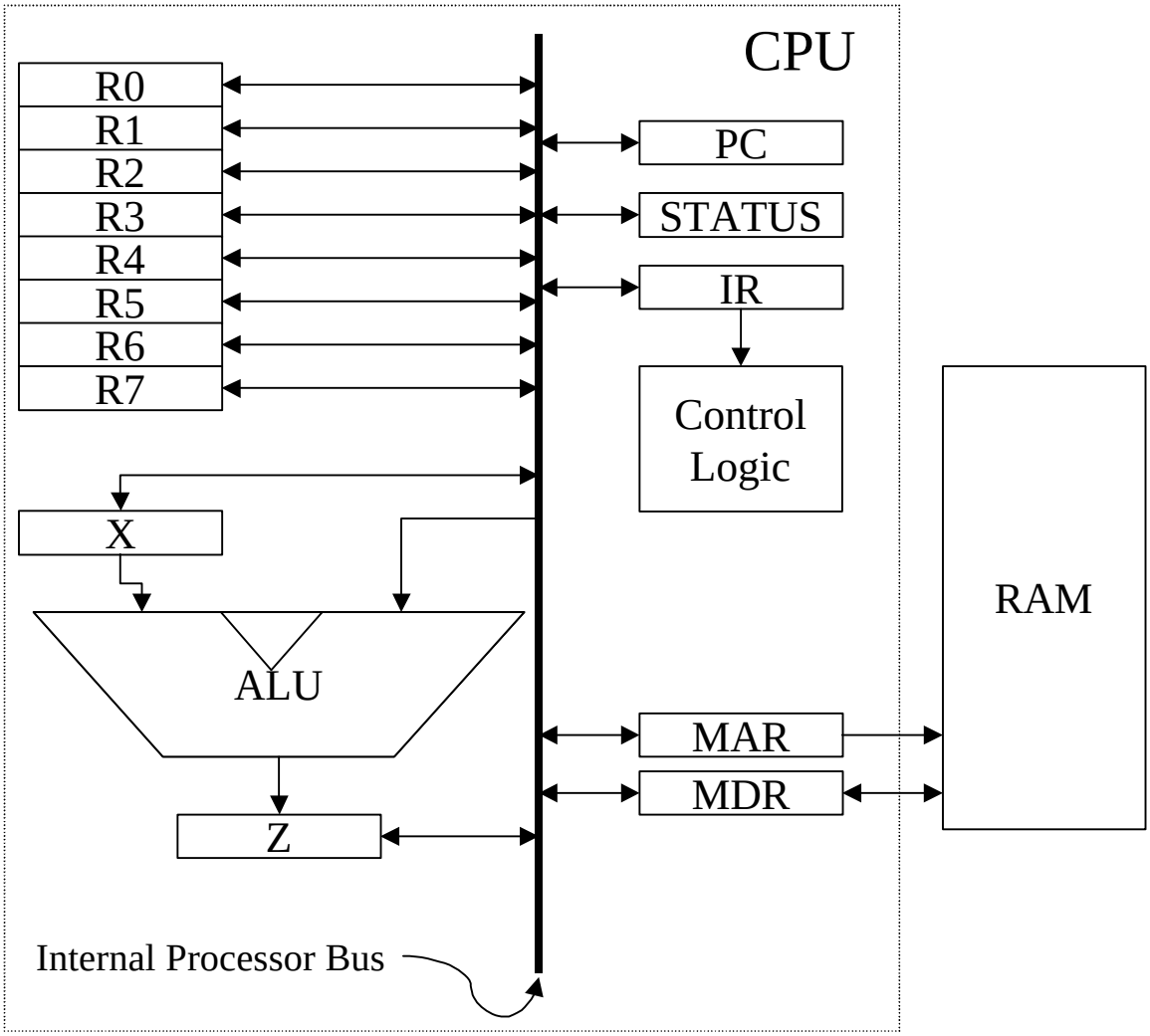
Fall 2003 – CSE 207 Digital Design Project #4

Background

Microprocessors are increasingly common in every day devices. Desktop computers use powerful general-purpose processors while wristwatches use fairly simple application specific processors. These processors differ in speed as well as their internal architecture. Advanced processors support features like massively parallel instruction execution, pipelines, caching, floating-point units, etc. Simple processors support a small set of instructions, have fewer registers, smaller data paths, generally no integrated peripheral support, etc. Common to virtually all processors are features like a fetch-execute cycle, internal data bus, external memory interface, registers, arithmetic logical units, etc. The goal of this project is to produce a simple processor that supports the functions necessary to run the programs required for the current project. These are provided below in the requirements section.

Architecture Overview



Internal Processor Bus

All of the internal components connect through the internal processor bus. This is a 16-bit bus that is used to move data around inside the processor. The control unit must only activate the output enable on one register at

a time to prevent conflicts on the bus. Any register can output a 16-bit value onto the bus and any register can read a 16-bit value off of the bus.

General Purpose Registers (R0..R7)

The general-purpose registers R0 through R7 are used for storing intermediate results. These registers have an input enable, output enable, and clock control signals. Each register is 16-bits wide. When the processor is reset these registers are in an unknown state.

Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a combinational circuit that allows one of several functions to be performed on the numbers that are input. The X and Z registers are used with the ALU to allow arbitrary values to be processed. A value can be loaded into X on one cycle and the value for Y can be placed on the bus in the next cycle. After the ALU settles, the value can be held in Z and then transferred over the bus to an arbitrary location. The ALU functions will be those in the 74x181. Since that is a 4-bit ALU, and our processor has a 16-bit ALU, four of these devices will need to be cascaded and packaged into a module. The X and Z registers have the same control lines that general-purpose registers have. Upon reset these registers are in an unknown state.

Program Counter (PC)

The program counter is a special purpose register that is a pointer to the next instruction to execute. Upon reset, this register is initialized to zero. This register has all the same control lines that a general-purpose register has plus a clear line and an increment line.

Instruction Register (IR)

This 16-bit register stores the current instruction to be executed. The Control Logic uses the current instruction to determine what internal steps need to be taken to perform the desired instruction. This is described in more detail in the micro-code section. Upon reset, this register is initialized to zero. This register has all the same control lines that a general-purpose register has plus a clear line. The output of this register does not connect to the internal bus, rather it connects directly to the control logic and always has its output enabled.

Status Registers (STATUS)

Sometimes called the flags register, this register stores the results of certain ALU operations. For example, if the result of an addition operation is zero the Z flag is set in the status register. This register is implemented as several independent D-FFs; different instructions change some flags but leave others alone. Flags that will be implemented are (Z)ero and (C)arry. Additional flags that are commonly found in processors include (S)ign and (O)verflow. See the instruction summary for information about which operations modify which flags (notice S and O are included in the summary even though you will not implement them since you do not have jump instructions that use them).

External Memory Address Register (MAR)

Since the processor only has 8 bytes of general-purpose memory, additional random access memory (RAM) is necessary to store programs and data. With only 16-bits the maximum size of memory is 65,536 bytes (64k). The MAR register has the same control lines that general-purpose registers have. Upon reset this register is in an unknown state. This register has the extra feature of always outputting its stored value to the external memory regardless of the output enable signal. Output enable only controls the flow of data from the MAR onto the internal processor bus.

External Memory Data Register (MDR)

To move data between main memory and the CPU core, the MDR serves as a buffer. External RAM is slower than the registers in the CPU and requires control lines to be toggled to read or write data. The MDR is a bi-directional register that has access to the high-speed internal processor bus and the low-speed external RAM data bus. Once the MAR register is set, the read and write control lines can be used to move data to or from the MDR and main memory. This is described in more detail in the micro-code section. MDR register has the same control lines that general-purpose registers have in addition to a memory bus input enable control line and a memory bus output enable control line. Upon reset this register is in an unknown state.

Micro-code

All of the above registers have control lines associated with them. To perform an instruction, different lines need to be switched on and off in a prescribed sequence. This sequence is sometimes referred to as micro-code. The notation used within this document called Register Transfer and is used to describe the movements of data between registers. For example, the fetch-execute sequence required to start processing every instruction can be documented as follows

Step	Register Transfer	Meaning
1	MAR ← PC	Move the address of the next instruction into the MAR
2	Read, Inc PC	Assert the Read line on external memory and point to the next instruction by incrementing the PC
3	MDR ← RAM(MAR)	Load the value pointed to by MAR to the MDR
4	IR ← MDR	Move the instruction from the MDR to the IR for execution
5	(execute sequence)	Based on the instruction perform additional micro-code steps

How do the above steps get implemented? To perform step 1, the PC output enable must be set along with the MAR input enable. On the next clock pulse, the contents of PC will be stored in the MAR. Step 2 sets the Read line going from the control unit to external memory. This may seem silly since the read line is always going to be set in this implementation, but it allows RAM time to locate the data value and put it on the memory bus. Remember, RAM is slower than the internal registers of a processor. Step 3 enables the memory bus input to the MDR.

These control lines and steps can be represented as a state machine. One method of implementing the state machine is to create a large look-up table. For the fetch sequence above, the table will look something like

Step	/PC_OE	PC_INC	MAR_IE	MDR_MIE	/RAM_OE	/MDR_OE	IR_IE
1	0	0	1	0	0	1	0
2	1	1	0	0	0	1	0
3	1	0	0	1	0	1	0
4	1	0	0	0	0	0	1

Notice, that RAM generally has its output enabled (read) unless data will be written into it. In that case, turn off the output enable, set up the data in the MDR and assert it on the bus, then enable write and disable write before removing the data from the bus. This will ensure that data is properly moved into RAM.

Testing Requirements

In addition to incremental testing of the pieces of your processor, you should execute the test program provided below. This is presented in 'C' to set it in context followed by a potential compiled assembly listing. Lastly, this program has been converted from assembly to the binary representation that will be used by your microprocessor. This program is stored in Intel HEX format so that it may be read into the RAM module in your circuit and executed by pressing reset on your processor. Your TA will test and verify your circuit using additional programs that will *not* be provided. You are encouraged to design very simple (1 or 2 instruction) programs so that you can verify the functionality of each instruction you implement. This only needs to be done at the binary level. You are NOT expected to understand how to write in 'C' nor how to compile to assembly. You are responsible for being able to test your circuit by creating binary files and testing them in Logic Works.

Program A – 'C'

```
BYTE A[] = { 7, 3, 8 };
BYTE B[] = { 9, 2, 6 };

int main(int argc, char *argv[])
{
     BYTE i;
     BYTE sum;

     sum = 0;
     for (i=0 i>sizeof(A); i++)
     {
          sum += A[i] + B[i];
     }
}
```

Program A – Assembly

```
ORG 0x8000
A DW 7, 3, 8          ; The A array
B DW 9, 2, 6          ; The B array
Sum DW ?              ; Sum of A and B lists

ORG 0x0000
     mov R0, 0x8000 ; R0 points to the A array
     mov R1, 0x8003 ; R1 points to the B array
     mov R2, 0x0000 ; Clear the sum variable
     mov R3, 0x03   ; R3 is our iteration counter

I_Loop:
     mov R4, [R0]   ; Get the next point in A
     add R2, R4     ; Add this point to the sum
     mov R4, [R1]   ; Get the next point in B
     add R2, R4     ; Add this point to the sum
     inc R0         ; Point to the next value in A
     inc R1         ; Point to the next value in B
     dec R3         ; We finished another loop
     jnz I_Loop     ; If I!=0 then loop again

     mov R0, 0x8006 ; Pointer to the Sum variable
     mov [R0], R2   ; Store the total in Sum variable

     halt           ; End of program.
```

Program A – Binary Code

Address	Value	Meaning
0000 0000 0000 0000	0001 0101 1000 0000	MOV R0,
0000 0000 0000 0001	1000 0000 0000 0000	0x8000
0000 0000 0000 0010	0001 0101 1001 0000	MOV R1,
0000 0000 0000 0011	1000 0000 0000 0011	0x8003
0000 0000 0000 0100	0001 0101 1010 0000	MOV R2,
0000 0000 0000 0101	0000 0000 0000 0000	0x0000
0000 0000 0000 0110	0001 0101 1011 0000	MOV R3,
0000 0000 0000 0111	0000 0000 0000 0011	0x0003
0000 0000 0000 1000	0001 0011 1100 1000	MOV R4, [R0]
0000 0000 0000 1001	1000 0001 1010 1100	ADD R2, R4
0000 0000 0000 1010	0001 0011 1100 1001	MOV R4, [R1]
0000 0000 0000 1011	1000 0001 1010 1100	ADD R2, R4
0000 0000 0000 1100	0110 1101 1000 0000	INC R0,
0000 0000 0000 1101	0110 1101 1001 0000	INC R1,
0000 0000 0000 1110	0110 0001 1011 0000	DEC R3
0000 0000 0000 1111	0011 0101 1111 1000	JNZ I_Loop
0000 0000 0001 0000	0001 0101 1000 0000	MOV R0,
0000 0000 0001 0001	1000 0000 0000 0110	0x8006
0000 0000 0001 0010	0001 0010 1000 1010	MOV [R0], R2
0000 0000 0001 0011	1111 1111 1111 1111	HALT
1000 0000 0000 0000	0000 0000 0000 0111	A[0]=7
1000 0000 0000 0001	0000 0000 0000 0011	A[1]=3
1000 0000 0000 0010	0000 0000 0000 1000	A[2]=8
1000 0000 0000 0011	0000 0000 0000 1001	B[0]=9
1000 0000 0000 0100	0000 0000 0000 0010	B[1]=2
1000 0000 0000 0101	0000 0000 0000 0110	B[2]=6
1000 0000 0000 0110	0000 0000 0000 0000	Sum

Instruction Summary

The representation of an assembly instruction (mnemonic) translates directly into a binary string that is understood by the microprocessor. The following pages show the mnemonics in all the different formats that are allowed. With each format, there is a binary string encoding that will be used. An example of the mnemonic as it would be used in a sample program is provided along with the number of bytes that the instruction occupies in memory. If any flags are impacted in the STATUS register, they are listed; all other flags are not changed during the execution of the instruction. The clock cycles required to complete the instruction are not provided since it is unknown at this time how your implementation will operate. Instructions operate at a rate lower than the clock frequency of the processor. This is known as a Complex Instruction Set Computer (CISC) since each instruction may require vastly different numbers of clock cycles.

Some notes about the following pages:

"ddd"	Destination register (R0=000, … R7=111)
"sss"	Source register (R0=000, … R7=111)
"cccc cccc cccc cccc"	Constant – 16-bits long
"rrrr rrrr"	Relative address – 8-bits added to the current address
"aaaa aaaa aaaa aaaa"	Absolute address – 16-bits
"xx"	Condition code (Zero, Carry, Not Overflow, etc)
"bbbb"	Bit index within a 16-bit word.

Operands to an operation can be registers (reg) R0 through R7, memory addresses that are pointed to by a register (mem) [R0] through [R7], or immediate values (immed) better known as constants. Additional operands can be addresses, which are essentially constants. To make these values stand out, they are referred to as displacements (disp8) if they are values that will sign extended to 16-bits and then added to the current address. If they are absolute addresses that specify the full address then they are referred to as locations (loc16). In general, the relative addresses will be used since they require less bytes to represent the complete instruction. The drawback is that they can only jump so far (+127 or –128 bytes) and to make longer jumps the absolute addresses must be used.

The list of instructions below is fairly long if all variants of the instructions are implemented. The instruction formats with asterisks (*) are optional and can be implemented for extra credit. All other instructions must be implemented.

The schematics, components, library, and component test files are available for download at the following link. Also contained in this archive is a utility program for converting Binary files to Intel HEX format that Logic Works can read to program RAM or ROM.
http://www.engr.uconn.edu/~barry/CSE207/FA03/circuits/207_FA03_4_Mapen_Barry_Mircoprocessor.zip

ADD Add two operands

Flags: Z, C, O, S

Syntax

ADD Destination, Source

Function

Destination = Destination + Source

Description

The ADD instruction adds the source and destination operands. The result is stored in the destination replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
ADD reg, reg	1000 0001 1ddd 1sss	ADD R0, R1	2	
*ADD mem, reg	1000 0010 1ddd 1sss	ADD [R0], R1	2	
*ADD reg, mem	1000 0011 1ddd 1sss	ADD R0, [R1]	2	
*ADD mem, immed	1000 0100 1ddd 0000 cccc cccc cccc cccc	ADD [R0], 0x1234	4	
ADD reg, immed	1000 0101 1ddd 0000 cccc cccc cccc cccc	ADD R0, 0x1234	4	

AND Logical AND of two operands

Flags: Z, C, O, S

Syntax

AND Destination, Source

Function

Destination = Destination AND Source

Description

The AND instruction performs a bit-wise AND of the source and destination operands. The result is stored in the destination replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
AND reg, reg	1010 0001 1ddd 1sss	AND R0, R1	2	
*AND mem, reg	1010 0010 1ddd 1sss	AND [R0], R1	2	
*AND reg, mem	1010 0011 1ddd 1sss	AND R0, [R1]	2	
*AND mem, immed	1010 0100 1ddd 0000 cccc cccc cccc cccc	AND [R0], 0x1234	4	
*AND reg, immed	1010 0101 1ddd 0000 cccc cccc cccc cccc	AND R0, 0x1234	4	

CMP Compare two operands

Flags: Z, C, O, S

Syntax

CMP Destination, Source

Function

Destination - Source

Description

The CMP instruction compares the source to the destination and sets the flags appropriately (done by storing the results from the ALU). The instruction is generally implemented as a subtraction, but the result is not stored in the destination. Both the source and destination values are unchanged at the end of this instruction. Usually this instruction is followed by a conditional jump.

	If signed comparison	If unsigned comparison
Destination>Source	Z=0 and S=O	C=0 and Z=0
Destination>=Source	S=O	C=0
Destination=Source	Z=1	Z=1
Destination<=Source	Z=1 and S!=0	C=1 and Z=1
Destination<Source	S=O	C=1

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
CMP reg, reg	`0100 0001 1ddd 1sss`	CMP R0, R1	2	
*CMP mem, reg	`0100 0010 1ddd 1sss`	CMP [R0], R1	2	
*CMP reg, mem	`0100 0011 1ddd 1sss`	CMP R0, [R1]	2	
*CMP mem, immed	`0100 0100 1ddd 0000` `cccc cccc cccc cccc`	CMP [R0], 0x1234	4	
CMP reg, immed	`0100 0101 1ddd 0000` `cccc cccc cccc cccc`	CMP R0, 0x1234	4	

DEC Decrement

Flags: Z, O, S

Syntax

DEC Operand

Function

Operand = Operand - 1

Description

The DEC instruction subtracts 1 from the operand, which is treated as an unsigned number. The carry flag is not changed by this operation.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
DEC reg	`0110 0001 1ddd 0000`	DEC R0	2	
*DEC mem	`0110 0010 1ddd 0000`	DEC [R0]	2	

HALT — Halt the processor

Flags: (none)

Syntax

HALT

Function

None

Description

The HALT instruction is the final instruction that stops the fetch-execute cycle in the processor until it is reset.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
HALT	`1111 1111 1111 1111`	HALT	2	

INC — Increment

Flags: Z, O, S

Syntax

INC Operand

Function

Operand = Operand + 1

Description

The INC instruction adds 1 to the operand, which is treated as an unsigned number. The carry flag is not changed by this operation.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
INC reg	`0110 1101 1ddd 0000`	INC R0	2	
*INC mem	`0110 1110 1ddd 0000`	INC [R0]	2	

JZ/JE Jump if Zero/Equal

Flags: (none)

Syntax

JZ Label

Function

If Z=1 then PC = PC + relative address (relative)
If Z=1 then PC = absolute address (absolute)

Description

The JZ/JE instruction moves the program counter to the location of the next instruction to be executed when the Zero flag is set. If the flag is not set then execution continues with the next instruction.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
JZ/JE disp8	`0011 0010 rrrr rrrr`	JZ Here	2	
*JZ/JE loc16	`0010 0010 0000 0000 aaaa aaaa aaaa aaaa`	JZ FarAway	4	

JAE/JNB/JNC Jump if Above or Equal/Not Below/No Carry

Flags: (none)

Syntax

JNC Label

Function

If C=0 then PC = PC + relative address (relative)
If C=0 then PC = absolute address (absolute)

Description

The JAE/JNB/JNC instruction moves the program counter to the location of the next instruction to be executed when the Carry flag is clear. If the flag is set then execution continues with the next instruction.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
JAE/JNB/JNC disp8	`0011 0100 rrrr rrrr`	JNC Here	2	
*JAE/JNB/JNC loc16	`0010 0100 0000 0000 aaaa aaaa aaaa aaaa`	JNC FarAway	4	

JB/JNAE/JC Jump if Below/Not Above or Equal/Carry

Flags: (none)

Syntax

JC Label

Function

If C=1 then PC = PC + relative address (relative)
If C=1 then PC = absolute address (absolute)

Description

The JB/JNAE/JC instruction moves the program counter to the location of the next instruction to be executed when the Carry flag is set. If the flag is not set then execution continues with the next instruction.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
JB/JNAE/JC disp8	`0011 0011 rrrr rrrr`	JC Here	2	
*JB/JNAE/JC loc16	`0010 0011 0000 0000 aaaa aaaa aaaa aaaa`	JC FarAway	4	

JMP Jump unconditionally

Flags: (none)

Syntax

JMP Label

Function

PC = PC + relative address (relative)
PC = absolute address (absolute)

Description

The JMP instruction unconditionally moves the program counter to the location of the next instruction to be executed.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
JMP disp8	`0011 0001 rrrr rrrr`	JMP Here	2	
JMP loc16	`0010 0001 0000 0000 aaaa aaaa aaaa aaaa`	JMP FarAway	4	

JNE/JNZ Jump if Not Equal/Not Zero

Flags: (none)

Syntax

JNZ Label

Function

If Z=0 then PC = PC + relative address (relative)
If Z=0 then PC = absolute address (absolute)

Description

The JNE/JNZ instruction moves the program counter to the location of the next instruction to be executed when the Zero flag is clear. If the flag is set then execution continues with the next instruction.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
JNE/JNZ disp8	`0011 0101 rrrr rrrr`	JNC Here	2	
*JNE/JNZ loc16	`0010 0101 0000 0000 aaaa aaaa aaaa aaaa`	JNC FarAway	4	

MOV Move Data

Flags: (none)

Syntax

MOV Destination, Source

Function

Destination = Source

Description

The MOV instruction copies data from the source to the destination.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
MOV reg, reg	`0001 0001 1ddd 1sss`	MOV R0, R1	2	
MOV mem, reg	`0001 0010 1ddd 1sss`	MOV [R0], R1	2	
MOV reg, mem	`0001 0011 1ddd 1sss`	MOV R0, [R1]	2	
MOV mem, immed	`0001 0100 1ddd 0000 cccc cccc cccc cccc`	MOV [R0], 0x1234	4	
MOV reg, immed	`0001 0101 1ddd 0000 cccc cccc cccc cccc`	MOV R0, 0x1234	4	

NOT — Logical NOT of an operand

Flags: Z, C, O, S

Syntax

NOT Operand

Function

Operand = NOT Operand

Description

The NOT instruction performs a bit-wise inversion of the operand. The result is stored in the operand replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
NOT reg	`1101 0001 1ddd 0000`	NOT R0, R1	2	
NOT mem	`1101 0010 1ddd 0000`	NOT [R0], R1	2	

OR — Logical inclusive OR of two operands

Flags: Z, C, O, S

Syntax

OR Destination, Source

Function

Destination = Destination OR Source

Description

The OR instruction performs a bit-wise inclusive OR of the source and destination operands. The result is stored in the destination replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
OR reg, reg	`1011 0001 1ddd 1sss`	OR R0, R1	2	
*OR mem, reg	`1011 0010 1ddd 1sss`	OR [R0], R1	2	
*OR reg, mem	`1011 0011 1ddd 1sss`	OR R0, [R1]	2	
*OR mem, immed	`1011 0100 1ddd 0000` `cccc cccc cccc cccc`	OR [R0], 0x1234	4	
*OR reg, immed	`1011 0101 1ddd 0000` `cccc cccc cccc cccc`	OR R0, 0x1234	4	

SUB Subtract two operands

Flags: Z, C, O, S

Syntax

SUB Destination, Source

Function

Destination = Destination - Source

Description

The SUB instruction subtracts the source operand from the destination operand. The result is stored in the destination replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
SUB reg, reg	1001 0001 1ddd 1sss	ADD R0, R1	2	
*SUB mem, reg	1001 0010 1ddd 1sss	ADD [R0], R1	2	
*SUB reg, mem	1001 0011 1ddd 1sss	ADD R0, [R1]	2	
*SUB mem, immed	1001 0100 1ddd 0000 cccc cccc cccc cccc	ADD [R0], 0x1234	4	
SUB reg, immed	1001 0101 1ddd 0000 cccc cccc cccc cccc	ADD R0, 0x1234	4	

TEST Test a bit within a number

Flags: Z

Syntax

TEST Operand, BitIndex

Function

If Operand(BitIndex)=1 then Z=0 else Z=1

Description

The TEST instruction looks at the bit specified by BitIndex within the operand to see if it is a 0 or 1. The value stored in the operand is unchanged at the end of this operation. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
TEST reg, immed	1101 1001 1ddd bbbb	TEST R0, 4	2	
*TEST mem, immed	1101 1010 1ddd bbbb	TEST [R0], 4	2	

XOR Logical exclusive OR of two operands Flags: Z, C, O, S

Syntax

XOR Destination, Source

Function

Destination = Destination XOR Source

Description

The XOR instruction performs a bit-wise exclusive OR of the source and destination operands. The result is stored in the destination replacing the previous value stored there. The flags are set to match the resulting value.

Timing and Encoding

Format	Encoding	Example	Bytes	Clocks
XOR reg, reg	`1100 0001 1ddd 1sss`	XOR R0, R1	2	
*XOR mem, reg	`1100 0010 1ddd 1sss`	XOR [R0], R1	2	
*XOR reg, mem	`1100 0011 1ddd 1sss`	XOR R0, [R1]	2	
*XOR mem, immed	`1100 0100 1ddd 0000 cccc cccc cccc cccc`	XOR [R0], 0x1234	4	
*XOR reg, immed	`1100 0101 1ddd 0000 cccc cccc cccc cccc`	XOR R0, 0x1234	4	

Due Dates

November 5th 2003: Initial Design – Have questions about the design from the top-level. Some of the details left out from the block diagram should be getting filled in.

November 12th, 2003: Functional Design – At this point you should have register-to-register transfers working. If you are having trouble, you need to let your TA know and seek extra help to get this functional.

November 19th, 2003: Preliminary Report – *This paper must be submitted to your TA to show your progress (5 points)*. This will be a printed report only submitted during discussion. If you have questions, you should identify these to your TA and ask for feedback.

December 3rd, 2003: Project Due – ***This project will not be accepted past the end of class on this date.*** Plan accordingly. **NO EXTENSIONS.** Reports submitted on or before November 30th, 2003 will be granted a 5-point early bird bonus. A full report is required along with copies of all schematics. If you have programs (other than the binary to PROM utility) you must include the source listing and executable. All files MUST be zipped and the archive must be named as required in the course information. Please ask if you have questions about how to do this. Your TA will set up a time to meet with you to demonstrate your finished circuit before the end of exams. This will be to help resolve questions about your solution and to aid the TAs with the grading process.