S4 mux.
reg_Aout: entity work.register_block
        generic map(dataSize => dataSize)
        port map(Q => AoutRegToS4,
                    D => ALU_OUT,
                    clk => clk,
                    rst=> rst,
                    En => '1');

-- Register MDR to store values read in from DMEM output for input
-- to Registerbank after S4 mux.
reg_MDR: entity work.register_block
        generic map(dataSize => dataSize)
        port map(Q => MDRRegToS4,
                    D => IN_DMEM,
                    clk => clk,
                    rst=> rst,
                    En => '1');

-- Connection of data memory data input to value from datapath register B.
-- (After register, before S1 mux)
OUT_DMEM <= regBtoMuxS1;

-- Connection of PC INC line to AOut register output (After AOut register, before S4 mux)
PC_INC <= AoutRegToS4;

-- Connection of ALU input B to either IMM line or value stored on register B dependent upon
-- S1 control signal (generated by FSM)
ALU_IN_B <=
            IMM when S1 = '1' else
            RegBtoMuxS1 when S1 = '0' else
            (others => 'U'); -- Catch-all clause of U to signify error

-- Connection of ALU input A to either PC line or value stored on register A dependent upon
-- S2 control signal (generated by FSM)
ALU_IN_A <=
```

```
            PC when S2 = '1' else
            RegAtoMuxS2 when S2 = '0' else
            (others => 'U');

-- Connection of DMEM memory data address to either IMM or calculated value from ALU output
-- register. IMM line never used as FSM loads IMM using S1 mux through ALU, costing an extra clock
-- cycle.
MDA <=
            IMM when S3 = '1' else
            AoutRegToS4 when S3 = '0' else
            (others => 'U');

-- Connection of Register bank data input to either data from memory or calculated using ALU
-- dependent upon S4 control signal (generated by FSM)
DATA_IN <=
            MDRRegToS4 when S4 = '1' else
            AoutRegToS4 when S4 = '0' else
            (others => 'U');

-------------------------------------------------------------------

-- Instantiation of ALU and Register bank, with connections to MUX inputs/outputs

ALU: entity work.ALU
    generic map(dataSize => dataSize)
    port map(A => alu_in_A,
             B => alu_in_B,
             X => SH,
             OPCODE => AL,
             ALU_OUT => alu_out,
             FLAGS => FLAGS);

REG_BANK: entity work.register_bank
    generic map(M => numRegisters, baseM => numRegistersBase, dataSize => dataSize)
    port map(DATA_IN => DATA_IN,
             DATA_OUT_RA => RegbankAToRegA,
             DATA_OUT_RB => RegbankBToRegB,
             RA => RA,
             RB => RB,
             WA => WA,
             rst => rst,
             clk => clk,
             WEn => WEN);

end Behavioral;
```

ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_signed.all;
use work.DigEng.ALL;

-- Parameterizable (by Number Size) Arithmetic and Logic Unit
-- Supported operations: Addition, Subtraction, Bitwise logic, Shift/Rotate
entity ALU is
    generic (dataSize : natural);
    Port ( A : in  STD_LOGIC_VECTOR (dataSize-1 downto 0); -- Input number A to be manipulated
           B : in  STD_LOGIC_VECTOR (dataSize-1 downto 0); -- Input number B for use in addition/subtraction
           X : in  STD_LOGIC_VECTOR (log2(dataSize)-1 downto 0); -- Number of bits to shift/rotate A by
              OPCODE : in  STD_LOGIC_VECTOR (3 downto 0); -- Specifies operation for ALU to perform
           ALU_OUT : out  STD_LOGIC_VECTOR (dataSize-1 downto 0); -- Result of ALU operation
           FLAGS : out  STD_LOGIC_VECTOR (6 downto 0) -- Flag bus to provide metadata for ALU result
           );
end ALU;

architecture Behavioral of ALU is
-- Internal signal for later use in flag generation
-- as output bus ALU OUT cannot be read from
    signal INT_OUT : SIGNED (dataSize-1 downto 0);
begin

INT_OUT <=
                signed(A) when OPCODE = "0000" else
                -- Bitwise operations (cast A and B to signed for
                -- direct allocation to signed bus)
                signed(A AND B) when OPCODE  = "0100" else
                signed(A OR B) when OPCODE = "0101" else
                signed(A XOR B) when OPCODE = "0110" else
                signed(NOT (A)) when OPCODE = "0111" else
                -- Arithmetic operations
                signed(A) + 1 when OPCODE = "1000" else
                signed(A) - 1 when OPCODE = "1001" else
                signed(A + B) when OPCODE = "1010" else
                signed(A - B) when OPCODE = "1011" else
                -- Shift/Rotate operations
                shift_left(signed(A), to_integer(unsigned(X))) when OPCODE = "1100" else
                shift_right(signed(A), to_integer(unsigned(X))) when OPCODE = "1101" else
                rotate_left(signed(A), to_integer(unsigned(X))) when OPCODE = "1110" else
                rotate_right(signed(A), to_integer(unsigned(X))) when OPCODE = "1111" else
                signed(A);

-- Flag generation dependent upon value of ALU result
FLAGS(0) <=
                '1' when INT_OUT = 0 else
                '0';
FLAGS(1) <=
                '1' when INT_OUT /= 0 else
                '0';
FLAGS(2) <=
                '1' when INT_OUT = 1 else
                '0';
FLAGS(3) <=
                '1' when INT_OUT < 0 else
                '0';
FLAGS(4) <=
                '1' when INT_OUT > 0 else
                '0';
FLAGS(5) <=
                '1' when INT_OUT <= 0 else
                '0';
FLAGS(6) <=
                '1' when INT_OUT >= 0 else
                '0';


---- Overflow bit generation, not required for CPU
--FLAGS(7) <=
```

```
--              '1' when OPCODE = "1000" AND (A(dataSize-1) /= INT_OUT(dataSize-1)) else -- Addition by 1:
Overflow if MSB has changed from A to output
--              '1' when OPCODE = "1001" AND (A(dataSize-1) /= INT OUT(dataSize-1)) else -- Subtraction by 1:
Overflow if MSB has changed from A to output
--              '1' when OPCODE = "1010" AND ((A(dataSize-1) AND (B(dataSize-1))) /= INT_OUT(dataSize-1))
else -- Addition of B to A: overflow if A and B MSB's are
--
-- 1 and MSB in output has changed
--              '1' when OPCODE = "1011" AND ((A(dataSize-1) AND (B(dataSize-1))) /= INT OUT(dataSize-1))
else -- Subtraction of B from A: overflow if A and B MSB's are
--
-- 1 and MSB in output has changed
--              '0';

-- Assign internal bus to module external output
ALU_OUT <= std_logic_vector(INT_OUT);

end Behavioral;
```

Parameterizable Dual Port Register Bank

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_unsigned.all;
use work.DigEng.ALL;

---------------------------------- REGISTER BANK- ---------------------------------

-- Parameterizable (data Size, num registers) dual port read reigster bank, single write.
-- Able to store M-1 different numbers (register 0 tied to ground), each of datasize

--------------------------------------------------------------------------------------
entity register_bank is
     generic (M : natural;
                baseM : natural;
                dataSize : natural
               );
    Port ( DATA_IN : in  std_logic_vector(dataSize-1 downto 0); -- Data to write to register selected by WA
           DATA_OUT_RA : out  std_logic_vector(dataSize-1 downto 0); -- Output of port A
              DATA_OUT_RB : out  std_logic_vector(dataSize-1 downto 0); -- Output of port B
              RA : in std_logic_vector (baseM-1 downto 0); -- Address of register to read onto DATA_OUT_RA
              RB : in std_logic_vector (baseM-1 downto 0); -- Address of register to read onto DATA_OUT_RB
              WA : in std_logic_vector (baseM-1 downto 0); -- Address to write DATA_IN to
           rst : in  STD_LOGIC; -- Reset values of all registers (exluding 0)
           clk : in  STD_LOGIC;
           WEn : in  STD_LOGIC); -- Write enable for DATA_IN onto register selected by WA
end register_bank;

architecture Behavioral of register_bank is
    -- Decoder for RA port. Selects register to read from using Tri state buffers
    signal decoder1ToTriArray1En : std_logic_vector(M-1 downto 0);
    -- Decoder for RB port. Selects register to read from using Tri state buffers
    signal decoder2ToTriArray2En : std_logic_vector(M-1 downto 0);
    -- Decoder for WA. Selects register to write to using Tri state buffers
    signal decoderInToRegArrayEn : std_logic_vector(M-1 downto 0);
    -- Large bus containing every bit of every register
    signal registerToTriArrays   : std_logic_vector((M*dataSize)-1 downto 0);
begin

