Computer Structure Lab - Fall 2000
Dr. Guy Even

Handout #6: A simplified DLX: design, testing, timing, and programming

In this handout we describe a simplified DLX-Architecture which you will be implementing on the RESA-CPU. It is assumed that the reader is familiar the following topics: (a) The DLX architecture and implementation that was taught in the course "Introduction to Digital Computers" as described in the book by Müller and Paul. The architecture described here is a simplified version. (b) The RESA bus and memory accesses. In particular, instruction fetch and execution of load/store instructions are done by initiating bus transactions.

1 General Architecture

1.1 Comparison between the DLX and the simplified DLX

The simplified DLX-Architecture differs from the DLX-Architecture in a few ways:

1. There are only two instruction formats: I-Type and R-Type. The J-Type-Format is canceled and unified into the I-Type by using the 16-bit immediate constant in the I-Type-Format.
2. The memory is word addressable which means that only words (32 bits) can be read from and written to the main memory. This means that successive word addresses in memory have successive addresses (instead of increments by 4 as in the DLX which is byte-addressable). The fact that the simplified DLX is word accessible simplifies load/store instruction execution because it is not necessary to shift the data anymore.
3. The instruction set is reduced.

1.2 Architectural Registers and Modules

The architectural registers of the simplified DLX are all 32 bits wide and listed below.

- 32 General Purpose Registers (GPR): `R0` to `R31`. Note that `R0` always holds the value 0;
- Program Counter (PC);
- Instruction Register (IR); and
- Special Registers: `MAR, MDR, A, B` and `C`;

The main modules in the datapath of the simplified DLX are as follows:

- The GPR-File is a dual-port RAM which supports either two reads or one write.

- The ALU supports 2's complement integer addition, subtraction, comparison and bitwise logical operations.
- The Shifter supports logical left and right shifts by one position.

1.3 Instruction Formats

There are two instruction formats:

1. An instruction in the I-Type-Format is divided into four fields depicted below.

6	5	5	16
Opcode	*RS1*	*RD*	*immediate*

2. An instruction in the R-Type-Format is divided into five fields depicted below. Note that the 5 bit field used in the DLX-Architecture for the shift amount (SA) is not used in the simplified DLX.

6	5	5	5	5	6
Opcode	*RS1*	*RS2*	*RD*		*Function*

1.4 Instruction Set

We list below the instruction set of the simplified DLX. Note that *imm* denotes the value of the immediate field in an I-Type-Instruction. *sext(imm)* denotes the 2's complement sign extension of *imm* to 32 bits.

- Load/Store-instructions:

Load/Store	Semantics
`lw RD RS1 imm`	RD := M(sext(imm)+RS1)
`sw RD RS1 imm`	M(sext(imm)+RS1) := RD

- Immediate–instructions:

Instruction	Semantics
`addi RD RS1 imm`	RD := RS1 + sext(imm)

- Shift–/Compute–Instructions:

Instruction	Semantics
`slli RD RS1`	RD := RS1 $<<$ 1
`srli RD RS1`	RD := RS1 $>>$ 1
`add RD RS1 RS2`	RD := RS1 $+$ RS2
`sub RD RS1 RS2`	RD := RS1 $-$ RS2
`and RD RS1 RS2`	RD := RS1 $\wedge$ RS2
`or RD RS1 RS2`	RD := RS1 $\vee$ RS2
`xor RD RS1 RS2`	RD := RS1 $\oplus$ RS2

- Test–Instructions:

Instruction	Semantics
`s`*rel*`i RD RS1 imm`	RD := 1, if condition is satisfied,
	RD := 0 otherwise
if *rel* =`lt`	test if RS1 $<$ sext(imm)
if *rel* =`eq`	test if RS1 $=$ sext(imm)
if *rel* =`gt`	test if RS1 $>$ sext(imm)
if *rel* =`le`	test if RS1 $\leq$ sext(imm)
if *rel* =`ge`	test if RS1 $\geq$ sext(imm)
if *rel* =`ne`	test if RS1 $\neq$ sext(imm)

- Jump–instructions:

Instruction	Semantics
`beqz RS1 imm`	PC = PC + 1 + sext(imm), if RS1 = 0
	PC = PC + 1, if RS1 $\neq$ 0
`bnez RS1 imm`	PC = PC + 1, if RS1 = 0
	PC = PC + 1 + sext(imm), if RS1 $\neq$ 0
`jr   RS1`	PC = RS1
`jalr RS1`	R31 = PC+1; PC = RS1

- Miscellaneous–instructions:

Instruction	Semantics
`special-nop`	causes transition to Init/Fetch states
`halt`	causes transition to HALT state

Encoding of the Instruction Set

Tables 1 and 2 specify the binary encoding of the instructions.

