Design for a simplified DLX (SDLX) processor

Rajat Moona
moona@iitk.ac.in

In this handout we shall see the design of a simplified DLX (SDLX) processor. We shall assume that the readers are familiar with the design of circuit elements such as combinatorial circuits using standard gates, sequential circuit elements such as flip-flops and latches and circuit building techniques using tri-state logic elements. We shall first present the instruction set architecture for the SDLX processor. We then describe the datapath design and hardwired control for the SDLX processor.

Instruction Set Architecture for the SDLX

The SDLX processor is a 32-bit architecture. It can perform operations on 32-bit data in the most natural manner. For example, the SDLX processor can add two 32-bit numbers using a single instruction. All instructions in the SDLX processor are 32-bit (4 bytes) wide. The SDLX processor implements only the integer operations on the data values. The floating-point instructions are not used in the SDLX. Complex integer instructions such as `mul` or `div` are also excluded from the instruction set of the SDLX.

SDLX has 32-bit memory addressing capabilities. It supports byte addressable memory. Thus the size of the address space of the SDLX processor is 2^{32} bytes. The operands larger than one byte are stored in multiple memory locations. The order of storage in such cases is big-endian. The most significant byte is stored first in the lower address. The least significant byte is stored last in the higher address. For example, if an integer 0x12345678 is stored in memory at address 520, it means to say that byte 0x12 is stored in memory at address 520, byte 0x34 in memory at address 521, byte 0x56 at address 522 and byte 0x78 at address 523.

SDLX supports 8 bit (byte), 16 bit (half word) and 32 bit (word) data types. Each of these can be signed integers or unsigned integers. SDLX has different instructions to handle these values, as we shall see later.

SDLX Register Set

The SDLX processor has 32 general-purpose registers, each 32-bit wide in size. Registers are named `R0` to `R31`. In addition, register `R0` is hardwired to `0`. A value that can ever be read from this register is 0. This register, even though is permitted to be the destination of an instruction, can never be used to hold any value. Thus a value written in this register is lost.

Having such a register often simplifies the instruction set in many ways. For example, there is no need to have an instruction like `neg`, which can be implemented using the `sub` instruction with `R0` as its first operand. Many processors have similar kind of a register in their architecture. Prominent examples are Sun-SPARC and MIPS processors.

Register `R31` of this processor has one special function in addition to the usual use. This register can be used as link register. SDLX processor does not have any hardware implementation of the stack unlike many other processors. The link register is used to save the address to the return instruction in case a function call is made. The stack can be built in the software. Some processors such as Sun-SPARC also have a similar feature but they use a separate link register.

The processor has one 30-bit program counter register (PC). It is used to read the instructions from the memory. Even though the address to the memory is 32-bit wide, 30-bit instruction address is enough as all instructions are four bytes wide and these are stored with a four-byte alignment in the memory. Thus the valid addresses for the instructions are 0, 4, 8, 12… all of which are a multiple of 4. The two least significant bits of the instruction addresses are always 0. Therefore it is enough to store just 30-bits of the instruction address in the PC register. The PC register therefore stores instruction address divided by 4.

Unlike many other processors, SDLX has no register for storing the special flags of the operations (such as carry, zero etc.). There is no special register called flags register or processor status word (PSW) register.

SDLX Instructions

SDLX instructions are categorized in the following four categories.

1. R type triadic instructions
2. R-I type triadic instructions
3. R type dyadic instructions
4. J type instructions

R Type triadic instructions have three register operands. The bit coding for such instructions is as given below.

31 26	25 21	20 16	15 11	10 6	5 0
Opcode (0)	RS1	RS2	RD	0	Func code

The R type triadic instruction format is used for instructions that involve three register-operands. These instructions (and other SDLX instructions) have six-bit operation code (opcode) field. The value of opcode for R type triadic instructions is 0. RS1 and RS2 indicate the two source registers (each field is 5 bits to indicate one of the 32 registers) for the instructions. The result of the instruction is stored in a register indicated by RD field in the instruction. The actual ALU operation to be performed is indicated by Func code. Five bits in this instruction format are unused and should remain 0 as shown.

The instructions that use this coding are ALU instructions (ADD, SUB, AND, OR, XOR, SLL (shift left logical), SRL (shift right logical), SRA (shift right arithmetic), ROL (rotate left), ROR (rotate right), SLT (signed less than comparison), SGT (signed greater than), SLE (signed less than or equal to comparison), SGE, UGT, ULT, ULE, UGE etc.). In shift instructions, one register provides the data, the second register provides the shift count and the result is stored in the destination register. In comparison instructions, two source registers are compared and the result of the comparison (0xFFFFFFFF for TRUE and 0 for FALSE) is stored in the destination register.

R-I Type triadic instructions have three operands. One of these operands is an immediate number (16 bit signed). The other source operand is necessarily a register. The destination operand is also a register. The bit coding for this kind of instructions is as shown below.

31 26	25 21	20 16	15 0
Opcode	RS1	RD	Immediate constant

The R-I type triadic instruction format is used for the following instructions.