-- Tie register 0 to ground
registerToTriArrays(dataSize-1 downto 0) <= (others => '0');

-- Generate registers up to M, with tri state buffers for both ports
m_bit_register : for i in 0 to M-1 generate
    reg_array : if i > 0 generate
        registers: entity work.register_block
            generic map(dataSize => dataSize)
            port map(Q => registerToTriArrays((dataSize*(i+1))-1 downto (i * dataSize)), D => DATA_IN, clk => clk, rst=>rst,
En => decoderInToRegArrayEn(i));
    end generate;

    tri_state_array1 : entity work.tri_state_buffer
        generic map(dataSize => dataSize)
        port map(En => decoder1ToTriArray1En(i), DATA_IN => registerToTriArrays((dataSize*(i+1))-1 downto (i * dataSize)),
DATA_OUT => DATA_OUT_RA);

    tri_state_array2 : entity work.tri_state_buffer
        generic map(dataSize => dataSize)
        port map(En => decoder2ToTriArray2En(i), DATA_IN => registerToTriArrays((dataSize*(i+1))-1 downto (i * dataSize)) ,
DATA_OUT => DATA_OUT_RB);
end generate;

-- Instantiate decoders required for reading and writing
decoder1 : entity work.decoder
    generic map(M => M)
    port map(data_in => RA, DATA_OUT => decoder1ToTriArray1En, En => '1');

decoder2 : entity work.decoder
    generic map(M => M)
    port map(data_in => RB, DATA_OUT => decoder2ToTriArray2En, En => '1');

decoderIn : entity work.decoder
    generic map(M => M)
    port map(data_in => WA, DATA_OUT => decoderInToRegArrayEn, En => WEn);

end Behavioral;
```

Parameterizable Register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--------- PARAMETERIZABLE REGISTER ----------
entity register_block is
     generic (dataSize : natural);
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           D : in  STD_LOGIC_VECTOR (dataSize-1 downto 0);
           Q : out  STD_LOGIC_VECTOR (dataSize-1 downto 0);
           En : in  STD_LOGIC);
end register_block;

architecture Behavioral of register_block is

begin

m_bit_register : for i in 0 to dataSize-1 generate
    flip_flop: entity work.D_FF
        port map(Q => Q(i), D => D(i), clk => clk, rst=>rst, En => En);
end generate;

end Behavioral;
```

Control Unit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

---------------------------------- CONTROL UNIT ----------------------------------

-- The control unit contains the sequencer, decode logic, FSM and instruction register.
-- All of the control signals for the processor are generated here based upon the instruction
-- read into the instruction register.

--------------------------------------------------------------------------------

entity CONTROL_UNIT is
    generic (dataSize : natural := 16; -- Size of data in processor
                numRegisters : natural := 32; -- Number of Registers
                numRegistersBase : natural := 5; -- Log2 number of registers
                busSize : natural := 4); -- Log2 size of data to give size of bus address
    Port ( clk : in STD_LOGIC;
              rst : in STD_LOGIC;

              INSTRUCTION : in  STD_LOGIC_VECTOR (31 downto 0);
           MIA : out  STD_LOGIC_VECTOR (7 downto 0);
              -------- CONTROL SIGNALS ------------
           RA    : out STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
           RB    : out STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);
           WA    : out STD_LOGIC_VECTOR (numRegistersBase-1 downto 0);

           IMM  : out STD_LOGIC_VECTOR(dataSize -1 downto 0);
           OEN  : out STD_LOGIC;
           WEN  : out STD_LOGIC;
           S    : out STD_LOGIC_VECTOR (3 downto 0);
           AL : out STD_LOGIC_VECTOR (3 downto 0);
           SH : out STD_LOGIC_VECTOR (busSize-1 downto 0);
           FLAGS : in  STD_LOGIC_VECTOR (6 downto 0);
              ------- PC ----------------
           PC : out  STD_LOGIC_VECTOR (15 downto 0);
           PC_INC : in  STD_LOGIC_VECTOR (15 downto 0));
end CONTROL_UNIT;

architecture Behavioral of CONTROL_UNIT is
    -- Signals from decoder to FSM, piped into FSM in case FSM needs to generate values
    -- dependent upon them
    signal decodeToFSM_OPCODE : STD_LOGIC_VECTOR (5 downto 0);
    signal decodeToFSM_IMM : STD_LOGIC_VECTOR(dataSize -1 downto 0);

    -- Connection of FSM enable signals to sequencer for PC related busses
    signal FSMToInstReg_MIAEN : STD_LOGIC;
    signal FSMToSequencer_PCEN : STD_LOGIC;

    -- Connection of instruction register output to decoder CLB
    signal instRegToDecoder_INSTR : STD_LOGIC_VECTOR(31 downto 0);
begin

-- Register to hold current instruction. Input loaded to output when MIA enable
-- comes through from FSM in fetch state of instruction.
INSTR_REGISTER: entity work.register_block
        generic map(dataSize => 32)
        port map(D => INSTRUCTION,
                    Q => instRegToDecoder_INSTR,
                    clk => clk,
                    rst=> rst,
                    En => FSMToInstReg_MIAEN);

-- Instantiation of instruction decoder, required to get opcode parameters from
-- 32 bit instruction
DECODER: entity work.INST_DECODER
    generic map(dataSize => dataSize,
                    numRegistersBase => numRegistersBase)
    port map(INSTRUCTION => instRegToDecoder_INSTR,
                RT => WA,
                RA => RA,
                RB => RB,
```

```
                IMM => decodeToFSM_IMM,
                N => SH,
                OPCODE => decodeToFSM_OPCODE);

-- Instantiation of sequencer, to handle branching and MIA generation
SEQUENCER: entity work.SEQUENCER
    generic map(dataSize => dataSize)
    port map(PC_INC => PC_INC,
           PC => PC,
              PC_EN => FSMToSequencer_PCEN,
              clk => clk,
              rst => rst,
              MIA => MIA);

-- Instantiation of FSM to set processor control signals through each stage of instruction
-- execution
FSM: entity work.PROCESSOR_FSM
    generic map(busSize => busSize,
                    dataSize => dataSize,
                    numRegisters => numRegisters,
                    numRegistersBase => numRegistersBase)
    port map(clk => clk,
                rst => rst,
               OPCODE => decodeToFSM_OPCODE,
                IMM_in => decodeToFSM_IMM,
                IMM=> IMM,
                OEN=> OEN,
                S  => S,
                AL => AL,
                WEN => WEN,
                PC_EN => FSMToSequencer_PCEN,
                FLAGS => FLAGS,
                MIA_EN => FSMToInstReg_MIAEN);

