Experiment 221- 1: A simple microprocessor

Aims

1 To produce a large, largely self-contained, circuit with complex state dependencies illustrating the benefits of fully synchronous design.

2 To gain experience at system level design – "putting it all together"

3 To gain some (limited) experience at group working.

Introduction

There is only one experiment for CS221. You have 5 sessions of 3 hours each to complete it. This is the first time that a practical of this complexity has been run in the 2nd year Engineering Lab, so that it is possible that the targets set may be revised during the semester in the light of experience. However, it is believed that the expectations are reasonable and that all students should be capable of finishing it in the allotted time.

You will normally work in teams of two, but in contrast to the practice in the 1st year Engineering Laboratory, each team member will be responsible for only part of the design and team members will be allocated a workstation each for their scheduled lab sessions. If it is necessary to form groups of three, the design will still be split into two parts, the design of one part being duplicated by one of the members.

The success of the project will depend on both team members completing their part of the design successfully. You are responsible for your own project management; ensure that your partner keeps to the agreed goals – however the progress of the project will be monitored by laboratory staff and if you run into problems you should inform the laboratory organiser as soon as possible.

The experiment will use the Xilinx gate array and the Viewlogic schematic capture and simulation tools that were used in the first year. The board to which the design is downloaded is more complex than the one used last year. It possesses RAM, ROM, & EPROM in addition to the facilities found on the 1st year lab board. Provided that the correct library is used, circuits that you designed in CS121/122 should still function on this board. Also provided on the board are various analogue components not required for this design. Details of the libraries available and support programs are given in an appendix.

In this practical assignment, you will design and commission a small microprocessor. You are expected to write simulation programs that test individual modules and to provide a small set of test programs. Your report should include a commentary on your design and comprehensive documentation. The interface between your part of the design and your partner's should be clearly documented.

> **This is not an easy experiment. You *must* give some thought to the design before attending the laboratory.**

Preparation

You may choose a partner by filling in the form on the laboratory notice board. Do this as soon as possible. Before your first lab session, you should have sketched on paper an outline design of the datapaths required. Some hints on how to do this are given later in manual.

You will already have most of the necessary building blocks from preceding experiments in CS121/122. Symbols representing the various components on the board (e.g . keyboard, memory, display etc. are in the *EngLab* library. Also available are circuits, such as counters, registers etc. in the *MX3000* library. You will need to add these libraries to list of searchable libraries in your *viewdraw.ini* file. Try to decide what other funcdtions you will need and how they might be implemented. The following sections are intended to stimulate ideas rather than explain exactly how to construct the processor.

If you are short of time then implement a subset of the circuit; for example some functions can be omitted without affecting the rest of the circuit. Alternatively if you have both the time and the inclination you are welcome to extend and elaborate the design. If you do wish to change the scope of the experiment you are encouraged to discuss what is feasible with a demonstrator.

Specification

The microprocessor is a simple CISC (Complex Instruction Set Computer). It is an 8-bit processor (i.e. the data path is to be 8 bits wide) and the address bus is also 8-bit (giving a maximum address range of 256 bytes. It has two user registers which are general purpose and fully interchangeable in function and are referred to as R0 and R1. There are five classes of instructions:

1 Diadic data operations

2 Monadic data operations

3 Store

4 Conditional jump

5 Halt

Addressing Modes

The addressing modes allowed are somewhat non-orthogonal. The *effective address* of one operand may be generated in four way as shown in Table 1. A second operand must reference a register. In the case of *store* instructions, the source operand must reside in a register whereas for other instructions, the destination must be a register.

0	Register	Rn	The effective address (EA) is the specified processor register
1	Indirect	[Rn]	The EA is a memory location whose address is contained in a processor register.
2	Absolute	nn	The EA is a memory location whose address is specified by the byte following the instruction in the programme.
2	Immediate	#nn	The EA is not an address, but a literal in a byte which follows the instruction in the programme.

Table 1: Addressing Modes

Note that for the *absolute* and *immediate* addressing modes, the instruction is two bytes long

Instruction Format:

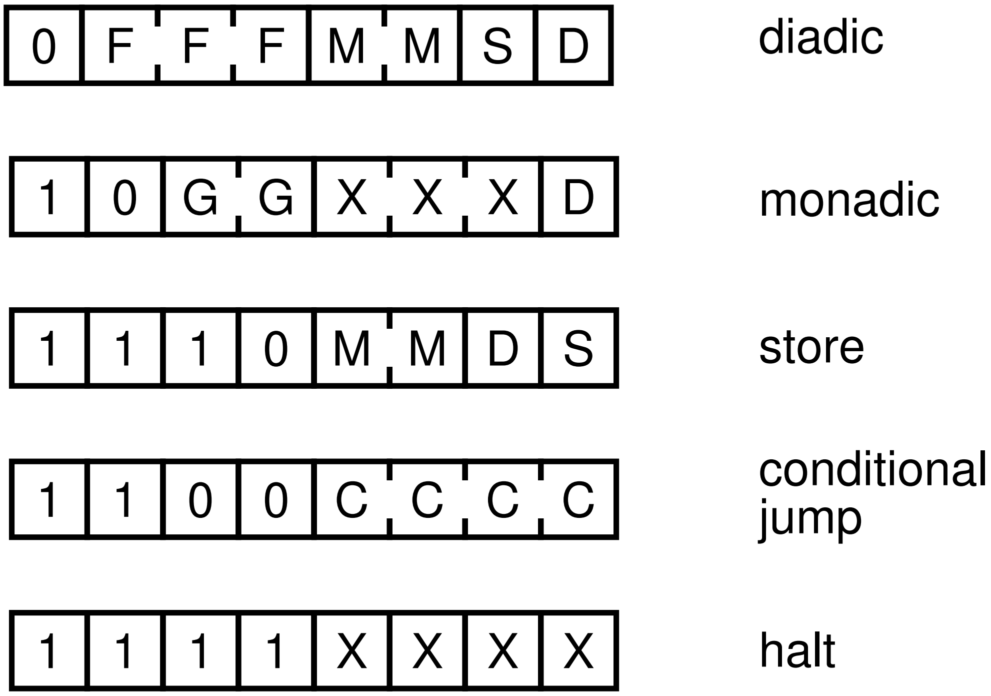

Figure 1: Instruction Format

The format of the 8-bit instruction op-code is shown in Fig. 1. The significance of the various bit fields are as follows:

FFF= diadic function code
000 = LOAD
001 = ADD
010 = SUB
011 = CMP
1XX = reserved

MM= addressing mode
00 = R
01 = [R]
10 = nn
11 = #nn

X= don't care

GG= monadic function code
00 = NOT
01 = INC
10 = DEC
11 = NEG

S= source register
0 = R0
1 = R1

D= destination register
0 = R0
1 = R1

CCCC= condition code

Note that the bit positions defining the source and destination registers are reversed in *store* operations, thus the register used in effective address calculations is always defined by bit position 1. A matrix defining the instruction set is shown in Table 2. Note that a number of the instructions can have more than one operation code or "**op. code**". This makes the instruction set inefficient, but easier to decode. The behaviour of certain op. codes has been marked as undefined or left blank. In fact the execution of any unspecified op. code could have an unknown effect; however a large block of op.codes have been left blank so that they may be used for other functions. For example the logic functions AND, OR and XOR are missing from

	MS bits:																
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
LS Bits	0000	LOAD R0,R0	ADD R0,R0	SUB R0,R0	CMP R0,R0					NOT R0	INC R0	DEC R0	NEG R0	JAL nn		undef	HALT
	0001	LOAD R0,R1	ADD R0,R1	SUB R0,R1	CMP R0,R1					NOT R1	INC R1	DEC R1	NEG R1	JNV nn		undef	HALT
	0010	LOAD R1,R0	ADD R1,R0	SUB R1,R0	CMP R1,R0					NOT R0	INC R0	DEC R0	NEG R0	JHI nn		undef	HALT
	0011	LOAD R1,R1	ADD R1,R1	SUB R1,R1	CMP R1,R1					NOT R1	INC R1	DEC R1	NEG R1	JLS nn		undef	HALT
	0100	LOAD [R0],R0	ADD [R0],R0	SUB [R0],R0	CMP [R0],R0					NOT R0	INC R0	DEC R0	NEG R0	JCC nn		STORE R0,[R0]	HALT
	0101	LOAD [R0],R1	ADD [R0],R1	SUB [R0],R1	CMP [R0],R1					NOT R1	INC R1	DEC R1	NEG R1	JCS nn		STORE R1,[R0]	HALT
	0110	LOAD, [R1],R0	ADD [R1],R0	SUB [R1],R0	CMP [R1],R0					NOT R0	INC R0	DEC R0	NEG R0	JNE nn		STORE R0,[R1]	HALT
	0111	LOAD [R1],R1	ADD [R1],R1	SUB [R1],R1	CMP [R1],R1					NOT R1	INC R1	DEC R1	NEG R1	JEQ nn		STORE R1,[R1]	HALT
	1000	LOAD nn,R0	ADD nn,R0	SUB nn,R0	CMP nn,R0					NOT R0	INC R0	DEC R0	NEG R0	JVC nn		STORE R0,nn	HALT
	1001	LOAD nn,R1	ADD nn,R1	SUB nn,R1	CMP nn,R1					NOT R1	INC R1	DEC R1	NEG R1	JVS nn		STORE R1,nn	HALT
	1010	LOAD nn,R0	ADD nn,R0	SUB nn,R0	CMP nn,R0					NOT R0	INC R0	DEC R0	NEG R0	JPL nn		STORE R0,nn	HALT
	1011	LOAD nn,R1	ADD nn,R1	SUB nn,R1	CMP nn,R1					NOT R1	INC R1	DEC R1	NEG R1	JMI nn		STORE R1,nn	HALT
	1100	LOAD #nn,R0	ADD #nn,R0	SUB #nn,R0	CMP #nn,R0					NOT R0	INC R0	DEC R0	NEG R0	JGE nn		undef	HALT
	1101	LOAD #nn,R1	ADD #nn,R1	SUB #nn,R1	CMP #nn,R1					NOT R1	INC R1	DEC R1	NEG R1	JLT nn		undef	HALT
	1110	LOAD #nn,R0	ADD #nn,R0	SUB #nn,R0	CMP #nn,R0					NOT R0	INC R0	DEC R0	NEG R0	JGT nn		undef	HALT
	1111	LOAD #nn,R1	ADD #nn,R1	SUB #nn,R1	CMP #nn,R1					NOT R1	INC R1	DEC R1	NEG R1	JLE nn		undef	HALT

Table 2: Instruction Set Matrix

this specification, and may be added as ALU functions using the same addressing mode generation logic.

The Problem

There are several problems in implementing a processor, but generally these break down into the data paths and the control logic. It is normal to define the data paths first, and then examine the necessary control for these. Generally speaking, the control logic is the more difficult design problem, although the instruction set has been chosen in an attempt to minimise this complexity somewhat.

The data paths are basically RTL blocks. The basic ‘loop’ is formed by the registers and the ALU, but other elements will also need access to these buses. For example - unless a separate incrementer is designed - it is likely that incrementing the programme counter will be done using the ALU; more on this later.

Register Bank

Although this processor is simple it does have two general purpose registers which are identical in every respect; thus the registers can be said to form a **register bank**. It may well be convenient to create a register bank as an RTL subcircuit. As the processor can perform diadic operations on the registers (e.g. adding R0 to R1) it is convenient to have two ‘read buses’ on the register bank. Each of these will have an associated address (each merely a single bit in this case) which specifies which register is output on that bus.

In addition a further 8-bit bus is required in order to write the result back to the relevant register. This must be accompanied by some means of addressing and enabling the result register. This could be a single address line and an enable line, or two separate enable signals (one for each register). This is needed because there are *three* possible operations in a given cycle:

1. Write to R0
2. Write to R1
3. No write operation

The last operation is necessary because for some operations (e.g. STORE), no register is affected.

Arithmetic Logic Unit (ALU)

For the processor as specified there is no need for the ALU to perform any logic operations; all the necessary functions can be performed arithmetically. Indeed with some pre-conditioning all the operations can be performed as additions. For example:

ADD A,B	A := A+B
SUB A,B	A := A+(-B)
LOAD A,B	A := 0 + B

Ideally the ALU should receive two operands and a **function code** and output a result and some condition code bits. All the necessary functions should be possible to code for in three bits.

The other outputs required from the ALU are the **flags** or **condition codes**. These indicate the information about the result of the function. There are normally four such bits shown in Table 3, although sometimes others are present.

Branches

The conditional jumps shown in the instruction code matrix in Table 2 use the condition codes as explained below in Table 4.

Other Registers

As well as the user’s two registers (R0 & R1) the processor will also require some other internal **state**. The most obvious such register is the **Programme Counter**[1] (usually abbreviated to **PC**), which indicates the position in memory from which instructions are fetched. It is usual (though not obligatory) for the PC to be incremented immediately after an instruction is fetched (i.e. **post-incremented**) so that during the actual execution phase it points at the next

1.some architectures use the name **Instruction Pointer (IP**)

Negative	N	Set if the result is a negative number. Simply a copy of the Most Significant Bit of the result
Zero/Equal	Z	Set if the result is zero, otherwise clear. Note that a zeo results if two equal numbers compared or subracted and may therefore be used for an equal condition
Carry/Borrow	C	A copy of the carry out of the most significant bit of the addition, inverted if the last operation was a subtraction to indicate a borrow. It indicates that the addition or subraction of two unsigned numbers has overflowed.
Overflow	V	Set if the result has overflowed the possible range of numbers represented by two's complement *signed* numbers. It is derived from the XOR of the carry into and out of the MSB.

Table 3: Condition Code Flags

Mnemonic	Explanation	Branch taken if:
JAL	Jump Always	Always
JNV	Jump Never	Never
JHI	Jump if Higher	$\overline{C}.\overline{Z}=1$
JLS	Jump if Lower or Same	$C+Z=1$
JCC	Jump if Carry Clear	$C=0$
JCS	Jump if Carry Set	$C=1$
JNE	Jump if Not Equal	$Z=0$
JEQ	Jump if Equal	$Z=1$
JVC	Jump if oVerflow Clear	$V=0$
JVS	Jump if oVerflow Set	$V=1$
JPL	Jump if Plus	$N=0$
JMI	Jump if Minus	$N=1$
JGE	Jump if Greater than or Equal to	$N.\overline{V} + \overline{N}.V = 0$
JLT	Jump if Less Than	$N.\overline{V} + \overline{N}.V = 1$
JGT	Jump if Greater Than	$N.V.\overline{Z} + \overline{N}.\overline{V}.\overline{Z}=1$
JLE	Jump if Less than or Equal to	$N.V.\overline{Z} + \overline{N}.\overline{V}.\overline{Z}=0$

Table 4: Conditional Branches

instruction. The obvious prerequisites of the PC are that it should be available for output on the address bus and also to some form of increment function (usually via the ALU). Slightly less obviously the PC must be 'loadable' with data from memory in order to execute a 'jump' instruction. Finally a means of resetting the PC must be provided to allow the instruction sequence to be restarted.

When an instruction is fetched it must be placed somewhere. This holding register is normally referred to as the **Instruction Register** (**IR**); it is generally 'invisible' to the software programmer who rarely cares what bit-pattern represents which instruction. The IR must be loadable from memory but usually its outputs are solely fed into the **instruction decoder** as part of the control logic. The IR should hold the instruction throughout its execution so that different phases may scheduled correctly.

Several other registers normally contained in a processor such as this are concerned with memory access. One of these is normally known as the **Memory Data Register** (**MDR**) and is used to hold data which is loaded from memory. This is necessary here, as can be demonstrated in a data operation using the ‘absolute’ addressing mode, where a value is loaded (from the instruction stream) and used as an address for a further load (or store). Sometimes other registers - such as a **Memory Address Register** (**MAR**) which is used to hold an address to be used in a subsequent memory access - are included in the processor architecture, but these can be eliminated here. It is recommended that as few registers as possible are used in order to reduce the control complexity.

Finally a **condition code register** or **flag register** may be employed to keep track of data necessary for conditional operations (just ‘JMP’s in this case). This retains the state of the last ALU function’s output so that it can subsequently be tested and used for control. In most processors only a subset of possible instructions alter the flags, although here all operations can set them if this makes decoding easier. However remember that it is just operation *execution* which alters these bits; they must not be changed by (for instance) the PC increment following an instruction fetch.

I/O Ports

The architecture must also contain some means of getting data in to and out of the processor. The processor generates addresses, which are output to memory, and ‘store’ operations also provide data which must be written to memory. However most of the time data is read from memory and input to the processor. Eventually the data input and output buses must be multiplexed onto a bidirectional data bus to interface with the memory device; however it is recommended that these ports are initially kept separate for convenience, especially in testing the design.

Bus Structure

Having decided on the registers and the ALU structure it is necessary to connect these with buses. In general the various register outputs must be multiplexed to reach the ALU, the address output and the data output. There are also two ‘sources’ of data (the ALU and the data input port) which must reach certain of the registers.

Note that the act of writing to registers may be done using a common bus and controlling the operation using register enable signals. If, however, two registers are to be written to in parallel then they must have separate buses to carry their separate values.

Keep the bus structure as simple as possible, bearing the various necessary operations in mind.

Processor Execution Control

There are a number of operations which must be performed during the operation of each instruction. This sequence will vary according to the instruction and addressing mode.

It is possible (with a little concurrency which influences the bus structure) to coerce the operation into four phases:

Phase 0	Instruction fetch	Address:= PC PC:= PC+1 IR:= Data-in
Phase 1	Operand fetch	Address:= PC PC:= PC+1 MDR:= Data-in

Table 5: Processor Cycles

Phase 2	Data fetch (absolute)	Address:= MDR MDR:= Data-in
	Data fetch (indirect)	Address:= Rs MDR:= Data-in
Phase 3	Execute (data op) register mode	Rd:= function(Rd,Rs)
	Execute (data op) not register mode	Rd:= function(Rd,MDR)
	Execute (jump)	PC:= MDR
	Store (absolute)	Address:= MDR Data-out:= Rs
	Store (indirect)	Address:= Rd Data-out:= Rs

Table 5: Processor Cycles

The state diagram for the sequencer following this pattern is shown in figure 2. Follow various instructions through this to see how each instruction can be sequenced. Other possibilities may be explored

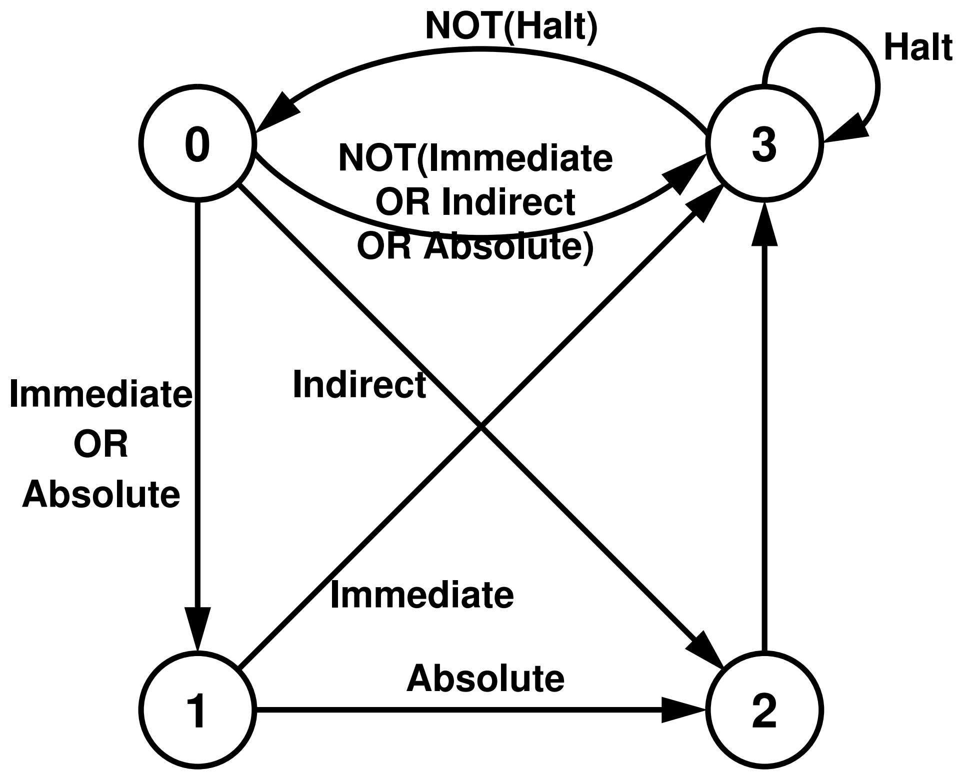

Figure 2: Possible state diagram for instruction execution

Remember that as well as a ‘bottom-up’ design method a ‘top-down’ approach can be used; it is possible to design a symbol which (initially) has no schematic and use this as a component of the top level schematic. The ‘innards’ of the symbol can be filled in later.

Processor Functionality

Implement the processor described in the specification above. The specification lists the minimum set of functions required; further operations, such as logical’ operations may be added if desired.

Project Milestones & Deliverables

1 **Team Formation:** Choose a partner – sign up on the notice in the lab as soon as possible.

2 **Datapath Design:** During your 1st lab session sketch an outline of the datapath and agree it with a demonstrator. Produce a Viewlogic schematic. Each of the major blocks should be annotated (e.g. with initials) to indicate the partner responsible for the design of that block. The schematic must be handed in by the end of the 1st lab session.

3 **Simulation Models:** Produce documented Viewsim simulation files for each major block. A demonstrator must agree that each simulation is adequate and that the module passes the simulation test. Dates for these deliverables must be agreed with a demonstrator when the datapath design is approved, but approximately 50% should be simulated by the end of your 3rd lab session nad the remainder demonstrated by the start of the final lab session.

4 **Test Programs:** A suite of test programs must be produced and shown running to a demonstrator by the end of your last scheduled lab session.

5 **Project Report:** The report must be handed in no later than two weeks after your last scheduled lab session. The report should:

- Explicitly list the modules designed by each group member and the proportion of the total work done by each partner (this figure should normally be the same for each team member; if it is not, the figure should be discussed wit the other team member(s) before submitting the report).
- Explain the design.
- Include all schematics
- Include commented listings of the test programs.
- Discuss performance issues: don’t worry if your machine is not the fastest possible architecture, but you should discuss how to improve performance if it is not.
- Optionally, discuss how the processor may be extended. For example what sequence of operations must be executed if a subroutine CALL and RETURN were to be implemented? (Do not worry if this is impossible with your particular RTL architecture)

Appendix A: How to Use the SMURpH Board

(**S**econd year **M**anchester **U**niversity **Rep**rogammable Hardware)

Compiling the Viewlogic Schematic

- Run the program **Sxmake <design name>**

Loading the Xilinx Array

- Connect the board via the RS232 connector to serial port B on the SUN workstation
- Set the program-mode-switch to “serial”.
- Press the red reset switch and the yellow program switch.
- Run the program **Sxld <design name>**

Loading the on-board RAM

- Connect the board via the RS232 connector to serial port B on the SUN workstation
- Set the program-mode-switch to “eprom”.
- Press the red reset switch and the the yellow program switch. The Xilinx gate array is then configured from a design held in EPROM to act as a serial interface to the on-board RAM. Note that any existing design in the Xilinx array will be overwritten and need to be reloaded afterwards.
- Run the program **Sdnld <file name>** to load a file in SMURpH hex format into RAM. The are a number of options available. These are detailed in the manual pages available online.
- Data may be uploaded from RAM using the command **Supld [filename]**

Hex Format

A simple format is used for tranferring hex data to and from the board. A file starts with the start address preceded by the character ‘.’ or ‘:’ which denote that the next 4 hex digits represent an address and set the loader format to 8 or 16 bits wide. A cooment is preceded by ‘|’:

```
.0000
34 54 23 34 | Download 0x34 0x54 0x23 0x34 to adresses 0, 1, 2, 3
:0034
4534 5465 2DDD | Download 0x4534 5465 0x22DD to addresses 0x34 0x35 0x36
```

ViewSim File Format

- Files can be converted from an 8 bit SMURpH hex file to the input format of ViewSim’s *loadm* command bu the utility **hex2mem <filename>**

Appendix B: The MX3000 Library

General Macros

GADD	Adder
GCOMP	Compare
GLTGT	2-Bit Less Than / Greater Than Comparator
GMUX	2-to-1 Mux

GXTL	Crystal Oscillator

Latch Macros

LD	Data Latch
LDRD	Data Latch with ResetDir
LDSD	Data Latch with SetDir
LRS	Set-Reset Latch with Reset Dominant

Flip-Flop Macros

FDR	D Flip-Flop with Reset
FDRD	D Flip-Flop with ResetDir
FDS	D Flip-Flop with Set
FDC	D Flip-Flop with ClkEna
FDCR	D Flip-Flop with ClkEna, Reset
FDCRD	D Flip-Flop with ClkEna, ResetDir
FDCS	D Flip-Flop with ClkEna, Set
FDM	D Flip-Flop with 2-Input Data Mux
FDMR	D Flip-Flop with 2-Input Data Mux, Reset
FDMS	D Flip-Flop with 2-Input Data Mux, Set
FDMRD	D Flip-Flop with 2-Input Data Mux, ResetDir
FSR	Set-Reset Flip-Flop with Set Dominant
FRS	Set-Reset Flip-Flop with Reset Dominant
FJK	J-K Flip-Flop
FJKS	J-K Flip-Flop with Set
FJKRD	J-K Flip-Flop with ResetDir
FTO	Self Toggle Flip-Flop
FTOR	Self Toggle Flip-Flop with Reset
FT	Toggle Flip-Flop
FTE	Toggle Flip-Flop with ParEna
FTPRD	Toggle Flip-Flop with ParEna, ResetDir
FTR	Toggle Flip-Flop with Reset
FTS	Toggle Flip-Flop with Set
FTORD	Self-Toggle Flip-Flop with ResetDir
FTRD	Toggle Flip-Flop with ResetDir

Decoder Macros

D2A	1 -of-4 Decoder
D2-4E	1-of-4 Decoder with Ena
74-139	1-of-4 Single Decoder with EnaL, Low Output
D3-8	1-of-8 Decoder
D3-8E	1-of-8 Decoder with Ena
74-138	1-of-8 Decoder with Enables, Low Output
74-42	1-of-lO Decoder with Low Output

Multiplexer Macros

M3-1	3-to-1 Mux
M3-1E	3-to-1 Mux with Ena
M4-1	4-to-1 Mux
M4-IE	4-to-1 Mux with Ena
M4-2	4-to-2 Mux
M8-1	8-to-1 Mux

M8-1E	8-to-1 Mux with Ena
74-151	8-to-1 Mux with EnaL, Complementary Outputs
74-152	8-to-1 Mux with Low Output
74-352	4-to-1 Mux with EnaL, Low Output

Register Macros

RD4	4-Bit Data Register
RD4RD	4-Bit Data Register
RD8	8-Bit Data Register
RD8CR	8-Bit Data Register with ClkEna, Reset
RD8RD	8-Bit Data Register with ResetDir
RS4	4-Bit Shift Register
RS4C	4-Bit Shift Register with ClkEna
RS4CR	4-Bit Shift Register with ClkEna, Reset
RS4CRD	4-Bit Shift Register with ClkEna, ResetDir
RS4RD	4-Bit Shift Register with ResetDir
74-195	4-Bit Serial to Parallel Shift Register with ParEna, MRLow
74-194	4-Bit Bi-Directional ShiftRegister with ClIlEna, ParEna, MIRLow
RS8	8-Bit Shift Register
RS8C	8-Bit Shift Register with ClkEna
RS8CR	8-Bit Shift Register with CIkEna, Reset
RS8CRD	8-Bit Shift Register with CIkEna, ResetDir
RS8PR	8-Bit Shift Register with ParEna, Reset
RS8R	8-Bit Shift Register with Reset
RS8RD	8-Bit Shift Register with ResetDir
74-164	8-Bit Serial to Parallel Shift Register with MRLow

Counter Macros

C2BCR	1-Bit Binary Counter with CIkEna, Reset
C2BCP	1-Bit Binary Counter with CIkEna, ParEna
C2BCRD	1-Bit Binary Counter with CIkEna, ResetDir
C2BCPRD	1-Bit Binary Counter with CIkEna, ParEna, ResetDir
C2BP	1-Bit Binary Counter with ParEna
C2BR	1-Bit Binary Counter with Reset
C2BRD	1-Bit Binary Counter with ResetDirfp
C4BCP	2-Bit Binary Counter with ClkEna, ParEna
C4BCR	2-Bit Binary Counter with CIkEna, Reset
C4BCPRD	2-Bit Binary Counter with ParEna, ResetDir
C4JX	2-Bit Expandable Johnson Counter
C4JXC	2-Bit Expandable Johnson Counter with CIkEna
C4JXCR	2-Bit Expandable Johnson Counter with CIkEna, Reset
C4JXCRD	2-Bit Expandable Johnson Counter with ClkEna, ResetDir
C4JXRD	2-Bit Expandable Johnson Counter with ResetDir
C6JCR	3-Bit Johnson Counter with CIkEna, Reset
C8BCP	3-Bit Binary Counter with CIkEna, ParEna
C8BCR	3-Bit Binary Counter with CIkEna, Reset
C8BCRD	3-Bit Binary Counter with CIkEna, ResetDir
C8BCPRD	3-Bit Binary Counter with CIkEna, ParEna, ResetDir
C8JCR	4-Bit Johnson Counter with CIkEna, Reset
C10BCRD	4-Bit BCD Counter with CIkEna, Dir

C10BCPRD	4-Bit BCD Counter with CIkEna, ParEna, ResetDir
C10BPRD	4-Bit BCD Counter with ParEna, ResetDir
74-160	4-Bit BCD Counter with CIkEna, ParEna, ResetDir
74-162	4-Bit BCD Counter with CIkEna, ParEnaL, ResetL
C10JCR	5-Bit Johnson Counter with Clkena, Reset
C12JCR	6-Bit Johnson Counter with Clkena, Reset
C16BARD	4-Bit Binary Ripple Counter with ResetDir
C16BCRD	4-Bit Binary Counter with CIkEna, ResetDir
C16BCP	4-Bit Binary Counter with CIkEna, ParEna
C16BCPR	4-Bit Binary Counter with CIkEna, ParEna, Reset
C16BCPRD	4-Bit Binary Counter with CIkEna, ParEna, ResetDir
74-161	4-Bit Binary Counter with CIkEna, ParEnaL, MIRLOw
74-163	4-Bit Binary Counter with CIkEna, ParEnaL, ResetL
C16BPRD	4-Bit Binary Counter with ParEna, Reset, ResetDir
C16BUDRD	4-Bit Binary Up-Down Counter with ParEna, ResetDir
C16JCR	8-Bit Johnson Counter with CIkEna, Reset
C256BCP	8-Bit Binary Counter with CIkEna, ParEna
C256BCPR	8-Bit Binary Counter with CIkEna, ParEna, Reset
C256BCR	8-Bit Binary Counter with CIkEna, Reset
C256BCRD	8-Bit Binary Counter with CIkEna, ResetDi
C256FCRD	8-BitModulo 256 Fecdback Shift Register with CIkEna, ResetDir

Experiment 221-1 Record

Name:

Datapath Design

Approved by: Date **Late**: Yes/No

Simulation Models (1st 50%)

List of Blocks to be simulated

List Approved by: Date
Demonstrated to: Date: **Late**: Yes/No:

Simulation Models (2nd 50%)

List of Blocks to be simulated

List Approved by: Date
Demonstrated to: Date: **Late**: Yes/No:

Test Programs

Demonstrated to: Date: **Late**: Yes/No:

Submit this page with your report