- ALU: ADDI, SUBI, ORI, ANDI, XORI, SLLI, SRLI, SRAI, SLTI, SGTI, SLEI, SGEI, ULTI, UGTI, ULEI, UGEI, LHI (load upper half of a register with a number derived from the RS1 and an immediate constant). In shift instructions, the shift count is provided as an immediate constant. In comparison instructions, the second operand is the immediate constant (treated as signed number for signed comparison and as unsigned number for unsigned comparison). LHI instruction is different from the standard DLX processor. In SDLX, LHI instruction causes, the register RS1 to be read and its upper half is replaced by the immediate constant before storing it in the register RD. Note that the contents of the register RS1 are not changed unless RD==RS1.
- DATA Transfer instructions: LB (load signed byte), LBU (load byte unsigned), LH, LHU, LW (load word), SB (store byte), SH (store half word), SW. In these instructions, address of the memory is specified as the contents of the register RS1 added with 16 bit immediate constant treated as signed number. In SDLX load instructions, register RD is updated with a new 32-bit number. For example, in case of LB instruction, a byte is read from the specified location of the memory, sign extended to 32 bits and stored in register RD. Similarly, in case of LBU, a byte is read from the specified location of the memory, zero padded to make it 32 bit number and stored in register RD. In case of loading half word or word data, lower address of the memory provides the most significant byte. For store instructions, register RD provides the data to be stored in the memory. The multi-byte data is stored most significant byte first.
  The load and store instructions support only one kind of addressing mode. The effective address is computed as RS1+16bit immediate sign extended to 32 bits. Using this several other addressing modes can be simulated. For example, if RS1 is 0 (for register `R0`), this addressing mode simulates the direct addressing. Similarly if the immediate constant is 0, it simulates the register indirect addressing.

R type dyadic instructions involve only two operands and use only one source register. The format of the instructions is as follows.

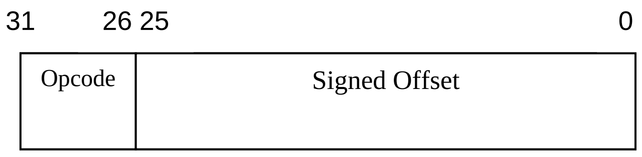
31 26	25 21	20 16	15 0
Opcode	RS1	0	Signed offset

The bit field 16-20 is unused in this format and should be 0. This kind of instruction coding is primarily used for the control transfer instructions as described below.

BNEZ (branch if not equal to zero), BEQZ (branch if equal to zero) are two conditional branch instructions supported in the SDLX. The control is transferred to instruction at address ($+4+4*Signed offset) depending upon the value in register RS1. The symbol $ is used to denote the memory address of the control transfer instruction that is being executed. ($+4 is the instruction stored immediately after the control transfer instruction). For example, BNEZ instruction will compare the register against zero and if the register has a value other than 0, the control is transferred to the instruction stored at ($+4+4*Signed offset). All control transfer instructions use a notion called *delayed branch* which roughly means that the instruction immediately following the control transfer instruction is executed irrespective of whether branch is taken or not. We shall see more about the delayed branch later.

The other control transfer instructions that use this coding are JR (jump with register RS1 providing the base address) and JALR (jump and link with register RS1 providing the base address). In case of JR, the instruction at address $+4 is executed and after that the execution starts from the instruction at address RS1+4*signed offset. Two least significant bits of the register RS1 are ignored (treated as 0) in order to ensure that the instructions are aligned at four byte boundaries. JALR instruction is similar to the JR instruction except that $+8 (the address of the instruction that shall execute upon return from the function call) is stored in the link register R31.

J type instruction coding is used for PC relative unconditional jump instructions. The bit coding for this instruction is as follows.



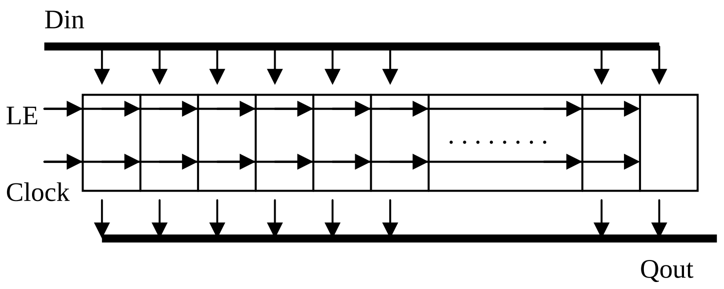
The instructions that use this coding are J and JAL. In case of J instruction, the instruction at $+4 is executed before reading instruction at address $+4+4*Signed Offset. In addition to this, in case of JAL instruction address $+8 is stored in the link register R31.