end Behavioral;
```

Instruction Decoder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

------------------------------ INSTRUCTION DECODER ---------------------------------

-- Pulls opcode parameters out from 32 bit instruction based upon OPCODE.
-- Passes these values out to FSM/directly to processing unit.

----------------------------------------------------------------------------------

entity INST_DECODER is
     generic (dataSize : natural := 16;
                  numRegistersBase : natural := 4);
    Port ( INSTRUCTION : in  STD_LOGIC_VECTOR (31 downto 0); -- 32 bit input instruction from Instruction register
           RT : out  STD_LOGIC_VECTOR (numRegistersBase-1 downto 0); -- Target register to write to, equivalent to WA
           RA : out  STD_LOGIC_VECTOR (numRegistersBase-1 downto 0); -- Register to read from pulled from instruction
           RB : out  STD_LOGIC_VECTOR (numRegistersBase-1 downto 0); -- Register to read from pulled from instruction
           IMM : out  STD_LOGIC_VECTOR (dataSize-1 downto 0); -- Immediate value pulled from instruction
           N : out  STD_LOGIC_VECTOR (3 downto 0); -- Number of bits to shift/rotate by pulled from instruction
              OPCODE : out STD_LOGIC_VECTOR (5 downto 0)); -- OPCODE pulled from instruction
end INST_DECODER;

architecture Behavioral of INST_DECODER is
    signal OPCODE_internal : STD_LOGIC_VECTOR(5 downto 0); -- Internal signal to hold OPCODE, as output port cant be read
from
begin

OPCODE_internal <= INSTRUCTION(31 downto 26);

-- Values decoded from instruction are derived from OPCODE coding in excel spreadsheet

-- RA is always present in bits 5 to 9
RA <=
    INSTRUCTION(9 downto 5);

-- RT is always present on bits 0 to 4, except for storr and storo
-- Otherwise RT is not used, and so we can set Rt to 0000
RT <=
    INSTRUCTION(4 downto 0) when OPCODE_internal(5 downto 3) /= "101" else
    "00000";

-- RB allocation based upon opcode coding
RB <=
    INSTRUCTION(4 downto 0) when OPCODE_internal(5 downto 3) = "101" else
    INSTRUCTION(20 downto 16) when OPCODE_internal = "000001" or
        OPCODE_internal = "000010" or
        OPCODE_internal = "010001" or
        OPCODE_internal = "010010" or
        OPCODE_internal = "010011" else
    "00000";

-- IMM decoded for andi, ori, xor, addi, subi, loadi, stori
IMM <=
    INSTRUCTION(25 downto 10) when OPCODE_internal (5 downto 2) = "0001" or
       OPCODE_internal(5 downto 2) = "0101" or
        OPCODE_internal = "100000" or
        OPCODE_internal = "101000" else
    -- IMM is a resized and signed value written as OFFSET(9:0) in opcode coding for storo and loado
    -- So is resized and kept as signed (offsets can be negative)
   std_logic_vector(resize(signed(INSTRUCTION(19 downto 10)), dataSize)) when OPCODE_internal = "100110" or OPCODE_internal
= "101110" else-- Resize OFFSET to size of IMM for offset addressing
    std_logic_vector(resize(signed(INSTRUCTION(18 downto 10)), dataSize)) when OPCODE_internal(5 downto 4) = "11" else --
Resize OFFSET to size of IMM for branch
    "0000000000000000";

-- Decode of N for shl, shr, rol, ror
N <=
    INSTRUCTION(25 downto 22) when OPCODE_internal(5 downto 3) = "011" else
    "0000";

-- Output decoded OPCODE
OPCODE <= OPCODE_internal;

end Behavioral;
```

Sequencer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

------------------------------------ SEQUENCER ----------------------------------

-- Sequencer stores current value of program counter using a register
-- The value of the program counter is changed to that computed by the ALU on the PC INC
-- line when PC EN = 1 (generated by FSM). Memory instruction address (address of the next
-- instruction to execute.) is set to the value of PC.

---------------------------------------------------------------------------------------

entity SEQUENCER is
     generic (dataSize : natural);
    Port ( PC_INC : in  STD_LOGIC_VECTOR (dataSize-1 downto 0);
           PC : out  STD_LOGIC_VECTOR (dataSize-1 downto 0);
              PC_EN : in STD_LOGIC;
              clk : in STD_LOGIC;
              rst : in STD_LOGIC;
              MIA : out STD_LOGIC_VECTOR(7 downto 0));
end SEQUENCER;

architecture Behavioral of SEQUENCER is
    -- Internal program counter bus to allow MIA to read from it. Needed as cant read from output port.
    signal PC_internal : STD_LOGIC_VECTOR(dataSize-1 downto 0);
begin

-- Register to store current value of program counter.
-- Value of PC set to PC_INC when PC enable (generated by FSM) signal goes to 1
PC_reg: entity work.register_block
        generic map(dataSize => dataSize)
        port map(D => PC_INC,
                 Q => PC_internal,
                    clk => clk,
                    rst=> rst,
                    En => PC_EN);

-- MIA set to value of PC internal.
MIA<=
        std_logic_vector(resize(unsigned(PC_internal), 8));

-- Connect ports to internal signals
PC <=
        PC_internal;

