

**Department of Electrical and Computer Engineering  
State University of New York at Stony brook**

**ESE 555 Advanced VLSI Systems Design (Fall 2006)**

**SUNYSB RISC PROCESSOR DESIGN**

**ISA for SUNYSB RISC Processor**

Mnemonic	Operands	OP Code [15:12]	Rdest [11:8]	ImmHi/ OP Code Ext [7:4]	ImmLo/ Rsrc [3:1]	Notes (* is Baseline)
ADD	Rsrc, Rdest	0000	Rdest	0101	Rsrc	*
ADDI	Imm, Rdest	0101	Rdest	ImmHi	ImmLo	*sign extended Imm
ADDU	Rsrc, Rdest	0000	Rdest	0110	Rsrc	
ADDUI	Imm, Rdest	0110	Rdest	ImmHi	ImmLo	sign extended Imm
ADDC	Rsrc, Rdest	0000	Rdest	0111	Rsrc	
ADDCI	Imm, Rdest	0111	Rdest	ImmHi	ImmLo	sign extended Imm
MUL	Rsrc, Rdest	0000	Rdest	1110	Rsrc	
MULI	Imm, Rdest	1110	Rdest	ImmHi	ImmLo	sign extended Imm
SUB	Rsrc, Rdest	0000	Rdest	1001	Rsrc	*
SUBI	Imm, Rdest	1001	Rdest	ImmHi	ImmLo	*sign extended Imm
SUBC	Rsrc, Rdest	0000	Rdest	1010	Rsrc	
SUBCI	Imm, Rdest	1010	Rdest	ImmHi	ImmLo	sign extended Imm
CMP	Rsrc, Rdest	0000	Rdest	1011	Rsrc	*
CMPI	Imm, Rdest	1011	Rdest	ImmHi	ImmLo	*sign extended Imm
AND	Rsrc, Rdest	0000	Rdest	0001	Rsrc	*
ANDI	Imm, Rdest	0001	Rdest	ImmHi	ImmLo	*zero extended Imm
OR	Rsrc, Rdest	0000	Rdest	0010	Rsrc	*NOP=OR R0, R0
ORI	Imm, Rdest	0010	Rdest	ImmHi	ImmLo	*zero extended imm
XOR	Rsrc, Rdest	0000	Rdest	0011	Rsrc	*
XORI	Imm, Rdest	0011	Rdest	ImmHi	ImmLo	*zero extended Imm
MOV	Rsrc, Rdest	0000	Rdest	1101	Rsrc	*
MOVI	Imm, Rdest	1101	Rdest	ImmHi	ImmLo	*zero extended Imm
LSH	Ramount, Rdest	1000	Rdest	0100	Ramount	*-15 to 15 (2s compl)
LSHI	Imm, Rdest	1000	Rdest	000s	ImmLo	*s=sign (0=left, 2s compl)
ASHU	Ramount, Rdest	1000	Rdest	0110	Ramount	-15 to 15 (2s compl)
ASHUI	Imm, Rdest	1000	Rdest	001s	ImmLo	s=sign (0=left, 2s compl)
LUI	Imm, Rdest	1111	Rdest	ImmHi	ImmLo	*load & 8 bit left shift
LOAD	Rdest, Raddr	0100	Rdest	0000	Raddr	*
STOR	Rsrc, Raddr	0100	Rsrc	0100	Raddr	*
SNXB	Rsrc, Rdest	0100	Rdest	0010	Rsrc	
ZRXB	Rsrc, Rdest	0100	Rdest	0110	Rsrc	
Scond	Rdest	0100	Rdest	1101	Cond	
Bcond	Disp	1100	Cond	DispHi	DispLo	*2s compl displacement
Jcond	Rtarget	0100	Cond	1100	Rtarget	*
JAL	Rlink, Rtarget	0100	Rlink	1000	Rtarget	*
TBIT	Roffset, Rsrc	0100	Rsrc	1010	Roffset	offset=0 to 15
TBITI	Imm, Rsrc	0100	Rsrc	1110	Offset	offset=0 to 15
LPR	Rsrc, Rproc	0100	Rsrc	0001	Rproc	
SPR	Rproc, Rdest	0100	Rproc	0101	Rdest	
DI		0100	0000	0011	0000	
EI		0100	0000	0111	0000	

EXCP	vector	0100	0000	1011	Vector	
RETX		0100	0000	1001	0000	
WAIT		0000	0000	0000	0000	
Unused		0000		0100		
Unused		0000		1000		
Unused		0000		1100		
Unused		0000		1111		
Unused		0100		1111		
Unused		1000		0101		
Unused		1000		0111		
Unused		1000		1xxx		

### COND Values for Jcond, Bcond, and Scond

Mnemonic	Bit pattern	Description	PSR Values
EQ	0000	equal	Z=1
NE	0001	not equal	Z=0
GE	1101	greater than or equal	N=1 or Z=1
CS	0010	carry set	C=1
CC	0011	carry clear	C=0
HI	0100	higher than	L=1
LS	0101	lower than or same as	L=0
LO	1010	lower than	L=0 and Z=0
HS	1011	higher than or same as	L=1 or Z=1
GT	0110	greater than	N=1
LE	0111	less than or equal	N=0
FS	1000	flag set	F=1
FC	1001	flag clear	F=0
LT	1100	less than	N=0 and Z=0
UC	1110	unconditional	N/A
	1111	never jump	N/A

### SUNYSB RISC PROCESSOR DESCRIPTION

The group projects for ESE 555 will be based on the processor specification given in this document. The processor specification is based on RISC concepts and is implemented as a three stage pipeline. It uses a 16 bit word and address space, although for simplicity, each address refers to a complete word (two bytes), so the address space is  $2^{17}$  bytes. All instructions are single word. Following the RISC approach, almost all instructions refer to a 16 entry register file. The highest nibble is the operation code (OP Code), the next nibble is usually the destination register address, the remaining byte is an immediate data value for some instructions, or is split into a four bit operation code extension and a four bit source register address for other instructions. (A few instructions are different, so read the specification carefully.) In order to make the project feasible for most groups in the available time, a “baseline” implementation is also given. This uses a selected subset of the instructions and the expectation is that implementation of the baseline processor is the minimum requirement for this course. Each group should plan a customization beyond the baseline, which makes the processor useful for a particular application. Typically, this may involve adding a few instructions beyond the baseline, special registers which allow certain operations to be done more efficiently, and an interface logic to external input/output devices. All baseline instructions should be implemented without change. Added instructions should use the “unused opcodes” which are listed in the

ISA. If it is necessary to replace some of the additional instructions (beyond baseline), discuss it with the instructor.

Block diagram is given of the architecture of the baseline processor as a guideline, but you are free to make additions and changes to it, but you should still follow the RISC approach with pipelined stages. The following sections discuss the functions of the instructions. You should also refer to the notes in the list of instruction.

### **Notes on the Baseline Instruction Set**

All ALU instructions (except CMP, CMPI – see below) write the result back to the destination register. Instructions ending with I are immediate and use the eight least significant bits of the instruction as data, the others are direct, (i.e. instruction” op Rsrc/Imm, Rdest” performs

Rdest <-Rdest op Imm (sign extended)

or

Rdest <- Rdest op Rsrc  
respectively)

For the baseline SUNYSB processor, the instructions marked with an asterisk in the instruction table should be implemented. Successive memory addresses can refer to 16 bit words instead of bytes. Of the baseline subset of instructions, the only ones which can change the program status register (PSR) are the arithmetic instructions ADD, ADDI, SUB, SUBI, CMP, CMPI. CMP and CMPI perform the same operations as SUB, SUBI but affect different PSR flags (see below) and do not write back the result. Only flags FLCNZ are needed for the baseline implementation.

ADD, ADDI, SUB, SUBI set the C flag if a carry/borrow from the most significant bit position occurs when the operands are treated as unsigned numbers, and set the F flag if an overflow occurs when the operands are treated as two’s complement numbers. (Note: the processor does not know which interpretation you are using, so must set both flags appropriately for each operation.) CMP, CMPI perform a subtraction without write back to Rdest and set the Z flag if the result is zero, set the L flag if Rsrc/Imm > Rdest when the operands are treated as unsigned numbers (i.e. when a carry/borrow occurs), and set the N flag if Rsrc/Imm > Rdest when the operands are treated as two’s complement numbers (N can be computed as the exclusive-or of L and the sign bits of Rsrc/Imm and Rdest.) All other baseline instructions leave the flag unchanged.

Jcond, Bcond are absolute and relative jumps respectively based on the condition codes specified in the condition code (cond) table. JAL (jump and link) stores the address of the next instruction in Rlink, and jumps to Rtarget. Its main use is for subroutine calls. Return with a JUC Rlink (where Rlink is the same register used to store the link).

LSH is a logical left shift by the number of bits specified in Rsrc/Imm treated as a signed two’s complement number (which must be in the range -15 to +15). A negative left shift is effectively a right shift.

LOAD and STOR instruction load from, and store to the data memory location whose address is in register Raddr. The NOP instruction is really OR r0,r0 and does not need to be implemented separately. Unconditional jumps (JUMP) and branches (BR) are equivalent to JUC and BUC respectively, so do not need separate implementation either. Compiler may have these alternative instruction ops for convenience, however.

LUI (load upper immediate) loads the 8 bit immediate data into the upper (most significant) bits of the destination register.

MOV copies the source register or immediate into the destination register.

### **Notes on the Additional Instruction Set**

In a full implementation, PSR (program status register) is a dedicated 16 bit register with flag entries (in the following order, MSB at the left) rrrrIPE0NZF00LTC, where the “r” entries are reserved, the “0” entries are zeros, I, E are used for interrupt processing, T, P are for program tracing (debugging), and the rest are flags have been defined elsewhere.

ADDU does the same as ADD but does not affect the PSR flags. ADDC does the same as ADD except the C flag is also added in. It affects the same flag.

ASHU does an arithmetic left shift interpreting both operands as signed two’s complement.

MUL multiplies:  $R_{dest} \leftarrow R_{src}/Imm * R_{dest}$ . High order bits are truncated if they do not fit in Rdest. No flags are affected.

SUBC does the same as SUB except that the C flag is also subtracted. It affects the same flags.

SNXB converts the 8-bit operand in Rsrc to 16 bits in Rdest with sign-extension.

Scond sets  $R_{dest}=1$  if the condition is true (i.e. if the bit in the PSR is set), and resets  $R_{dest}=0$  if it is false (same condition codes as jump and branch instructions).

DI, EI, EXCP, RETX deal with interrupts and exceptions. Ask if you are interested in implementing any of them.

LPR, SPR load the PSR from Rsrc and store PSR into Rdest, respectively.

TBIT copies the bit in position *offset* to the F flag of the PSR.

WAIT suspends program execution until an interrupt occurs (or forever, if interrupts are not implemented).

ZRXB converts the 8-bit operand in Rsrc to 16 bits in Rdest with zeros-extension.

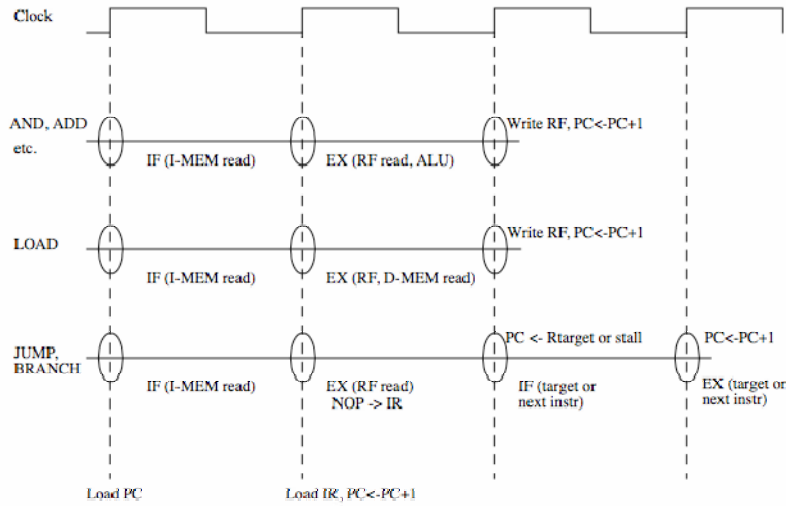
### **Assignment**

Develop and draw schematic of architecture for SUNYSB ISA. You will have to select the depth of pipeline stage. Simulation is not necessary at this stage. However, you have to verify your architecture for each instruction. Grading is based on clarity of your report.