Datapath design for the SDLX

Register File

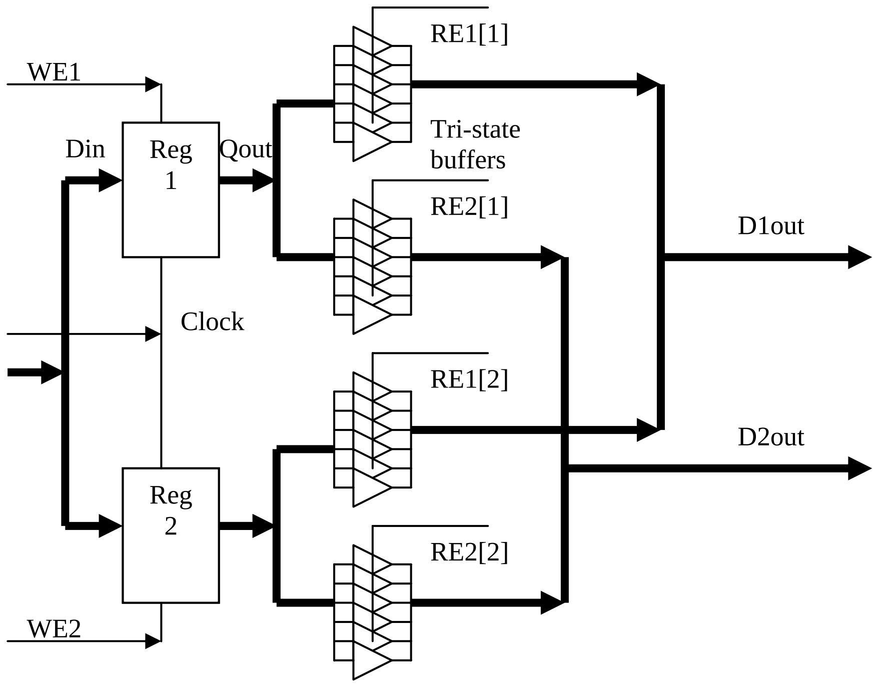
In the execution of the SDLX instructions, there may be up to three registers involved where two of them are read and the third one is updated. Therefore, a register file in the SDLX must have at least three ports (two read and one write). For registers we shall assume a circuit based on the flip-flops that use a positive edge of the clock to load the value. As a rule we shall not put logic on the clock to avoid clock delays/skews and other clock related problems.

Register files are composed using parallel load and parallel out registers, each 32 bit wide. The registers are made using the flip-flops as follows.



There are several flip-flops (32 in total for one single 32-bit register). A single clock drives each flip-flop. In addition, there is a control signal called LE (latch enable). The contents of the flip-flops are changed only when the LE is 1. Otherwise, the flip-flop outputs reflect the old value itself. At the end of the clock pulse (high to low transition), values on the Din bus gets stored in the flip-flop. The Qout bus always shows the contents of the register.

Using such a register, it is possible to make a multi-port register file. In order to do so several register outputs have to be connected on a common bus. That can be done using tri-state buffers. Here we show a simple interconnection of two registers in a register file with one write port and two read ports.

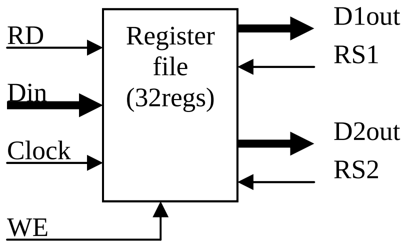


There are two separate output buses, D1out and D2out. The data from a register appears on the bus if the corresponding RE signal is active. RE1 signal is primarily used for activating the output on D1out bus. Similarly RE2 signal is used to activate the output on D2out bus. At one time, only one of the RE1[1] or RE1[2] signal can be active. Similarly only one of two RE2 signals can be active at any time. While data from the Din bus can be written on both the registers simultaneously by activating WE1 and WE2 signals, this feature is usually not required in any processor. At one time, only one of the ports is activated and data is written in that register.

For a 32-register register file with two output ports and one input port, there need to be 32 WE, 32 RE1 and 32 RE2 signals. Since only one of these signals in each of the groups can be active, a 5 bit to 32 decoder can be used to generate 32 of these signals. Thus a register file will need the following control inputs apart from the three buses.

5-bit destination register number (RD), 5-bit source register on D1out (RS1), 5-bit source register on D2out (RS2), 1 clock line and 1 WE signal. The WE signal is used to permit the data transfer from Din to any of the registers (selected using RD lines). If WE signal is 0, no register is updated. Thus the write enable signals for individual registers are derived from 5-bit RS and one bit WE controls.

We shall use the following representation for the 32-register register file.



Arithmetic and Logic Unit (ALU)

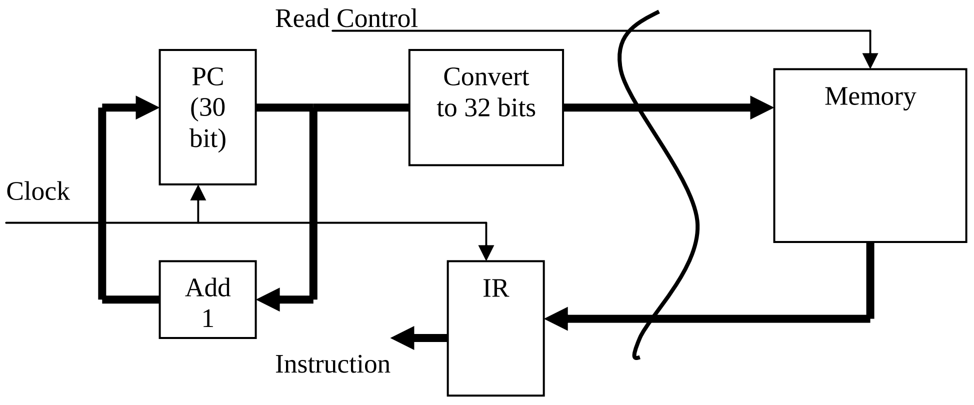
ALU in the SDLX is used to perform the operations on the input operands and to produce the output. A 32-bit ALU is provided in the SDLX. Its design is not given in this handout but the following functions are implemented in the ALU. The corresponding function codes are indicated by a symbolic name and are shown underlined in parentheses.