end Behavioral;
```

Finite State Machine

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.DigEng.ALL;

------------ FINITE STATE MACHINE ------------------

-- This is the combinational logic block for the multi cycle architecture .
-- It uses a finite state machine to generate the control signals for the rest
-- of the processor.

-----------------------------------------------------
entity PROCESSOR_FSM is
    generic (dataSize : natural := 16;
                numRegisters : natural := 16;
                numRegistersBase : natural := 4;
                busSize : natural := 4);
    Port (      clk : in  STD_LOGIC;
                rst : in  STD_LOGIC;
                -------- INSTRUCTION CODING ---------
                -- OPCODE is the the six MSBs from the instruction bus.
                OPCODE : in STD_LOGIC_VECTOR(5 downto 0);

                -- Immediate value(IMM), and number of shift bits (N).
                IMM_in : in STD_LOGIC_VECTOR (dataSize-1 downto 0);

                -- FLAGS generated by ALU for branch condition detection
                FLAGS : in STD_LOGIC_VECTOR(6 downto 0);

                -------- CONTROL SIGNALS ----------
                -- Multiplexor selection bits.
                S   : out STD_LOGIC_VECTOR (3 downto 0);

                -- ALU inputs.
                AL : out  STD_LOGIC_VECTOR (3 downto 0);
                IMM: out STD_LOGIC_VECTOR  (dataSize-1 downto 0);

                -- Enables.
                WEN : out STD_LOGIC;
                MIA_EN : out STD_LOGIC;
                PC_EN : out STD_LOGIC;
                OEN : out STD_LOGIC
                );
end PROCESSOR_FSM;

architecture Behavioral of PROCESSOR_FSM is

    --------- STATES ----------
    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8);
    --------- INTERNAL SIGNALS -----------
    signal state, next_state : state_type;

    -- S bus for Multiplexors .
    signal S_internal : STD_LOGIC_VECTOR (3 downto 0);
    -- Flags from ALU .
    signal FLAG_SET : STD_LOGIC;

begin

state_assignment: process (clk) is
    begin
        if rising_edge(clk) then
            if (rst = '1') then
                state <= S0;
            else
                state <= next_state;
            end if;
        end if;
end process state_assignment;

-- Definitions for the state transitions .
fsm_process: process (state, OPCODE) is
begin
    case state is
        -- Fetch State.
            when S0 =>
                next_state <= S1;
        -- Reg Read State.
            when S1 =>
                if (OPCODE(5) = '0' and OPCODE /= "000000") or (OPCODE = "100111") then
                -- Arithmetic and Logic. Special case for Move instruction, as skips MemRW.
                    next_state <= S2;
                elsif OPCODE(5 downto 4) = "10" and OPCODE /= "100111" then
                -- Transfer. Exclude MemRW.
                    next_state <= S4;
                elsif OPCODE = "000000"  then
```

```
                -- If NOP instruction the return to S0 and load fetch instruction.
                    next_state <= S0;
                else
                -- Branch.
                    next_state <= S8;
                end if;

        -- ALU State (REG).
            when S2 =>
              next_state <= S3;

        -- Reg Write State (REG).
            when S3 =>
              next_state <= S0;

        -- ALU state (MEM).
            when S4 =>
                if (OPCODE(3) = '1') then
                -- If OPCODE is specific to store then go to State 5.
                    next_state <= S5;
                else
                    next_state <= S6;
                end if;

        -- MEM Read/Write State (MEM - STORE).
            when S5 =>
              next_state <= S0;

        -- MEM Read/Write State (MEM - LOAD).
            when S6 =>
              next_state <= S7;

        -- Reg Write State (MEM - LOAD).
            when S7 =>
              next_state <= S0;

        -- ALU State (BRANCH).
            when S8 =>
                    next_state <= S0;
    end case;
end process fsm_process;

-- Select bits for Multiplexors.
S <=
        -- s2 select set to 1 at State 0 to allow the PC to enter ALU
        "0100" when state = S0 else

        -- Jump and Branch instruction - s1 and s2 set to 1 to allow addition of pc and offset
        "1100" when state = S1 else

        -- If there is nothing on the immediate bus (therefore no immediate containing instruction)
        -- S1 and S2 mux set to 0 to allow RA and RB pass through.
        -- If Immediate is present then set S1 mux to 1 to allow IMM and RA pass through to ALU.
        "0000" when state = S2 and IMM_in = "0000000000000000" else
        "1000" when state = S2 else
        -- No instance where we manipulate Ra and RB inside ALU, so never
        -- need S1 at 0. Add Offset to Ra, but this is handled by IMM.
        -- IMM will be set to 0 by decoder for instructions that don't require it
        -- Having no effect when added by ALU.

        -- S4 set to 0 to allow write to WA from Aout reg.
        "0000" when state = S3 else

        -- No Rb input for ALU in any transfer instructions, only ever use Immediate and Ra.
        "1000" when state = S4 else

        -- S3 set to 0 in all Store and Load instructions at State 5/6 as to Directly address Memory (MDA).
        "0000" when state = S5 else
        "0000" when state = S6 else

        -- S4 set to 1 to allow the writing of data from Memory into the Register Bank.
        "0001" when state = S7 else

        "0000";

-- ALU OPCODE being set dependant on the OPCODE in the 32 bit instruction
AL <=
        -- Fetch will always require an increment on the PC
        "1000" when state = S0 else

        -- Branch Instruction verification stage (FLAG Generation) else no ALU activity
        "1010" when state = S1 and OPCODE(5 downto 4) = "11" else
        "0000" when state = S1 else

        -- ALU stage for all Logic and Arithmetic instructions (including move)
        "0000" when state = S2 and OPCODE = "000000" else  -- nop
        "1010" when state = S2 and OPCODE = "000001" else  -- add rt, ra, rb
        "1011" when state = S2 and OPCODE = "000010" else  -- sub rt, ra, rb
```

```
        "1010" when state = S2 and OPCODE = "000101" else  -- addi rt, ra, imm
        "1011" when state = S2 and OPCODE = "000110" else  -- subi rt, ra, imm
        "1000" when state = S2 and OPCODE = "001001" else  -- inc rt, ra
        "1001" when state = S2 and OPCODE = "001010" else  -- dec rt, ra
        "0111" when state = S2 and OPCODE = "010000" else  -- not rt, ra
        "0101" when state = S2 and OPCODE = "010001" else  -- or rt, ra, rb
        "0100" when state = S2 and OPCODE = "010010" else  -- and rt, ra,rb
        "0110" when state = S2 and OPCODE = "010011" else  -- xor rt, ra, rb
        "0100" when state = S2 and OPCODE = "010101" else  -- andi rt, ra, imm
        "0101" when state = S2 and OPCODE = "010110" else  -- ori rt, ra, imm
        "0110" when state = S2 and OPCODE = "010111" else  -- xori rt, ra, imm
        "1100" when state = S2 and OPCODE = "011001" else  -- shl rt, ra, n
        "1101" when state = S2 and OPCODE = "011010" else  -- shr rt, ra, n
        "1110" when state = S2 and OPCODE = "011101" else  -- rol rt, ra, n
        "1111" when state = S2 and OPCODE = "011110" else  -- ror rt, ra, n
        "1010" when state = S2 and OPCODE = "100111" else  -- move rt, ra

        -- No ALU activity as State 3 is a REG WRITE
        "0000" when state = S3 else

        -- ALU Stage for all Transfer Instructions (excluding Move)
        "1010" when state = S4 else

        -- No ALU activity as State 5 and 6 are MEM READ/WRITE
        "0000" when state = S5 else
        "0000" when state = S6 else

        -- No ALU activity as State 7 is a REG WRITE
        "0000" when state = S7 else

        "0000";

-- Write to memory if in state S4 and OPCODE is a store
OEN <=
        '1' when state = S5 else
        '0';

-- Write enable should only be high at write states (State 3 and 7)
WEN <=
        '1' when state = S3 else
        '1' when state = S7 else
        '0';

-- Pass through of OPCODE parameters from Decoder.
IMM <= IMM_in;

-- MIA_EN only High in State 0 as this is when instruction are fetched. Gets enabled when Jumping, to jump to new MIA
immediately.
MIA_EN <=
            '1' when state = S0 or (state = S1 and OPCODE(5 downto 3) = "111") else
            '0';

-- PC_EN is set high in state 1 as this is when PC+ enters the sequencer.
-- PC_EN is set high in state 9 as this is when PC+ (from Branch/Jump instruction) enters the sequencer.
PC_EN <=
        '1' when state = S1 else
        '1' when state = S8 and ((FLAG_SET = '1') or opcode (5 downto 3) = "111") else
        '0';

-- based upon the branch OPCODE Flag set will be used as a reference to see if the condition to branch has been met.
FLAG_SET <= FLAGS(0) when OPCODE(2 downto 0) = "001" else
            FLAGS(1) when OPCODE(2 downto 0) = "010" else
            FLAGS(2) when OPCODE(2 downto 0) = "011" else
            FLAGS(3) when OPCODE(2 downto 0) = "100" else
            FLAGS(4) when OPCODE(2 downto 0) = "101" else
            FLAGS(5) when OPCODE(2 downto 0) = "110" else
            FLAGS(6) when OPCODE(2 downto 0) = "111" else
            '0';

end Behavioral;
```

Dual Port Memory

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


-------------------------------- DUAL PORT MEMORY ----------------------------------

-- This source file functions as a wrapper for the generated distributed IP core created
-- by Xilinx.

-----------------------------------------------------------------------------------
entity DP_MEM is
    Port(clk : in STD_LOGIC;
          INST_ADDRESS : in  STD_LOGIC_VECTOR (6 downto 0); -- Address for instruction, connected to MIA
          DATA_ADDRESS : in  STD_LOGIC_VECTOR (6 downto 0); -- Address for DATA
          DATA_In : in STD_LOGIC_VECTOR(31 downto 0); -- Data to write to memory core
          WEn : in STD_LOGIC; -- Write enable for DATA_IN to DATA_ADDRESS
          Data_Out : out STD_LOGIC_VECTOR(31 downto 0); -- Output data presebt at DATA address
          INSTR_DATA : out STD_LOGIC_VECTOR(31 downto 0) -- 32 bit instruction output present at INST_ADDRESS
          );
end DP_MEM;

architecture Behavioral of DP_MEM is
  COMPONENT RAM_int
  PORT (
    a : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
    d : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    dpra : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
    clk : IN STD_LOGIC;
    we : IN STD_LOGIC;
    spo : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    dpo : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT;
begin