2 Implementation

2.1 Datapath

The datapath of the simplified DLX is depicted in Figure 1. The proposed datapath is not a typical datapath; it lacks busses and drivers. There are two reasons for this: (a) the interface with the I/O Control Logic does not requires busses; and (b) to protect the hardware we do no allow using drivers. Note that control signals as well as some connections between the environments are not depicted in Fig. 1.

ALU environment

Even though the 32-bit adder in the ALU can be designed using VHDL, we suggest that you use three 16-bit adder/subtractors from the Xilinx library (ADSU16) to build a 32-bit conditional sum adder. The Xilinx library component is optimized with respect to the FPGA's technology and the usage of the adder/subtractor eliminates the need to invert the second argument in a subtraction instruction.

It is very likely that the ALU will lie on the critical path. A VHDL design will probably end up much slower and costlier. Moreover, a costly design might create additional delay due to wire and routing delays.

IR[31 : 26]	Mnemonic	Semantics
Data Transfer		
100 011	lw	RD = M(sext(imm)+RS1)
101 011	sw	M(sext(imm)+RS1) = RD
Arithmetic, Logical Operation		
001 011	addi	RD = RS1 + sext(imm)
Test Set Operation		
011 rel	s rel i	RD = (RS1 rel sext(imm))
011 001	sgti	RD = (RS1 $>$ sext(imm))
011 010	seqi	RD = (RS1 $=$ sext(imm))
011 011	sgei	RD = (RS1 $\geq$ sext(imm))
011 100	slti	RD = (RS1 $<$ sext(imm))
011 101	snei	RD = (RS1 $\neq$ sext(imm))
011 110	slei	RD = (RS1 $\leq$ sext(imm))
Control Operation		
000 100	beqz	PC = PC + 1 + (RS1 $= 0$? sext(imm) : 0)
000 101	bnez	PC = PC + 1 + (RS1 $\neq 0$? sext(imm) : 0)
010 110	jr	PC = RS1
010 111	jalr	R31 = PC + 1; PC = RS1
Miscellaneous Instructions		
110 000		no operation (not supported by DLX assembler)
111 111		stop program (not supported by DLX assembler)

Table 1: I-type Instructions

The ALU supports bitwise logical instructions and comparison instructions. A comparison is implemented by a subtraction followed by examining the sign of the result and whether the result equals zero. More implementation details appear in the book of Müller and Paul.

Shifter environment

The shifter is a 32-bit left/right logical shifter. This means that a zero is pushed in from the right (left) in case of a left (right) shift. The control inputs of the shifter are: shift and right. The shift input indicates whether a shift should take place (otherwise the output equals the input). The right signal indicates whether the shift is a right shift. (Why do we want the shifter to support also the identity function?)

The GPR environment

The GPR environment is identical to that of the Load/Store Machine.

IR$[5:0]$	Mnemonic	Semantics
Shift Operation		
000 000	slli	RD = RS1 << 1
000 010	srli	RD = RS1 >> 1
Arithmetic, Logical Operation		
100 011	add	RD = RS1 $+$ RS2
100 010	sub	RD = RS1 $-$ RS2
100 110	and	RD = RS1 $\wedge$ RS2
100 101	or	RD = RS1 $\vee$ RS2
100 100	xor	RD = RS1 $\oplus$ RS2

Table 2: R-type Instructions (in R-type instructions IR$[31:26] = 0^6$)

2.2 Control

Figure 2 depicts the state diagram of the control of the simplified DLX. Note that the transitions leading to the Init State and Fetch State are depicted as one "edge" although in reality these are two separate transitions. Access to the memory is done via the Memory Access Control module as described for the Load/Store Machine. Note that Figure 2 does not depict the `reset` signal generated by the I/O control logic. The `reset` signal causes a transition in the control of the DLX to "init" state.

The control signals that appear in Fig. 2 have the following meanings:

1. `step_en` is the `step_en` signal from from the I/O control logic.
2. `busy` is the `busy` signal from the memory access control.
3. `D2..D13` are the monomials corresponding to the decoding of the instructions.
4. `else` corresponds to an illegal instruction, namely, when all the monomials `D2..D13` are not satisfied.
5. `bt` (branch taken) corresponds to the event that the condition of a conditional branch is satisfied.

The following control signals are used to communicate between the datapath and the control:

1. `IRCE, PCCE, ACE, BCE, CCE, MARCE, MDRCE`: clock enable signals of the registers.
2. `S1SEL0, S1SEL1, S2SEL0, S2SEL1, DINTSEL, MDRSEL, ASEL`: select signals of the muxes.
3. `ADD, TEST, SHIFT, RIGHT, ALUF0, ALUF1, ALUF2`: signals that control the functionality of the ALU.
4. `ITYPE`: active when the current instruction is an i-type instruction.

5. `JLINK`: active during a “jalr” instruction.

You may want to add some control signals that are active during the “init” and “halt” states. These control signals can be used by the Monitor Slave (i.e. Logic Analyzer).

Table 3 lists the active control signals in each state. (An RTL instruction describes the action that takes place during the state).

Table 4 lists the control signals of the simplified DLX.

Table 5 lists the monomials computed by the control. These monomials are correct under the assumption that all instructions are legal. We henceforth refer to decoding under the assumption that all the instructions are legal as *imprecise decoding*.

For example, to simplify the decoding of a transition to the ALU state, we suggest to check if $IR[31:28] = 0^4$ and $IR[5] = 0$. That means that if $IR[31:28] = 0^4$, the bits $IR[27:26]$ are ignored. If they happen not to be equal to zero, then the instruction is an illegal instruction. Nevertheless, in this case we choose to treat it as an ALU instruction. Now, one might ask what about the bits of $IR[4:0]$? Suppose we decode an instruction as an ALU instruction, but the bits in $IR[4:0]$ do not encode a legal instruction? We do not wish to add a transition from the ALU state to the HALT state, so some ALU instruction must take place in this case.

The advantage of imprecise decoding is that it is easier, and therefore, faster. The justification for using imprecise decoding is that legal programs are decoded correctly by imprecise decoding and that it is very easy to write a program that checks if a legal program is stored in a given file (or memory segment). Note that if imprecise decoding is used, then a illegal instruction is decoded as a legal one.

It is possible to almost perfectly combine the advantages of precise decoding and imprecise decoding by performing imprecise decoding. Every instruction execution has a “commit” stage (namely, write-back, store, or changing of the PC in a jump). One could condition the commit stage of execution upon the legality of the instruction. The legality of the instruction is computed while the instruction is executed (not in the decode stage), and so time is not wasted on checking the legality of legal instructions. We will not consider this more sophisticated option, and use only imprecise decoding.

The conclusion is that we suggest to do an imprecise decoding of the instructions. That mean that some illegal instructions are considered as legal encodings of other instructions. We leave it to you to choose which instructions are decoded in such cases.

The usage of imprecise decoding means that we cannot reach the Halt state using an illegal instruction. We therefore, add a HALT instruction that causes a transition to the Halt state.

3 The RESA working environment

3.1 Special DLX assembly instructions

The DLX assembly language includes for convenience two additional instructions which are aliases for DLX instructions.

- MOVE. A move instruction has two arguments (e.g. `move RD RS1`), and is an alias for `addi RD RS1 0`.

Name	RTL Instruction	Active Control Signals
Fetch	$IR = M(PC)$	MR, IRce
Decode	$A = RS1$, $B = RS2$ $PC = PC + 1$	Ace, Bce, S2sel[1], S2sel[0] PCce, add
Alu	$C = A$ op B	S1sel[0], Cce
TestI	$C = (A$ rel $imm)$	S1sel[0], S2sel[0], Cce, test, Itype
AluI(add)	$C = A + imm$	S1sel[0], S2sel[0], Cce, add, Itype
ShiftI	$C = A$ shift sa $sa = 1, (-1)$	S1sel[0], Cce, Itype DINTsel, shift (,right)
Adr.Comp	$MAR = A + imm$	S1sel[0], S2sel[0], MARce, add
Load	$MDR = M(MAR)$	MDRce, Asel, MR, MDRsel
Store	$M(MAR) = MDR$	Asel, MW
CopyMDR2C	$C = MDR(\gg 0)$	S1sel[0], S1sel[1], S2sel[1], DINTsel, Cce
CopyGPR2MDR	$MDR = B(\ll 0)$	S1sel[1], S2sel[1], DINTsel, MDRce
WBR	$RD = C$ (R-type)	GPR_WE
WBI	$RD = C$ (I-type)	GPR_WE, Itype
Branch	branch taken?	
Btaken	$PC = PC + imm$	S2sel[0], add, PCce
JR	$PC = A$	S1sel[0], S2sel[1], add, PCce
Save PC	$C = PC$	S2sel[1], add, Cce
JALR	$PC = A$ $R31 = PC$	S1sel[0], S2sel[1], add, PCce GPR_WE, jlink

Table 3: The active control signals in each state

Signal	Value	Semantics
ALUf[2:0]		Controls the functionality of ALU
Rce		Register clock enable
S1sel[1:0]	00 01 10 11	PC A B MDR
S2sel[1:0]	00 01 10 11	B IR 0 1
DINTsel	0 1	ALU Shifter
MDRsel	0 1	DINT DI
Asel	0 1	PC MAR
shift		explicit Shift-Instruction
right		Shift to the right
add		Forces an addition
test		Forces a test (in the ALU)
MR		Memory Read
MW		Memory Write
GPR_WE		GPR write enable
itype		Itype-Instruction
jlink		jump and link

Table 4: List of control signals

Nontrivial DNF	Target State	IR[31 : 26]	IR[5 : 0]
D1	Init/Fetch	110***	******
D2	Alu	0000**	1*****
D4	ShiftI	0000**	0*****
D5	AluI	001***	******
D6	TestI	011***	******
D7	Adr.Comp	10****	******
D8	JR	010**0	******
D9	JALR	010**1	******
D12	Branch	0001**	******
D13	Copy GPR2MDR	**1***	******
/D13	Load	**0***	******
bt	Btaken	$AEQZ \oplus IR[26]$	
/bt	Fetch	$/(AEQZ \oplus IR[26])$	

Table 5: Monomials of the control

- 2. NOP. A no-operation instruction has no arguments and is an alias for `addi R0 R0`. Note that the Special-NOP instruction is not supported by the DLX assembler. The execution of NOP and Special-NOP are different: one goes through the ALU state and the other transitions immediately to the Fetch or Init States. A Special-NOP instruction can be used by the "dc" directive.

3.2 Memory map

Your DLX design should work in the RESA environment. We have already described the RESA bus, the usage of the I/O Control Logic, and the RESACTRL Program. We now elaborate on the memory map of the RESA.

Recall that the bus protocol is address based and that every "item" has a 32 bit address (for example, the status register in your Logic Analyzer has a 32 bit address). You control (and are aware of) only the 8 least significant bits of this address. This means that there is a "map" that specifies which addresses are allocated to which boards. Within the memory board (which hosts the main memory of your DLX design) not all addresses may be used by you. We outline the memory map of the RESA below.

1. 0x00000000 - 0x0000ffff : 64KW of ROM. This ROM stores a test program for the DLX design. We elaborate more on this program in Section 3.4.

2. 0x00010000 - 0x007fffff : This address space is not allocated to any device.

3. 0x00800000 - 0x008fffff: 1MW of RAM. This space is used for the user programs.

4. Some of the other address space is allocated to various devices (CPU board, monitor slave, etc.). *Do not attempt to access addresses in this space since it might damage the RESA*.

3.3 Translating logical addresses to physical addresses

In the Load/Store Machine, the logical addresses of the machine are translated to the physical address space 0x00800000 - 0x008fffff. The fixed nature of this translation should be clear now that we have described the memory map.

In the DLX Machine, we want to have two modes. In the first mode, no translation takes place (i.e. the logical and physical addresses are identical). In the second mode, we wish to use the same translation that is used in the Load/Store Machine. The reason we wish to have both modes is that we would like to run the test programs (which means that no translation should take place at all) and we would like to run user programs (that are written under the assumption that the user's logical memory space starts at 0x00000000).

Our goal is not to have to worry about the memory translation when we design the DLX. For example, we want to set the `PC` register to zero whenever a `reset` signal is active (just as we did with the Load/Store Machine). This uniform abstraction is obtained by using a Memory Mapping Unit (MMU) that maps a physical address to a logical one. The inputs to this mapping are the `AO` (address out) bus of the DLX and a `mem_map` signal. The output of the MMU is the `MMU_AO` bus that connects to the I/O control logic. The `mem_map` signal tells the MMU how to translate the logical address into a physical one. **The MMU should set address bits in positions** $[31 : 24]$ **to zero unconditionally to avoid access to restricted areas.**

We suggest that the `mem_map` signal be taken from bit 23 of the command register. This bit controls the functionality of the MMU as follows: If the memory space should start at 0x0 (i.e. the test program is running), then `MMU_AO`=`AO`. If the memory space should start at 0x00800000 (i.e. a user program is running), then `MMU AO` = `AO` + 0x00800000. You may assume that the user program's logical address space is limited to $[0, 2^{23} - 1]$ to simplify the MMU design.

3.4 The Test Program

The test program consists of nine test procedures. The source of the test program is posted in the lab's homepage. An outline is depicted below.

```
*-----------------------------------------------------------------*
* Outline of DLX test program
*-----------------------------------------------------------------*
        pc=0x00000000
test0:  ...

RESA:   dc 0x52455341           * "RESA"
USER:   dc 0x00800001           * start address of user's program
RAM:    dc 0x00800000

test1:  ...
test8:  dc 0xc0000008           * Special-NOP

        lw   R30 R0  RAM        * R30 := address of RAM
        lw   R1  R0  RESA
        lw   R2  R30 0x0        * read from M[0x800000]
        xor  R2  R2  R1         * if (M[0x800000]=="RESA")
        beqz R2  start          *   then jump to user's program
        bnez R2  ramtst         * else perform memory test
        ...

start:  lw   R30 R0 RAM         * load start address of RAM
        lw   R1  R0 USER        * load start address of user's program
        jalr R1                 * jump to user's program
```

```
end:    beqz R0  end
```

The first 8 tests are not “destructive” (i.e. they do not write to the user’s memory space). After 8 tests are executed by test program, the 9th test reads and writes to the whole user space to test the memory (which is something one would not like to happen if a user’s program is stored in that area). Therefore, before the 9th test starts, it is possible to jump to the user’s program instead of running the 9th test. This is done as follows: The value stored in the address 0x00800000 is compared with a special value (given the name ”RESA” in the listing above). If the values are equal, then the 9th test is skipped and a jump to address 0x00800001 is performed. Otherwise, the 9th test takes place.

3.5 Executing a user’s program after the test program

If you would like to start running your program only after the first 8 tests are executed, you can use the following mechanism.

```
*--------------------------------
* having your program run after the first 8 tests
*--------------------------------
        pc= 0x800000            * beginning of user’s memory space

        dc 0x52455341           * "RESA"
        beqz R0 start           * first instruction

a:      dc 10                   * 0x800002
b:      dc 20                   * 0x800003
erg:    ds 1

start: lw   R1  R30  a
        lw   R2  R30  b
        add  R1  R1   R2
        sw   R1  R30  erg       * erg = a + b

end:   beqz R0  end
```

Note that (a) In address 0x00800000 the constant corresponding to the ”RESA” special value is stored. (b) The jump to address 0x00800001 is implemented in the test program by a jalr instruction (it could be also implemented by a jr instruction). The register R1 is set to hold the value 0x00800001 (which can not be stored by a 16-bit immediate constant) using a variable that is initialized to this value in the test program. (c) The register R30 is set to hold the value 0x00800000 also by using a variable that is initialized to this value in the test program. The R30 register can then be used by the user’s program as the offset to compute addresses so that the user’s program variables are accessible (e.g. load and store instructions).

Note also that a failure in one of the 8 tests (due to an error in the DLX’s functionality) might cause an endless loop, in which case the user’s program is not executed at all.

3.6 Executing a user’s program without the test program

We avoid running the test program by mapping the logical address 0x00000000 to the physical address 0x00800000, as described in the section on translating addresses.

3.7 The ID register

One of the read-back inputs of the Monitor Slave should be fed by a 32-bit register, called the ID-register. The value stored in the ID-register should be the code of your lab group.

By reading this value, it can be verified that the design is indeed yours. You should use this to make sure that you are executing your own design.

4 Homework

Each question counts as one assignment. Submission deadline is given next to each question. Failure to meet these deadlines will mean that you will not be able to complete all the lab assignments - the lab closes after the semester ends. No more assignments or quizzes will be given during this lab. We highly recommend that you divide the load between students in the same group, otherwise you will find the burden too high.

1. (*submit at the beginning of your* 11*th lab meeting.*) Design and test the simplified DLX.

   - Submit schematics of the project and of the new modules. Submit results of testing using test vectors of each module.
   - Submit VHDL file of the control. Describe how you tested the control. Submit results of testing the control using test vectors.
   - Test your design to see if the correct RTL instruction is executed in each state. Write short DLX programs that test every instruction. Note that some instructions must tested more than once (e.g. "branch taken" and "branch not taken"). Store these programs in the RAM of the I/O_SIMUL module. Build a project from your DLX design and the modified I/O_SIMUL module. Simulate this project. Submit: (1) Your programs. For every instruction, list the states that are traversed. (2) Waveforms of a simulation of your design. Add remarks to the waveforms that show the traversed states, the RTL instruction that were executed, and what they did.

   After completing these stages successfully, it is likely that your design can be run on the RESA and monitored to see if it functions properly.

2. (*submit at the beginning of your* 12*th lab meeting.*) Implement and test your design on the RESA.

   - Implement your design. You should easily meet the timing requirements (clock rate at least 5 MHz). Submit a report generated by the implementation software stating that your design is fast enough and that implementation was successful.
   - Re-use the test programs you used to test the RTL instructions on the RESA. Use the Logic Analyzer to verify that the RTL instructions are executed properly. Submit printouts of the monitoring with remarks explaining why you got the right results.
   - Read the RESA test program. Describe how it tests the design.
   - Run the RESA test procedure on your design. Your design should pass the test! Get Marko's approval of your design.

3. (*submit at the beginning of your* 13*th lab meeting.*) Try to design the fastest design you can. You should at least meet the 10MHZ threshold. Implement your design and meet the timing requirements (clock rate at least 10 MHz). There are a few ways to try to decrease the feasible clock period.

   - The first method is to try to reduce delays due to routing in the FPGA. This can be done by "helping" the place & route tools. You will be told more on how this is done.
   - The second method is to identify the critical path and try to shorten its delay. This method is applicable provided that there are only a few critical paths.
   - The third technique can be used if you failed in shortening the delay of critical paths. Suppose the critical path includes the ALU. A way to solve this critical path is to allow two cycles for the ALU. This requires a change in the control so that the each of the corresponding states are divided into two states. We do not recommend using this method (for example, the software will still report timing problems because it does not know that the signal is not sampled after one cycle). You should be able to solve the timing problem without it.

   After you meet the timing requirements, Implement and test your design on the RESA. Note that the timing analysis done by the software tools is often too pessimistic. Your design may be fast enough even if the software reports otherwise.

   The 3 fastest designs submitted among those submitted on-time, will receive a 50% bonus for this assignment.

4. (*submit at the beginning of your* 14*th lab meeting.*) Write a DLX program that computes shortest paths in an undirected graph using Dijkstra's algorithm.

   The input should be as follows (starting from address 0x008000002):

   (a) $n \in \{2, 1024\}$ - the number of nodes of the graph. The vertex is set is $\{v_0, \ldots, v_{n-1}\}$.
   (b) $m \in \{1, 1024 \cdot 11\}$ - the number of edges of the graph.
   (c) A list of edges and edge lengths. For each edge, three numbers are given: i, j, ℓ, where (i, j) denotes the edges and ℓ denotes the length of the edge (always positive).

   The output should be as follows (starting from address 0x00800C000):

   (a) A list of distances: $d_1, d_2, \ldots, d_{n-1}$, where d_i is the length of the shortest path from v_0 to v_i.

   Submit:

   (a) A hand-written explanation of your implementation: what stages you have, how data is stored in your algorithm, what is the complexity of you algorithm. *A 50% bonus on this assignment will be given to handouts submitted on-time with a running time that is* $O(m \log n)$.
   (b) A printout of your Assembly program.

(c) Test your program on three graphs each containing 1024 nodes: (a) A path, where the leftmost vertex is v_0 and the rightmost vertex is v_{1023}. All edges have unit length. (b) A hypercube. The vertex set is $\{0, 1\}^n$, and there is an edge between u and v if their binary representations agree on all but one bit. All edges have unit length. (c) A graph of your choice (but not with unit edge lengths!).

Submit a printout comparing the results of your computation on these graphs with the results of a C program (or any other high level language). Write how long it took the program to run on each graph.

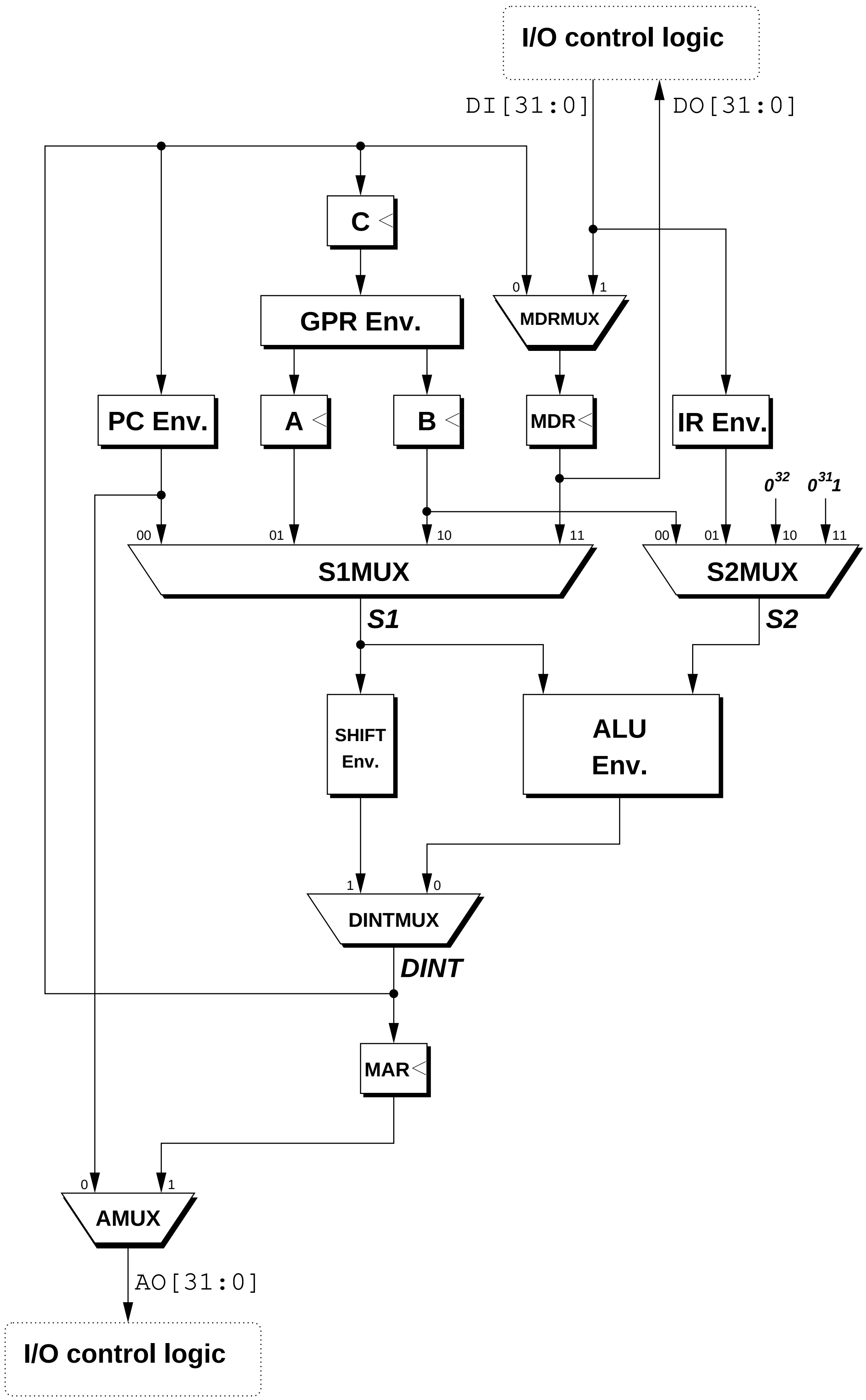

Figure 1: Datapath of the simplified DLX machine

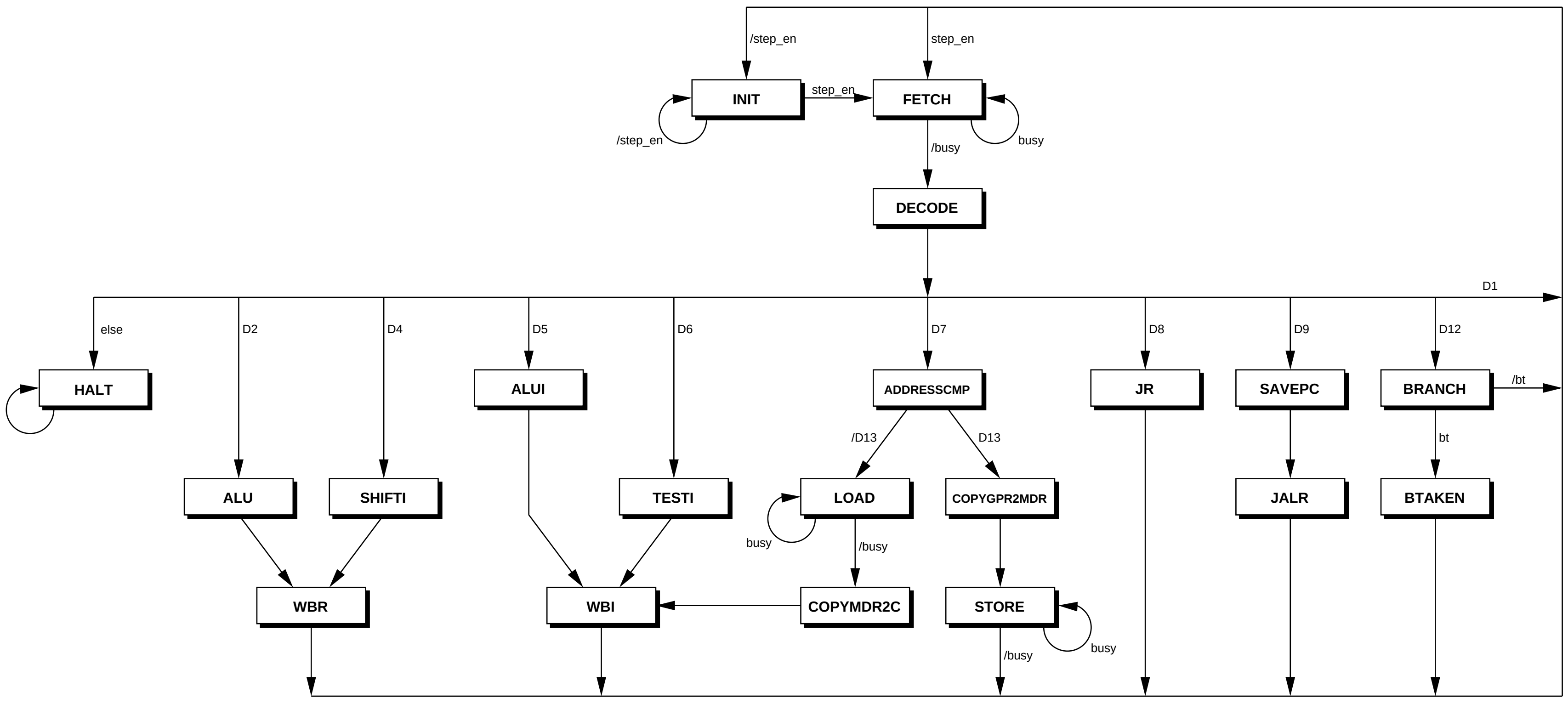

Figure 2: Finite state diagram of the control of the simplified DLX machine