- Addition (<u>ADD</u>)
- Addition (<u>ADD4</u>). The second number is multiplied by 4 and added in the first number. This operation will be needed in JR and JALR instructions.
- Subtraction (<u>SUB</u>)
- Left shift of the first operand by a count provided by the second operand (only five least significant bits of the second operand are used) (<u>LS</u>)
- Arithmetic right shift (<u>RSA</u>)
- Logical right shift (<u>RSL</u>)
- Bit wise AND (<u>AND</u>), Bit wise OR (<u>OR</u>), Bit wise XOR (<u>XOR</u>)
- Signed greater than comparison of two operands (output is all 1's if first operand is greater than the second one) (<u>SGT</u>)
- Unsigned greater than comparison (<u>UGT</u>)
- Signed less than comparison (<u>SLT</u>)
- Unsigned less than comparison (<u>ULT</u>)
- Signed less than or equal to comparison (*aka* signed NOT greater than comparison) (<u>SLE</u>)
- Unsigned less than or equal to comparison (<u>ULE</u>)
- Signed greater than or equal to comparison (<u>SGE</u>)
- Unsigned greater than or equal to comparison (<u>UGE</u>)
- Data mixing (output 32 bits are composed from the two operands. The lower order 16 bits are taken from the first operand and the higher order 16 bits are taken from the second operand) (<u>DM</u>)

All the ALU functions can be implemented using 6-bit function code. (For the functions listed here even 5-bit control code is enough. However, as in DLX, we have allocated 6 bits for this). Depending upon the instructions, appropriate codes are presented to the control input of the ALU.

Reading Instructions from the Memory

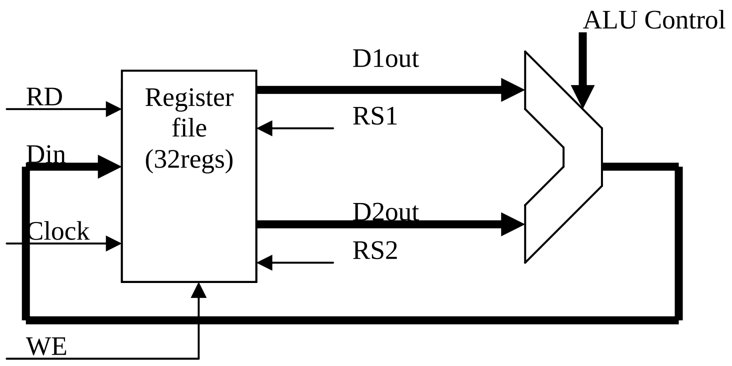
The instructions are read from the memory location whose address is determined from the 30-bit program counter (PC) register. The lower order two bits of the address are taken as 0 (to make it aligned at 4 byte boundaries). The upper order 30 bits of the address are taken from the PC. At the end of the clock, PC register is updated to the new value of the PC (i.e. adding 1 to it so that the next address is $+4). Also at the same time, the 32-bit instruction read from the memory is stored in the instruction register (IR). Various controls to the datapath elements are then derived from the contents of the instruction register. The following circuit implements this operation.



It may be noted that there is one clock delay between the instruction made available to the processor (or loaded in the IR register) and the address provided to the memory.

Datapath interconnection for the R type triadic instructions.

The R type triadic instructions need to read two registers and write one register after computing its value as specified by the instruction. The following datapath is needed for implementing the R type triadic instructions.



For the instructions to work, WE should be 1 (All of R type triadic instructions update the register). RD should be same as IR[15:11], RS1 should be same as IR[25:21], RS2 should be same as IR[20:16]. ALU control are derived from IR[5:0]. The operation of this circuit is shown with an example. Consider the following two instructions in a program. The time line of the entire processor datapath discussed till now shall be as follows. In the description below, the PC contents are shown as the address of the memory. In reality it should be address divided by 4.