-- INST_DATA_ADDRESS <=
Internal_RAM : RAM_int
  PORT MAP (
    a => DATA_ADDRESS,
    d => DATA_In,
    dpra => INST_ADDRESS,
    clk => clk,
    we => WEn,
    spo => Data_Out,
    dpo => INSTR_DATA
  );

end Behavioral;
```

MMU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

------------------------------ MEMORY MANAGEMENT UNIT ------------------------------

-- Memory management unit to allow allow a dual port 32 bit wide distributed memory IP core
-- to read and write 16 bit data effectively into only the upper memory region (64 - 128).

-- MEMORY WRITE: Achieved by first reading an entire 32 bit data segment from the write location,
-- and modifying either the upper or lower 16 bits of read data to the desired data to be written
-- based upon the LSB of the write address. This is then written to the desired address.

-- MEMORY READ: Read 32 bit value from data at desired read location, and return to the processor
-- either the upper or lower half of this data dependent upon the LSB of the modified read address.
-- Read address is generated by tagging the MSB of the shortened read address with a '1' to ensure
-- only reading/writing the upper half of memory.

-- INSTRUCTION READ: MIA generated by control unit based upon PC is first shortened by removing
-- 2 MSB's. We then tag the MSB with a 0 so that we can only address from 0 to 63. This modified
-- MIA is then passed to the memory core as the instruction address. As this exists on its own port
-- the corresponding output port from the memory returns the 32 bit instruction directly to the control
-- unit

----------------------------------------------------------------------------------

entity MMU is
    Port ( ------------ TO/FROM PROCESSOR -----------
              MIA : in  STD_LOGIC_VECTOR (7 downto 0);                -- Memory instruction address
           DMEM_RW_ADDRESS : in  STD_LOGIC_VECTOR (15 downto 0); -- Desired address to read/write to in DMEM
           DMEM_OUT_TOPROC : out  STD_LOGIC_VECTOR (15 downto 0);-- Data out from the memory core to the
processing unit
           OEn : in  STD_LOGIC; -- Write enable signal to Output Reg/ memory core
           DMEM_IN_FROMPROC : in  STD_LOGIC_VECTOR (15 downto 0);-- Data to store into the data memory from
the processing unit
              -------------- TO/FROM DMEM --------------
           INST_ADDRESS : out  STD_LOGIC_VECTOR (6 downto 0);    -- Modified MIA address, to retrieve next
instruction from lower half of mem
              DATA_ADDRESS : out STD_LOGIC_VECTOR(6 downto 0);        -- Modified DATA address, to retrieve
data from upper half of memory
           DMEM_DATA_TOWRITE : out  STD_LOGIC_VECTOR (31 downto 0); -- Modified data to write to processor,
using upper/lower half of data at write location
           DMEM_DATA_READOUT : in  STD_LOGIC_VECTOR (31 downto 0); -- Data read in from the memory
           DMEM_WEn : out  STD_LOGIC;                                    -- Actual Write enable signal to
Output Reg/ memory core, piped from OEn input port
              ----------------- TO/FROM MMIO ----------------
           MMIO_DATA : out  STD_LOGIC_VECTOR (15 downto 0);
           MMIO_WEn : out  STD_LOGIC;
              PB_IN : in STD_LOGIC);
end MMU;

architecture Behavioral of MMU is
    -- Modified MIA address
    signal MIA_Short : STD_LOGIC_VECTOR(6 downto 0);
    -- Modified data read/write address
    signal DMEM_RWA_Short : STD_LOGIC_VECTOR(6 downto 0);
    -- Temporary signal for storing correctly formed data packet for write address
    signal DMEM_IN_internal : STD_LOGIC_VECTOR(31 downto 0);
    -- Bus to store STD_LOGIC signal from pushbutton into, so processor can read expected 16 bit value
    signal PB_IN_bus : STD_LOGIC_VECTOR(15 downto 0);
begin

-- Throw away the 2 most significant bits, down to 6 bits, between 0 and 63
    MIA_Short(5 downto 0)  <= MIA(5 downto 0);

-- We need to tag with a 0 to address lower half of memory (0 - 63)
    MIA_Short(6) <= '0';

-- Set instruction address to the modified MIA bus value
    INST_ADDRESS <= MIA_Short;
```

```
-------------- READ -------------------

-- Pushbutton connection. Upper 15 bits of this address are set to 0.
-- The LSB is set to the bit from pushbutton, so a pressed button returns h0001.
    PB_IN_bus(15 downto 1) <= (others => '0');
    PB_IN_bus(0) <= PB_IN;

-- Ignore upper bits of desired memory read address as address lines to dual port memory
-- are only 7 bits wide.
    DMEM_RWA_Short(5 downto 0) <= DMEM_RW_ADDRESS(5 downto 0);

-- Tag the memory address for read/write with a 1 to address the upper half of memory always.
    DMEM_RWA_Short(6) <= '1';

-- Set 16 bit data output to processor equal to either the upper or lower half of the 32 bit
-- data at memory read location dependent upon LSB of read address
    DMEM_OUT_TOPROC <=
        PB_IN_bus when DMEM_RW_ADDRESS = "0000000111110000" else
        DMEM_DATA_READOUT(31 downto 16) when DMEM_RWA_Short(0) = '1' else
        DMEM_DATA_READOUT(15 downto 0) when DMEM_RWA_Short(0) = '0' else
        "0000000000000000";

-- Set data address line to memory to modified Read/Write address
DATA_ADDRESS <= DMEM_RWA_Short;

------------ WRITE -------------------

-- Pass through Write enable signal generated by FSM to core and to output register.
DMEM_WEn <= OEn;
MMIO_WEn <= OEn;

-- Set upper half of data value to write to upper half of memory location
DMEM_IN_internal(31 downto 16) <=
        DMEM_DATA_READOUT(31 downto 16) when DMEM_RWA_Short(0) = '0' and OEn = '1' else
        DMEM_IN_FROMPROC;

-- Set lower half of data value to write to lower half of memory location
DMEM_IN_internal(15 downto 0) <=
        DMEM_DATA_READOUT(15 downto 0) when DMEM_RWA_Short(0) = '1' and OEn = '1' else
        DMEM_IN_FROMPROC;

-- Set data to write to dual port memory to modified write data
DMEM_DATA_TOWRITE <= DMEM_IN_internal;

-- Set value to output to LED register to 16 bit data from processor when write address
MMIO_DATA <=
        DMEM_IN_FROMPROC when DMEM_RW_ADDRESS = "0000000111111000" else
        "0000000000000000";

end Behavioral;
```