Due date: Feb. 22

Example: 2-stage pipeline

## Instruction Timing for Baseline Processor



On the second clock rise the PC loads the addr of the next instruction. On the third clock rise the PC loads the addr of the next + 1 instruction for AND, ADD etc. and LOAD It loads the target addr or remains unchanged for JP/BR. On the fourth clock rise PC points to the next + 1 instruction for untaken JP/BR, and the target + 1 for taken JP/BR.

1

## 2-Stage Pipeline for Baseline Processor Using NOPs

Code  
`or r1, r2`  
`load r4, r3`  
`and r3, r2`  
`and r1, r2`

Pipe Stages for load

	Fetch	Execute
clock1	or r1, r2	prev
clock2	load r4, r3	or r1, r2
clock3	and r3, r2	load r4, r3
clock4	and r1, r2	and r3, r2
clock5	next	and r1, r2

Clock Cycles for load

	clock0	clock1	clock2	clock3	clock4	clock5	clock6
prev	IF	EX					
or r1, r2		IF	EX				
load r4, r3			IF	EX			
and r3, r2				IF	EX		
and r1, r2					IF	EX	

2

## 2-Stage Pipeline for Baseline Processor Using NOPs

Code (Jump Taken)

```

or r1, r2
jmp r4
nop
and r3, r2
...
...
...
and r1, r2 ←
...
...

```

Pipe Stages for jmp

	Fetch	Execute
clock 1	or r1, r2	prev
clock 2	jmp r4	or r1, r2
clock 3	nop	jmp r4
clock 4	and r1, r2	nop
clock 5	next	and r1, r2

Clock Cycles for jmp

	clock 0	clock 1	clock 2	clock 3	clock 4	clock 5
prev	IF	EX				
or r1, r2		IF	EX			
jmp r4			IF	EX		
nop				IF	EX	
and r1, r2					IF	EX

## 2-Stage Pipeline for Baseline Processor Using NOPs

Code (Jump Not Taken)

```

or r1, r2
jmp r4
nop
and r3, r2
...
...
...
and r1, r2
...
...

```

Pipe Stages for jmp

	Fetch	Execute
clock 1	or r1, r2	prev
clock 2	jmp r4	or r1, r2
clock 3	nop	jmp r4
clock 4	and r3, r2	nop

Clock Cycles for jmp

	clock 0	clock 1	clock 2	clock 3	clock 4	clock 5
prev	IF	EX				
or r1, r2		IF	EX			
jmp r4			IF	EX		
nop				IF	EX	
and r3, r2					IF	EX

## 2-Stage Pipeline for Baseline Processor Using Stalls

Code (Jump Taken)

```

or r1, r2
jmp r4
and r3, r2
...
...
...
and r1, r2 ←
...
...

```

Pipe Stages for jmp

	Fetch	Execute
clock 1	or r1, r2	prev
clock 2	jmp r4	or r1, r2
clock 3	and r3, r2	jmp r4
clock 4	and r1, r2	forced nop

Clock Cycles for jmp

	clock0	clock1	clock2	clock3	clock4	clock5
prev	IF	EX				
or r1, r2		IF	EX			
jmp r4			IF	EX	(New PC)	
and r3, r2				IF	stall	
and r1, r2					IF	EX

5

## 2-Stage Pipeline for Baseline Processor Using Stalls

Code (Jump Not Taken)

```

or r1, r2
jmp r4
and r3, r2
...
...
...
and r1, r2

```

Pipe Stages for jmp

	Fetch	Execute
clock 1	or r1, r2	prev
clock 2	jmp r4	or r1, r2
clock 3	and r3, r2	jmp r4
clock 4	and r3, r2	forced nop

Clock Cycles for jmp

	clock 0	clock 1	clock 2	clock 3	clock 4	clock 5
prev	IF	EX				
or r1, r2		IF	EX			
jmp r4			IF	EX		
and r3, r2				IF	stall	EX
next instr						IF

6

## **2-Stage Pipeline for Baseline Processor Using Stalls**

### **Optimization for Untaken Jumps and Branches**

- Do not stall when a jump or branch is not taken.
- Since the next instruction has already been fetched, it can be executed on the next clock cycle.
- Simple to implement on a 2-stage pipe, more complex in deeper pipes.