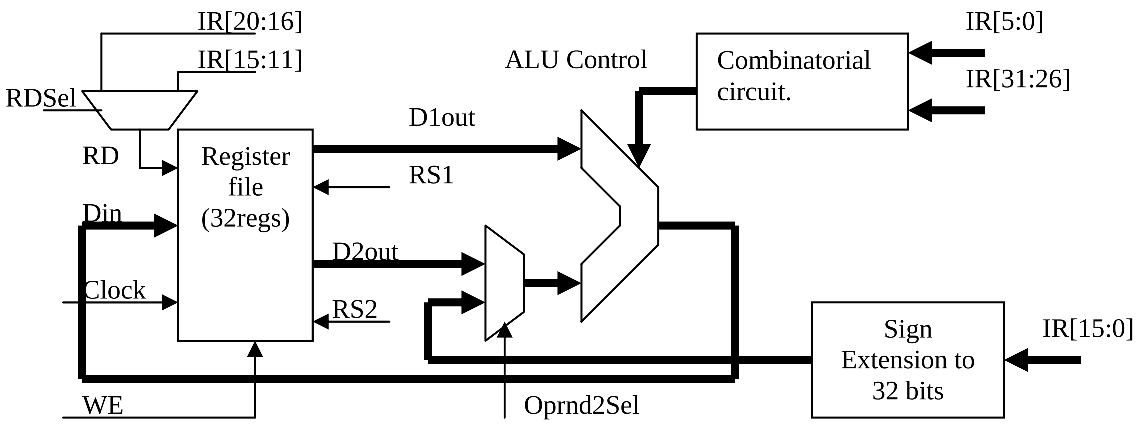
```
100:   ADD   R2, R3, R4        // (R4 = R3 + R2)
104:   AND   R4, R3, R4        // (R4 = R4 & R3)
```

Clock Cycle	PC contents	IR contents	Remarks
i	100	XXX	instruction fetched in the previous cycle is executed.
i+1	104	ADD inst	During the clock cycle 1, ADD operation is done and at the end of cycle R4 is updated. ALU control in this cycle is <u>ADD</u>.
i+2	108	AND inst	During this clock R4, R3 are read, AND operation is done and at the end, R4 is updated. ALU control is <u>AND</u>.

Datapath interconnection for the non-load-store R-I type dyadic instructions.

The non-load-store instructions of this type are essentially the ALU instructions. The datapath for the execution of these instructions is modified as shown in the next diagram.

The various controls have to be as follows. RD is same as IR[20:16]. RS1 is same as IR[25:21]. The ALU controls are derived from the Opcode field only. Note that there is a need for a multiplexer that selects one of the two 32-bit inputs to the second port of the ALU. A control called Oprnd2Sel is used to select the input. The two values that are possible for this control are D2outSel and ImmSel. One of these can be taken as 0 and the other one as 1.



There is a conflict in this datapath at the RD, RS2 and ALU controls. These can not be the same as the ones used in the R type triadic instructions. We need to resolve that before moving ahead and implementing more instructions. The RD field for these instructions is taken from IR[20:16] (as opposed to IR[15:11] in R-type triadic instructions). We resolve that by putting a multiplexer with a RDSel control. The RDSel has two possible values, IR20 and IR15. One of these is taken as 0 and the other one is 1. The RS2 field in this datapath is not really used and therefore there is no real concern to handle it. However, the ALU control can not be derived with just IR[5:0] as in R type triadic instructions. We need to put a combinatorial circuit that provides the ALU functions from Opcode (i.e. IR[31:26]) and func code (i.e. IR[5:0]) as shown in the diagram.

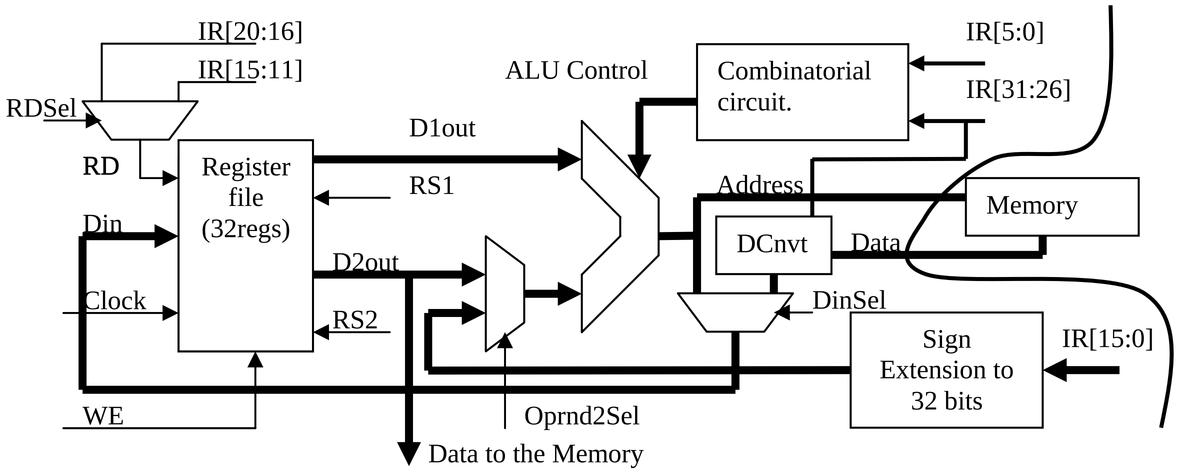
Datapath interconnection for the load-store instructions.