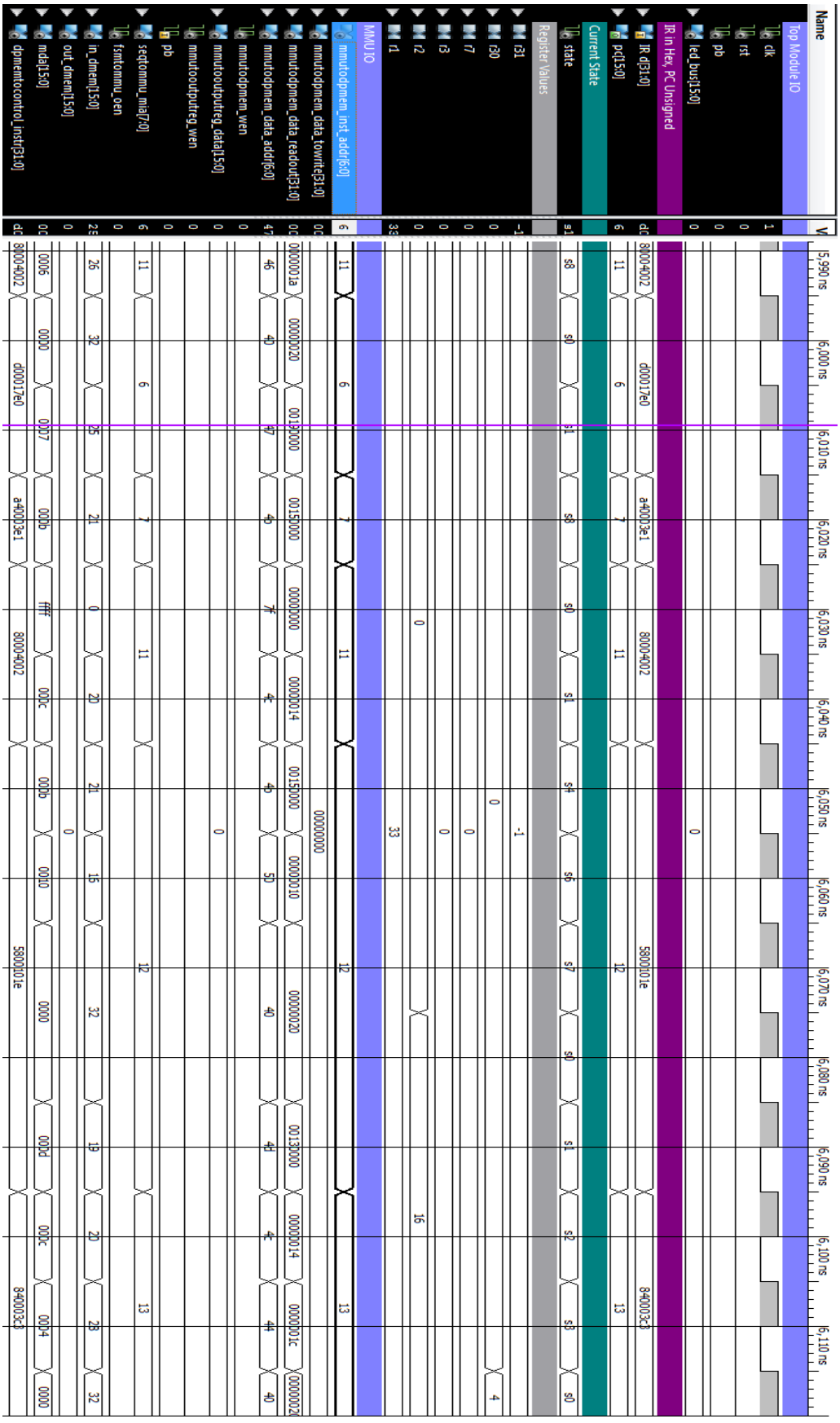
Testbench Code

Testbench code is minimal as the test program ensures functionality of all constituent components. Stimuli to the circuit come from the instructions stored in memory (.coe file) and subsequently the decoder/FSM. We simply simulate a pushbutton press to break out of the initial loop:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY CPU_TEST IS
END CPU_TEST;

ARCHITECTURE behavior OF CPU_TEST IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT top_level
    PORT(
         clk : IN  std_logic;
         rst : IN  std_logic;
         LED_BUS : OUT  std_logic_vector(15 downto 0);
         PB : IN  std_logic
            );
    END COMPONENT;

   --Inputs
   signal clk : std_logic := '0';
   signal rst : std_logic := '0';
   signal PB : std_logic := '0';

    --Outputs
   signal LED_BUS : std_logic_vector(15 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
   uut: top_level PORT MAP (
          clk => clk,
          rst => rst,
          LED_BUS => LED_BUS,
          PB => PB
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
        wait for 100 ns;
      -- hold reset state for 100 ns.
        rst <='1';
        wait for clk_period*2;
        rst <= '0';
        wait for clk_period*2;

        -- No need for other stimuli as instructions come from memory coe file.
        -- Simulate pushbutton press to break out of initial branch
        PB <= '1';
        wait for 300 ns;
        PB <= '0';
      wait;
   end process;

END;
```

Simulation

6-11-12 Sequence



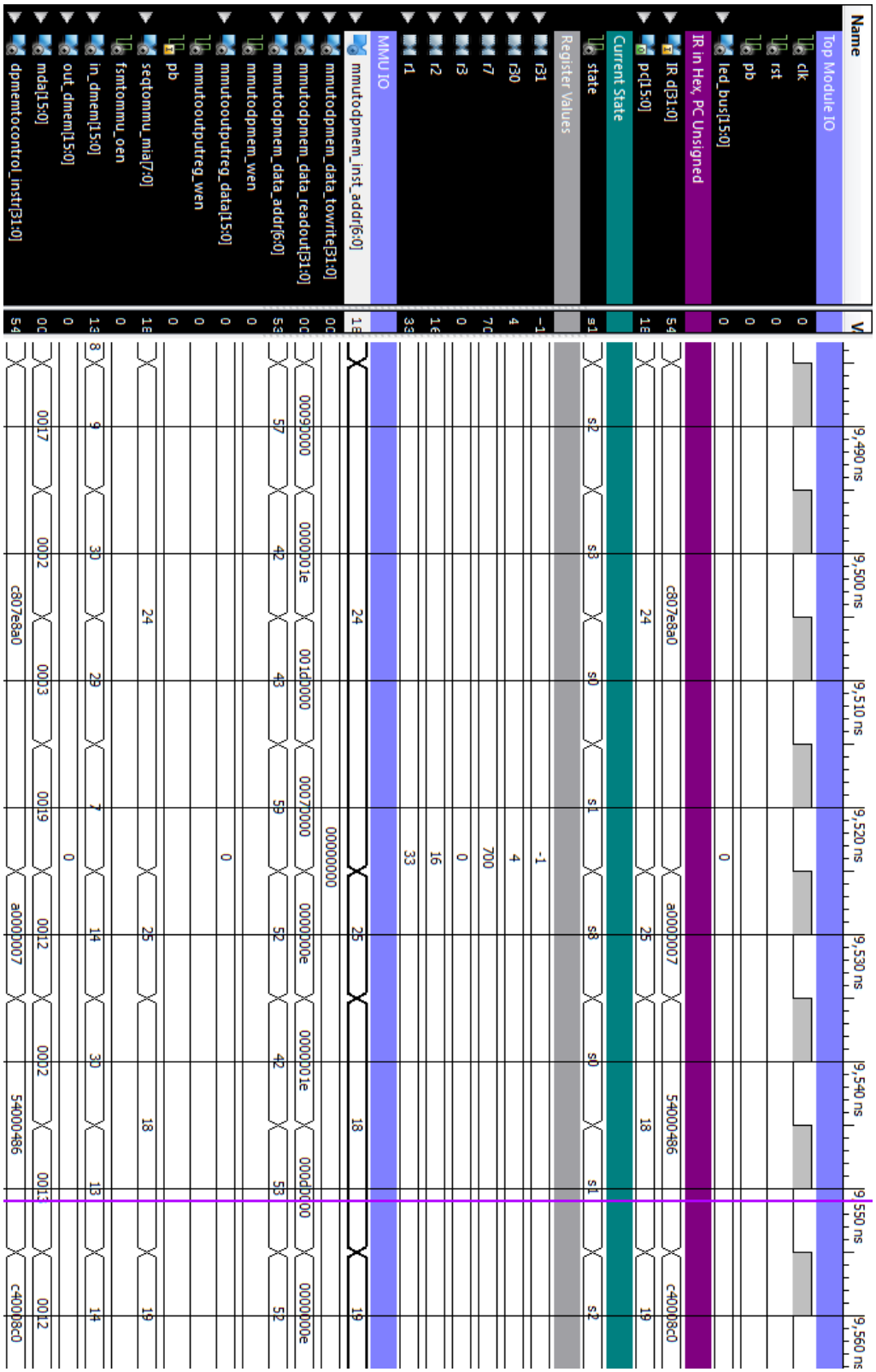
The first instruction in this sequence is a branch with instruction code of D00017E0 on Register 31 for a value less than 0. Now that r31 has been decremented by the previous loop, the value is -1 and so the branch condition is met, advancing execution forwards by 5 to the loadi instruction, 80004002.

This instruction loads the value of data memory at h0010, which corresponds to DMEM[16]. DMEM[16]'s value contains the value 16, which can be seen committed to the target register 2 at the end of the S7 FSM stage.

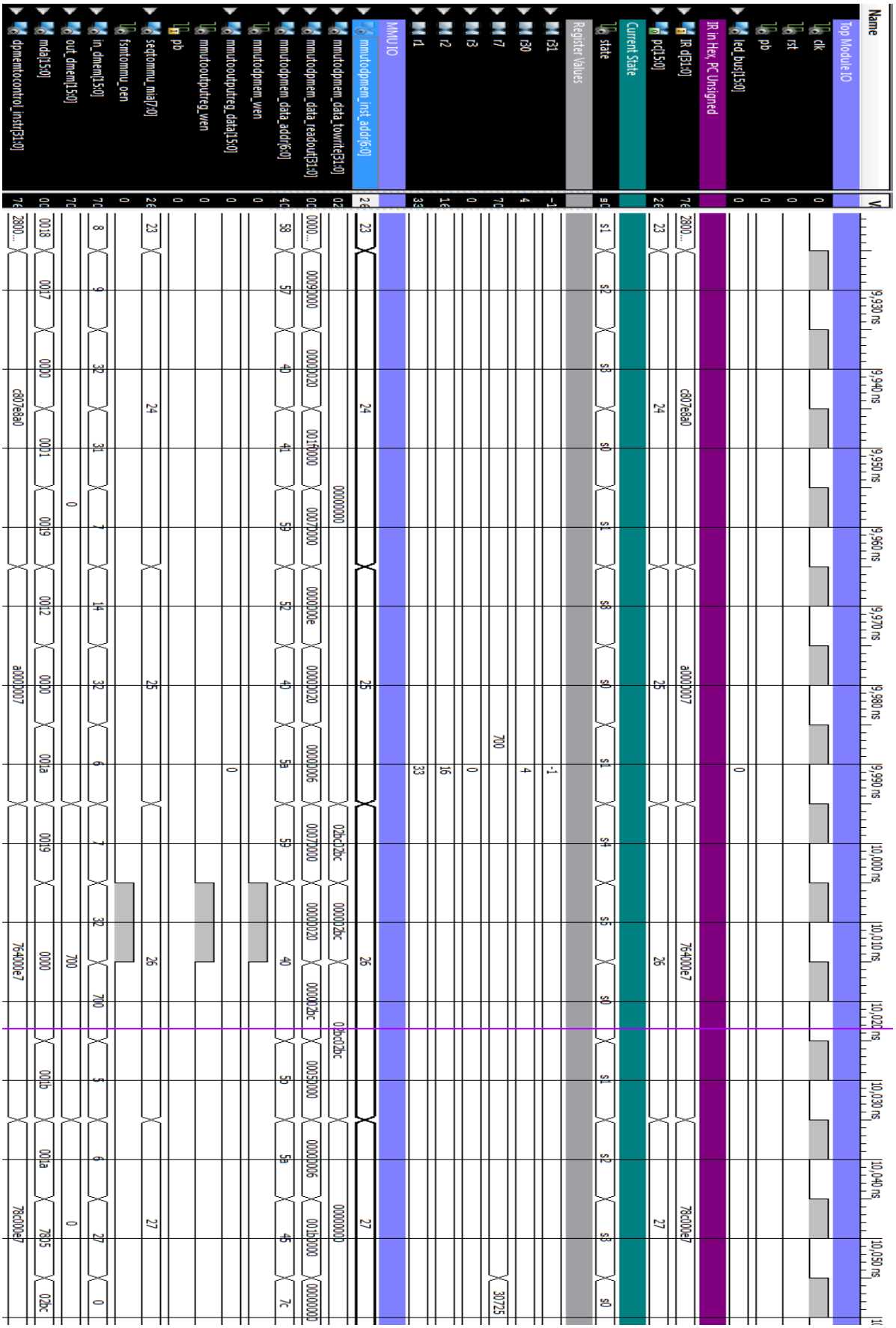
The PC is advanced, and the new ori instruction is loaded (5800101E). This bitwise ORs the value of register 0 with an immediate of 4, which will return 4. The result of which is committed to the processors memory at register 30, and this can be seen at the S8 FSM state.

24-25-26 Instruction Sequence

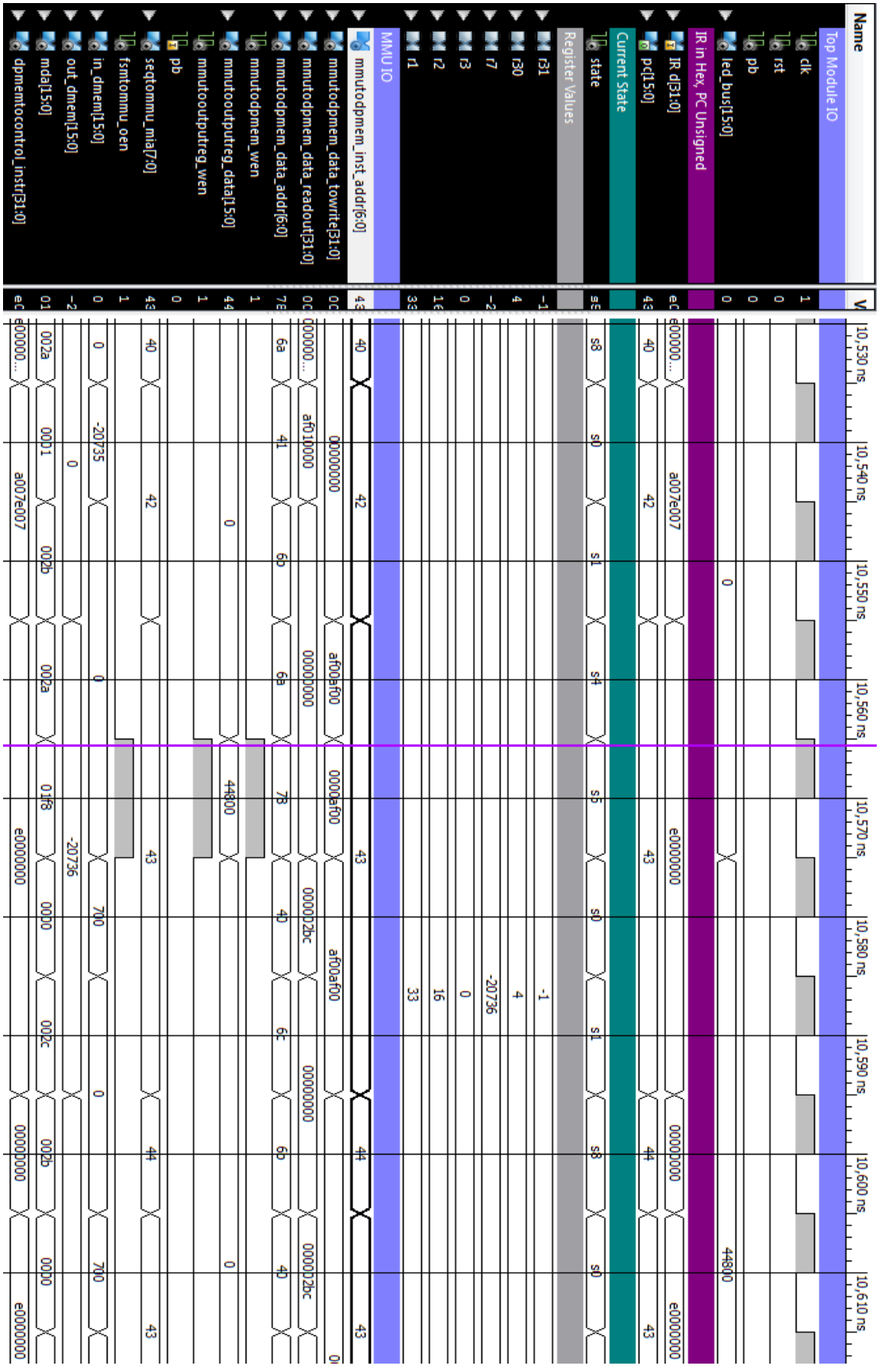
The first instruction (C807E8A0) in this sequence is a branch on register 5 for a condition of not equal to 0. If register 5 not equal to zero the branch execution will occur causing the program to jump back 6 instructions, which happens in the below simulation:



In this particular case where the program transitions through 24-25-26 the branch condition is not met and as a result the program increments normally to the following STORI instruction, and subsequently the rol that causes r7 to be equal to 30725 due to the signed number wrapping around. This can be seen in the simulation to the left.



42-43 Instruction Sequence



The first instruction in this sequence stores value of R7 into DMEM[h01f8]. This cannot be seen in data memory as the memory contents are too large to show on the simulation. Fortunately, the MMU inputs and outputs show the write to memory of the contents of R7. Noting the *MDA* line, the hexadecimal address 01f8 can be seen at the point that the *FSMToMMU_Oen* signal goes high, enabling write. The intended data to be written to the address is -20736, but the MMU must first read back the value contained at the address and concatenate it onto the write data depending on the LSB of the address. As a result, the actual data written to the memory core is 0000AF00, which is 44800.

The jump instruction is a plus 0 offset, which will cause the program to stay in an infinite loop of this instruction. It is worth noting that the s8 state loads the next instruction (in this case nothing/NOP) however this is only fetched and never executed as in s0 the desired instruction is loading in.

HDL Synthesis

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================

Synthesizing Unit <top_level>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\top_level.vhd".
        dataSize = 16
        numRegisters = 32
        numRegistersBase = 5
        busSize = 4
    Summary:
        no macro.
Unit <top_level> synthesized.

Synthesizing Unit <PROCESSING_UNIT>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\PROCESSING_UNIT.vhd".
        dataSize = 16
        numRegisters = 32
        numRegistersBase = 5
        shiftSize = 4
    Summary:
        inferred   4 Multiplexer(s).
Unit <PROCESSING_UNIT> synthesized.

Synthesizing Unit <register_block_1>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\register.vhd".
        dataSize = 16
    Summary:
        no macro.
Unit <register_block_1> synthesized.

Synthesizing Unit <D_FF>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\D_FF.vhd".
    Found 1-bit register for signal <Q>.
    Summary:
        inferred   1 D-type flip-flop(s).
Unit <D_FF> synthesized.

Synthesizing Unit <ALU>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\ALU.vhd".
        dataSize = 16
    Found 16-bit adder for signal <A[15]_B[15]_add_23_OUT> created at line 37.
    Found 16-bit adder for signal <A[15]_GND_12_o_add_27_OUT> created at line 1253.
    Found 16-bit subtractor for signal <A[15]_B[15]_sub_22_OUT<15:0>> created at line 38.
    Found 16-bit subtractor for signal <A[15]_GND_12_o_sub_26_OUT<15:0>> created at line 1320.
    Found 16-bit shifter rotate right for signal <A[15]_X[3]_rotate_right_13_OUT> created at line 3021
    Found 16-bit shifter rotate left for signal <A[15]_X[3]_rotate_left_15_OUT> created at line 3012
    Found 16-bit shifter arithmetic right for signal <A[15]_X[3]_shift_right_17_OUT> created at line 2982
    Found 16-bit shifter logical left for signal <A[15]_X[3]_shift_left_19_OUT> created at line 2973
    Found 16-bit 13-to-1 multiplexer for signal <INT_OUT> created at line 23.
    Found 16-bit comparator greater for signal <FLAGS<3>> created at line 57
    Found 16-bit comparator greater for signal <FLAGS<4>> created at line 60
    Summary:
        inferred   1 Adder/Subtractor(s).
        inferred   2 Comparator(s).
        inferred  11 Multiplexer(s).
        inferred   4 Combinational logic shifter(s).
Unit <ALU> synthesized.

Synthesizing Unit <register_bank>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\register_bank.vhd".
        M = 32
        baseM = 5
        dataSize = 16
    Summary:
        no macro.
Unit <register_bank> synthesized.

Synthesizing Unit <tri_state_buffer>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\tri_state_buffer.vhd".
        dataSize = 16
    Found 1-bit tristate buffer for signal <Data_out<15>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<14>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<13>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<12>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<11>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<10>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<9>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<8>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<7>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<6>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<5>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<4>> created at line 17
```

```
    Found 1-bit tristate buffer for signal <Data_out<3>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<2>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<1>> created at line 17
    Found 1-bit tristate buffer for signal <Data_out<0>> created at line 17
    Summary:
        inferred  16 Tristate(s).
Unit <tri_state_buffer> synthesized.

Synthesizing Unit <decoder>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\decoder.vhd".
        M = 32
    Summary:
        inferred   1 Multiplexer(s).
Unit <decoder> synthesized.

Synthesizing Unit <CONTROL_UNIT>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\CONTROL_UNIT.vhd".
        dataSize = 16
        numRegisters = 32
        numRegistersBase = 5
        busSize = 4
    Summary:
        no macro.
Unit <CONTROL_UNIT> synthesized.

Synthesizing Unit <register_block_2>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\register.vhd".
        dataSize = 32
    Summary:
        no macro.
Unit <register_block_2> synthesized.

Synthesizing Unit <INST_DECODER>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\INST_DECODER.vhd".
        dataSize = 16
        numRegistersBase = 5
    Summary:
        inferred   7 Multiplexer(s).
Unit <INST_DECODER> synthesized.

Synthesizing Unit <SEQUENCER>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\SEQUENCER.vhd".
        dataSize = 16
    Summary:
        no macro.
Unit <SEQUENCER> synthesized.

Synthesizing Unit <PROCESSOR_FSM>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\PROCESSOR_FSM.vhd".
        dataSize = 16
        numRegisters = 32
        numRegistersBase = 5
        busSize = 4
    Found 4-bit register for signal <state>.
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------------------------
    | States             | 9                                              |
    | Transitions        | 15                                             |
    | Inputs             | 5                                              |
    | Outputs            | 15                                             |
    | Clock              | clk (rising_edge)                              |
    | Reset              | rst (positive)                                 |
    | Reset type         | synchronous                                    |
    | Reset State        | s0                                             |
    | Power Up State     | s0                                             |
    | Encoding           | auto                                           |
    | Implementation     | LUT                                            |
    -----------------------------------------------------------------------
    Found 1-bit 8-to-1 multiplexer for signal <FLAG_SET> created at line 56.
    Summary:
        inferred  26 Multiplexer(s).
        inferred   1 Finite State Machine(s).
Unit <PROCESSOR_FSM> synthesized.

Synthesizing Unit <DP_MEM>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\DP_MEM.vhd".
    Summary:
        no macro.
Unit <DP_MEM> synthesized.

Synthesizing Unit <MMU>.
    Related source file is "D:\ISE Projects\Year 2\Computer Arch\CPUAct\MMU.vhd".
WARNING:Xst:647 - Input <MIA<7:6>> is never used. This port will be preserved and left unconnected if it
belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
    Summary:
        inferred   6 Multiplexer(s).
Unit <MMU> synthesized.
```