For the load store instructions, the ALU can be used to compute the address of the memory. Currently we assume that the load store instructions provide the address that is aligned with respect to the size of the data. Any address can be given to load a byte from the memory. However an address must be a multiple of two to load a halfword from the memory. Similarly the address must be a multiple of four to load a word from the memory. In case, these conditions are not met, the processor generates an exception and the exception handlers handle such conditions. We shall not see this currently and describe the process later during our discussion on the exception handling.

The datapath provided earlier can compute the effective address by providing the immediate constant (sign extended) to one input of the ALU and the contents of the register RS1 on the other input of the ALU. However the output of the ALU should not be stored in the register file, rather should be used as an address to the memory. For load instructions the data read from the memory should be stored in the register file. In case of the store instructions, the contents of the register RS2 should be used to store in the memory (after applying appropriate size conversions).

The modified datapath is shown in the next diagram. Notice the inclusion of one box called DCnvt and one more multiplexer that selects between the output from the memory and that from the ALU. The DCnvt box is used to convert the data read from the memory to 32 bits. For example, in case of LBU instruction, it would pad the 8 bit data with 24 zeros to the left before passing it to the register file. Similarly in case of LB, it would sign extend the 8-bit data to 32-bit. The DCnvt box is implemented by a combinatorial circuit with data read from the memory and instruction opcode (IR[31:26]) as input.

The additional multiplexer selects between the output from the memory and the output from the ALU. For load instructions, the output from the memory is chosen to appear at the Din port of the register file. For all other instructions, the output from the ALU is chosen to appear at the register file. A control signal called DinSel is used for this that can have one of the two values, MemD and ALUD.



For all the instructions discussed till now, WE control has to be 1 as all instructions generate an output that has to be written in the register file. For store instruction however, there is a difference. The store instruction does not write any value in the register file. Rather it uses the value of the RD register and writes that to the memory. For this, datapath requires minimal changes. WE control is set to 0 for store instructions and data read from the register file is presented to the memory data input.

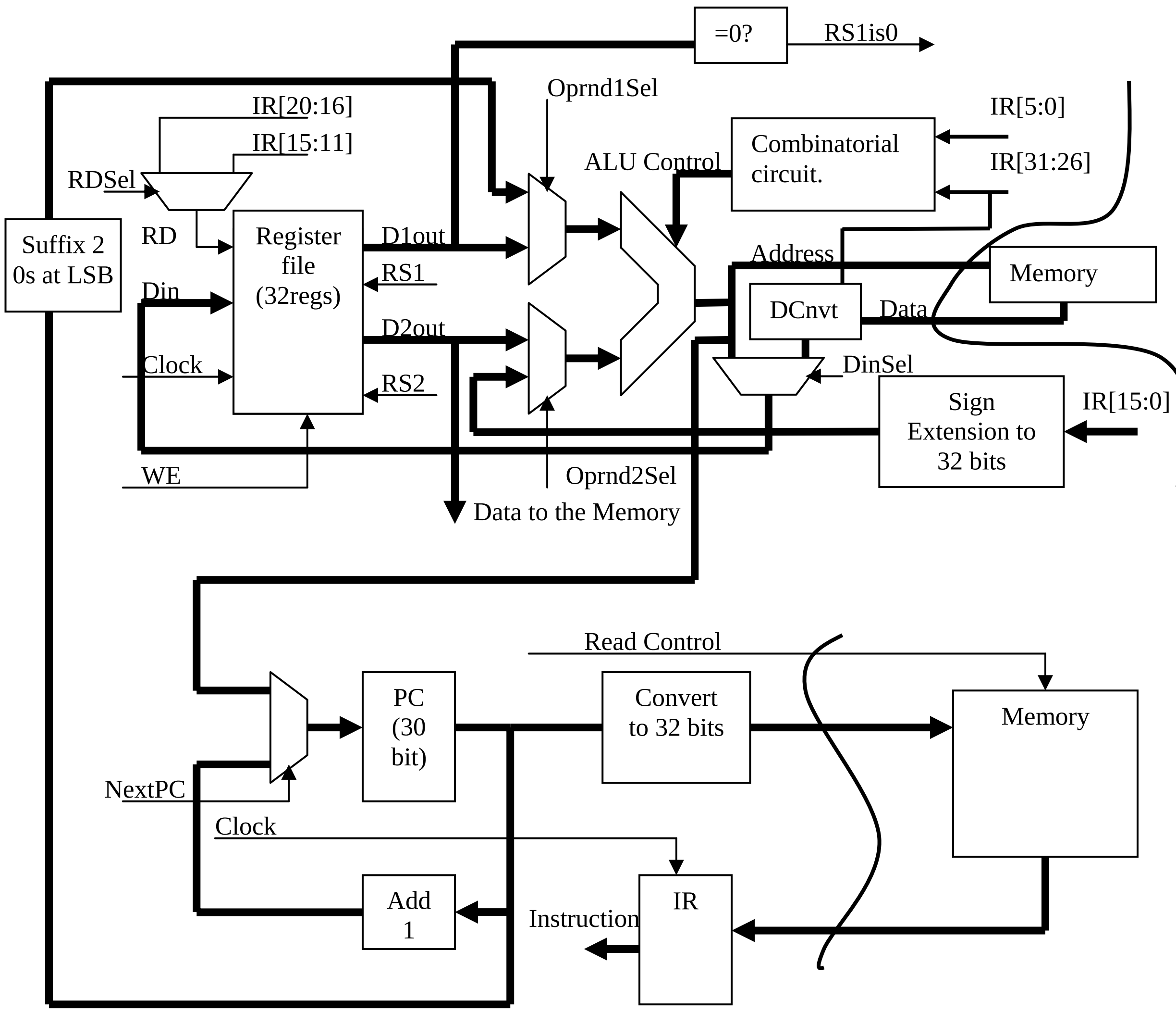
Depending upon the size of the data, various controls have to be used for reading and writing the data. For this we use four controls, EN0, EN1, EN2, and EN3. Consider a memory organization as 4 byte wide where write to each individual byte is separately controlled. Each of these controls enables data read or write in one of these bytes of the memory. Given the two least significant bits of the generated address and the size of the memory to read or write, these controls can be generated as given in the following table. A simple combinatorial logic can be used to generate these signals and is not shown.

Address	Size	EN0	EN1	EN2	EN3
00	Byte	1	0	0	0
	Halfword	1	1	0	0
	Word	1	1	1	1
01	Byte	0	1	0	0
	Halfword	Unaligned Access			
	Word	Unaligned Access			
10	Byte	0	0	1	0
	Halfword	0	0	1	1
	Word	Unaligned Access			
11	Byte	0	0	0	1
	Halfword	Unaligned Access			
	Word	Unaligned Access			

Datepath interconnection for the BNEZ and BEQZ instructions.

The BNEZ and BEQZ instructions compare the register value read on D1out port of the register file with 0. This comparison is performed by a simple 32-bit OR gate (at least conceptually). The output of this gate is 0 only when all the input bits are 0. Thus this output is used to decide if the BNEZ or BEQZ instructions should change the value of the PC register or not.

The datapath for this operation is modified as follows. Notice the addition of three blocks in this circuit. A '=0?' block is used to compare the register RS1 being 0 or not. It generates a signal called RS1is0 that is 1 if RS1 is 0.



The signal RS1is0 is used to decide if the new value of the PC should be loaded or not. The regular ALU is used to compute the new value of the PC. For this, the offset is presented on the second input of the ALU. To present the value of the PC register on the first input, another multiplexer is used on the first input of the ALU. This multiplexer is controlled by a signal called Oprnd1Sel. Two values that are possible for this signal are PCSel and RS1Sel one of which is 0 while the other one is 1. The input to the PC register is selected through a multiplexer with control being NextPC. This multiplexer selects either PC+1 (for non-branch instructions or for the not taken branches), or the value computed by the ALU.

Note that during the execution of BNEZ or BEQZ instruction, PC register contains a value corresponding to the address of the next instruction ($+4). Further note that the 30-bit PC register is used to provide the value to the ALU operand. We assume that in the 32-bit value, upper two bits are unused. Thus this instruction loads the new value in the PC as $+4+4*offset where $ is the address of the BNEZ/BEQZ instruction.

This datapath explains the delayed branch also. Consider an example of the following code.

```
100: ADD    R1, R2, R4 // R4 = R1+R2
104: BNEZ   R1, 200    // Branch to 200 if R1!=0. Immediate constant in
                       // the instruction is 23.
108: SUB    R5, R6, R7 // Some instruction after the BNEZ
.
.
.
200: AND    R1, R2, R3 // Instruction at the target location.
.
.
```

The time line of the execution of this instruction stream is given below. Again for clarity PC is shown to contain the address of the instruction (rather than the address divided by 4).

Clock Cycle	PC contents	IR Contents	Remarks
0	100	XXX	An instruction fetched earlier
1	104	ADD	At the end of the cycle R4 is updated.
2	108	BNEZ	At the end PC is updated to 200
3	200	SUB	SUB instruction gets executed
4	204	AND	Instruction from the target

As shown in the time line of the execution, SUB instruction gets executed. This concept is known as delayed branch and this phenomenon is shown by all SDLX instructions that modify the PC register.

One last thing, since these instructions do not modify any register in the register file, WE signal is set to 0.

Datapath interconnection for the JR and JALR instructions.

The JR and JALR instructions are now simple to implement. The value of the RS1 register is added with the four times the immediate constant in the instruction (Note the ADD4 operation on the ALU). The least order two bits of the output are dropped (or the result is shifted by two bits to the right). By choosing appropriate controls, RS1 and immediate constant can be sent to the ALU ports. The output of the ALU is used to load the PC register.

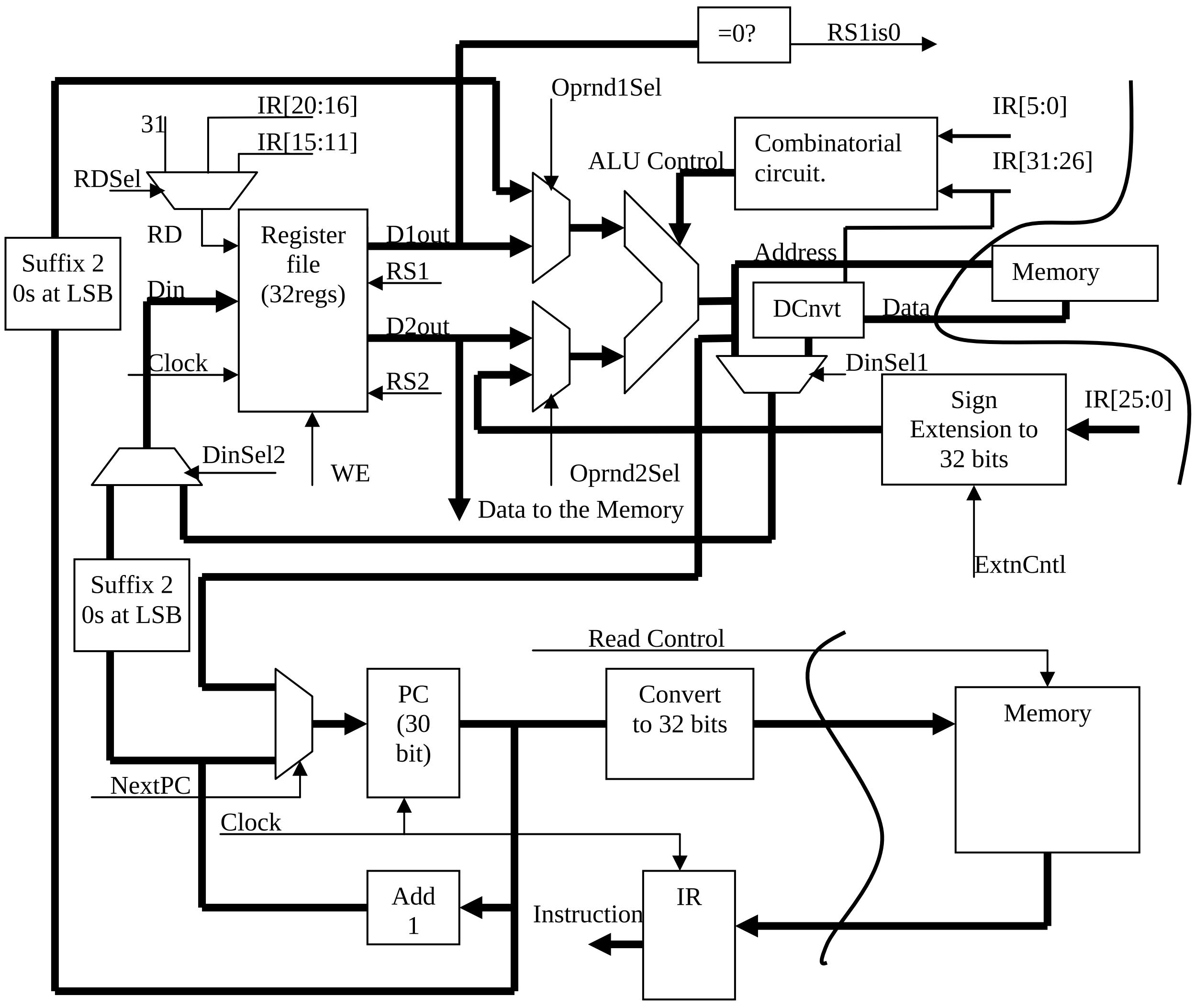
In JR instruction, no value of the register gets updated. However in case of the JALR instruction, $+8 is stored in the R31, the link register. Thus for the JR instruction WE is set to 0 while in case of JALR, some modifications are needed. RD field should carry 31 (all 1s). Din of the register file should carry 4*(PC+1) (note that the PC already contains a value corresponding to the $+4). Multiplication by 4 is achieved by a simple shift and putting two least significant bits as 0. Appropriate changes in the datapath are shown in the next diagram. DinSel2 control signal selects either the ALU/Memory output or 4*(PC+1) to the Din input of the register file. The possible values are ALUMemOut or PCout.

Datapath interconnection for the J and JAL instructions.

The only change needed in the datapath to implement J and JAL instructions is to be able to handle 26 bit immediate constant. In order to implement this, all that is needed is a more sophisticated sign extension mechanism for the immediate data. We will also need an ExtnCntl that can have two values Imm16 and Imm26 to indicate what kind of sign extension is needed (16-bit or 26-bit). Any one of these can be taken as 0 and the other one can be taken as 1.

On the ALU inputs, we need to provide the contents of the PC register that can be chosen by appropriate control signals. For the J instruction, WE should be 0 while for the JAL instruction WE should be 1, 31 should be chosen on the RD and PC+1 should be chosen on the Din of the register file. The next diagram shows all the changes needed to implement this.

Note that the datapath shown in this diagram is enough to implement all the instructions of the SDLX processor.



Final design of the SDLX Datapath

Design of the Controller for the SDLX

The SDLX processor controller generates the appropriate control signals for the processor and will be discussed more in details later.