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 1
 16-bit addsub                                        : 1
# Registers                                            : 624
 1-bit register                                       : 624
# Comparators                                          : 2
 16-bit comparator greater                            : 2
# Multiplexers                                         : 57
 1-bit 2-to-1 multiplexer                             : 2
 1-bit 8-to-1 multiplexer                             : 1
 16-bit 2-to-1 multiplexer                            : 23
 32-bit 2-to-1 multiplexer                            : 3
 4-bit 2-to-1 multiplexer                             : 25
 5-bit 2-to-1 multiplexer                             : 3
# Logic shifters                                       : 4
 16-bit shifter arithmetic right                      : 1
 16-bit shifter logical left                          : 1
 16-bit shifter rotate left                           : 1
 16-bit shifter rotate right                          : 1
# Tristates                                            : 1024
 1-bit tristate buffer                                : 1024
# FSMs                                                 : 1
# Xors                                                 : 1
 16-bit xor2                                          : 1
```

Synthesis Warnings

Program	All Implementation Messages - Errors, Warnings, and Infos
xst	Xst:647 - Input <MIA<7:6>> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.
xst	Xst:2042 - Unit tri_state_buffer: 16 internal tristates are replaced by logic (pull-up yes):

- The first warning occurs as a result of only using the lower 5 bits from MIA within the MMU. The top two bits are not used, and the MSB is tagged as a 0 within the MMU. This could be mitigated by having a smaller 5 bit MIA bus come out from the control unit, and resizing upwards when within the MMU. For the sake of matching the specification exactly, this was not implemented.
- The second warning occurs as a result of our register bank generation. We generate an array of tri states for every generated register in order to read/write to the register. For register 0, the tri state array is not connected to a register, but is instead grounded to 0. As a result of this, the tristates are replaced by VCC.

UCF File

```
NET "clk" LOC = L15;

NET "PB" LOC = F5;

NET "rst" LOC = T15;

NET "LED_BUS[15]" LOC = N12;

NET "LED_BUS[14]" LOC = P16;

NET "LED_BUS[13]" LOC = D4;

NET "LED_BUS[12]" LOC = M13;

NET "LED_BUS[11]" LOC = L14;

NET "LED_BUS[10]" LOC = N14;

NET "LED_BUS[9]" LOC = M14;

NET "LED_BUS[8]" LOC = U18;
```

FPGA Board Upload Output

All correct bits lit up on LED bus to show MSB's of 44800 once reset and PB pressed: