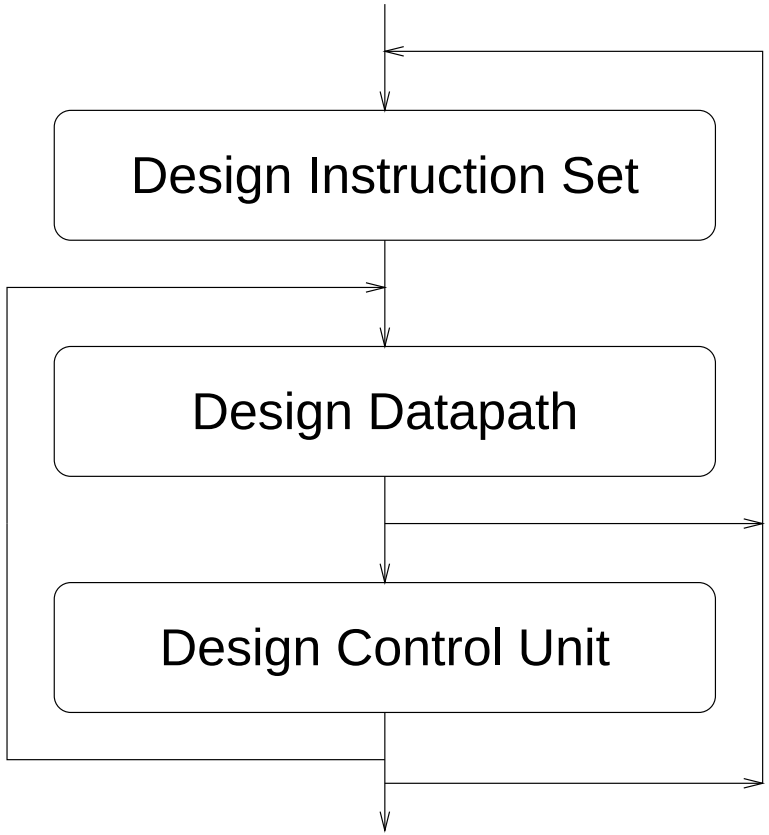
Basic Processor Design



This lecture deals with Instruction Set Design.

Instruction Set Terminology

Mnemonic (Instruction Name)

SUBI

Example

Syntax (Language Structure) *Assembly Language*

`SUBI Rx, Ry, imm4`

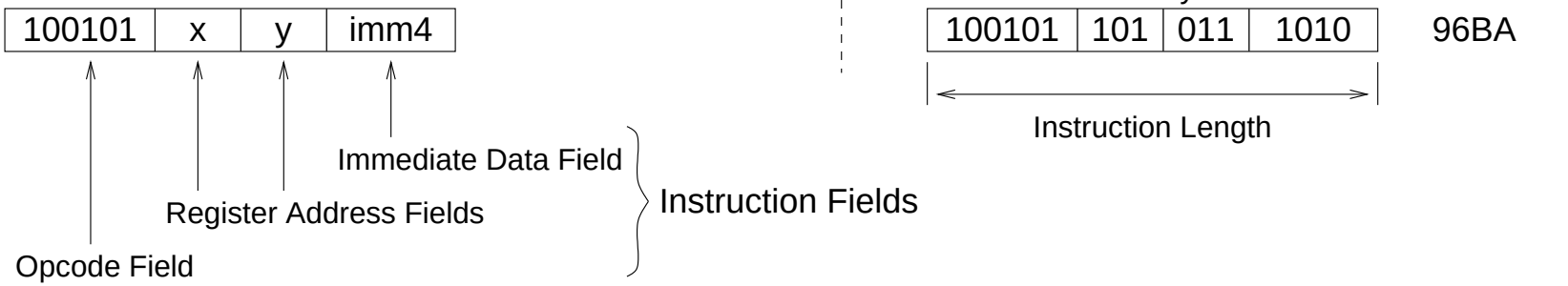
`SUBI R5, R1, 10`

Semantics (Meaning) *Register Transfer Language*

Rx ⟵ Ry - imm4

R5 ⟵ R1 - 10

Coding *Machine Code*



The *instruction set* for a processor is the set of all the instructions supported by that processor.

First Decision:

RISC or CISC

RISC: Reduced Instruction Set Computer

CISC: Complex Instruction Set Computer[1]

Hybrid: Based on $\begin{matrix}\text{RISC}\\\text{CISC}\end{matrix}$ with $\begin{matrix}\text{CISC}\\\text{RISC}\end{matrix}$ attributes

Note that whichever philosophy you follow, you will be designing a simple machine. Choosing the CISC philosophy will not necessarily result in a more complex design.

[1] actually CISC is used to describe all none RISC machines regardless of complexity

Basic Processor Design – RISC

- All instructions are the same length
  - typically 1 word (i.e. 16 bits)
- Load/Store Architecture
  - All arithmetic and logic instructions deal only with registers and immediate values[2]

    e.g. `ADD R3,R2,R5` R5 ← R3 + R2

    `OR R3,13,R5` R5 ← R3 | 13
  - Separate instructions are needed for access to locations in memory.

    e.g. `LD [R4+13],R7` R7 ← *mem*(R4 + 13)

    `ST R7,[R4+13]` *mem*(R4 + 13) ← R7

    mem(nnn) is shorthand for the data location in memory with address nnn.
  - Instruction set is maximally orthogonal

[2] an immediate (or literal) value is a data value encoded in the instruction word.

Basic Processor Design – RISC

Q1$_{RISC}$
How many Register Addresses in an Arithmetic/Logic Instruction?

- Usually 2 or 3 for RISC

  2: `ADD Rx,Ry` Rx $\leftarrow$ Rx + Ry

  3: `ADD Rx,Ry,Rz` Rz $\leftarrow$ Rx + Ry

Q2$_{RISC}$
How many General Purpose Registers?

- Usually $2^n - 1$ for RISC

  This gives 2^n addressable registers including the dummy register, R0[3].

  We then need n bits per register address in the instruction.

  With a 16 bit instruction length, $n = 2$ (i.e. 3 registers + R0) or $n = 3$ (i.e. 7 registers + R0) are sensible values.

[3] R0 is always zero

Basic Processor Design – RISC

Q3$_{RISC}$
How many Bits do we use for Short Immediates?

- Used in instructions like

`ADD R3,5,R2`	R2 $\leftarrow$ R3 + 5
`ST R7,[R4+13]`	*mem*(R4 + 13) $\leftarrow$ R7

Sensible values for a 16 bit instruction length are in the range $4 \leq s \leq 9$
giving 2's complement values in the range $-2^{s-1} \leq imm \leq 2^{s-1} - 1$

Q3A$_{RISC}$
Do we support All Arithmetic/Logic instructions and All Load/Store instructions in both Register-Register and Register-Immediate forms?

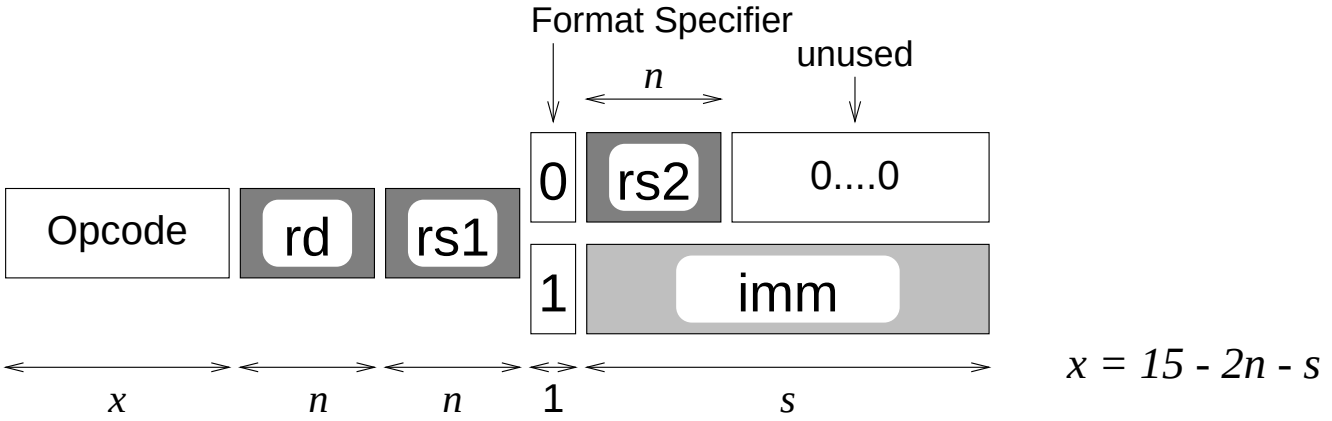
- Some RISC processors support only Register-Immediate form for Load/Store instructions.
- Some RISC processors support Register-Register form for all Arithmetic/Logic instructions and Register-Immediate form for a subset of these instructions.

Basic Processor Design – RISC

$Q4_{RISC}$
What Instruction Fields do we provide? How are they arranged?

- RISC instruction coding is highly orthogonal - any instruction may use any registers.
- Requirement for maximum length short immediate makes RISC coding tight.

Assume $Q1_{RISC}$: 3, $Q2_{RISC}$: $2^n - 1$, $Q3_{RISC}$: s, $Q3A_{RISC}$: YES
A suitable coding for Arithmetic/Logic and Load/Store instructions is:



Example: If $n = 2$ and $s = 7$ then $x = 4$ giving up to 2^x (=16) instructions.

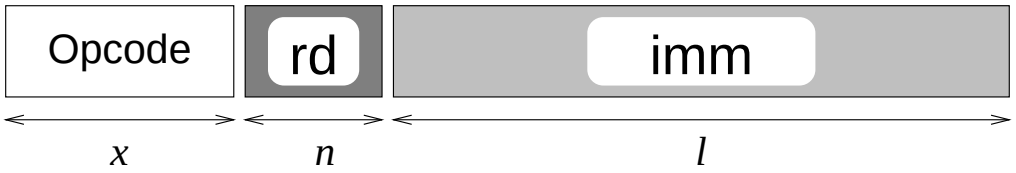
Basic Processor Design – RISC

Most RISC processors support an instruction to set the upper bits of a register. The MIPS processor calls it LUI (load upper immediate) while the SPARC processor calls it SETHI.
For SPARC, the sequence of instructions required to set upper and lower parts of a register is:

`SETHI 200,Rx`	Rx $\leftarrow$ 200 $\times 2^{10}$
`ADD Rx,5,Rx`	Rx $\leftarrow$ Rx + 5

Note that the $\times 2^{10}$ (i.e. shift left by 10) value comes from the SPARC word length (32) less the length of the long immediate used for SETHI (22). In our example it will be $\times 2^{16-l}$ where l is the length of our long immediate.

To code a SETHI or LUI instruction we need fewer fields:

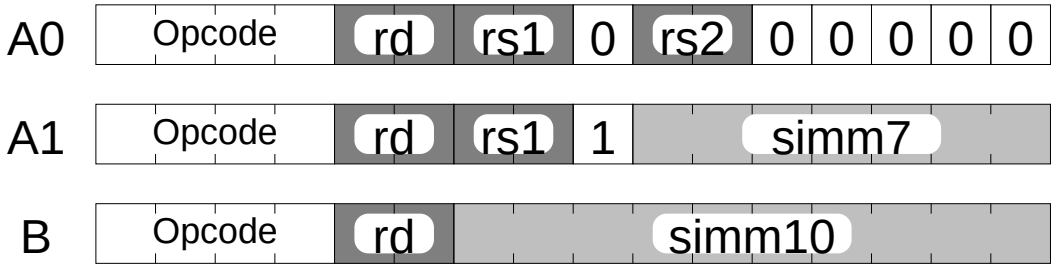


$l = 16 - n - x$

Example: If $n = 2$ and $x = 4$ then $l = 10$.
Note: $s + l \geq 16$ for the SETHI/ADD sequence to produce a 16 bit result.

Example Coding #1

Assume $Q1_{RISC}$: 3, $Q2_{RISC}$: 3 ($n = 2$), $Q3_{RISC}$: 7 ($s = 7$), $Q3A_{RISC}$: YES
This gives $x = 4$ and $l = 10$ with the coding shown below[4]:



In this case up to 16 instructions are supported, each of which supports either coding A (i.e. A0 and A1) or coding B.

This is just one of many possible codings (this one is loosely based on the SPARC instruction coding).

[4]note that in this example short and long immediates (`simm7` and `simm10`) are signed numbers

Alternative Example Coding #2

An alternative solution is based on the following assumptions:
Q1$_{RISC}$: 3 (or 2 for register-immediate instructions), Q2$_{RISC}$: 7 ($n = 3$),
Q3$_{RISC}$: 8 ($s = 8$ and $l = 8$), Q3A$_{RISC}$: NO

OpcodeA allows for up to 32 Register-Register Arithmetic/Logic instructions.

OpcodeB allows for up to 8 Register-Immediate Arithmetic/Logic instructions.

OpcodeC allows for up to 2 Register-Immediate Load/Store instructions[5].

[5] in this case we know that one of the two will be load and the other will be store

Basic Processor Design – RISC

$Q5_{RISC}$
What Instructions will we support?

Type	Function	Mnemonic	Function	Mnemonic
Arithmetic	Add	ADD	Subtract	SUB
	Add with Carry	ADDX	Subtract with Borrow	SUBX
Logic	Bitwise AND	AND	Bitwise OR	OR
	Bitwise Exclusive OR	XOR	Logical Shift Right	LSR
Data Movement	Load	LD	Store	ST
	Set High	SETHI		
Control Transfer	Branch if Zero	BZ	Branch to Subroutine	BSR
	Jump and Link	JMPL		

This set has been chosen based on a maximum of 16 instructions (to match example coding #1), with support for a complete set of common arithmetic and logical functions (note that multiply is too complex to be included, while shift left is accomplished by adding a number to itself).

The control transfer functions are a minimum set to support subroutines. BZ provides a conditional PC relative branch (unconditional if R0 is tested). BSR provides a PC relative branch which stores the calling address in a register. JMPL provides the ability to return from a subroutine and also the ability to jump a long way.

Basic Processor Design – RISC

Q5A$_{RISC}$ Define Assembly Language Syntax[6] and Semantics.

	Mnemonic	Format	Syntax	Semantics
Arithmetic	ADD*	A	`ADD Rs1,Op2,Rd`	$Rd \leftarrow Rs1 + Op2$
			where `Op2` is either `Rs2` or `simm7`	
	SUB*	A	`SUB Rs1,Op2,Rd`	$Rd \leftarrow Rs1 - Op2$
	ADDX*	A	`ADDX Rs1,Op2,Rd`	$Rd \leftarrow Rs1 + Op2 + C$
	SUBX*	A	`SUBX Rs1,Op2,Rd`	$Rd \leftarrow Rs1 - Op2 - C$
Logic	AND	A	`AND Rs1,Op2,Rd`	$Rd \leftarrow Rs1$ & $Op2$
	OR	A	`OR Rs1,Op2,Rd`	$Rd \leftarrow Rs1$ | $Op2$
	XOR	A	`XOR Rs1,Op2,Rd`	$Rd \leftarrow Rs1$ ^ $Op2$
	LSR*	A	`LSR Rs1,Rd`	$Rd \leftarrow Rs1 >> 1$
Data Movement	LD	A	`LD [Rs1+Op2],Rd`	$Rd \leftarrow mem(Rs1 + Op2)$
	ST	A	`ST Rd,[Rs1+Op2]`	$mem(Rs1 + Op2) \leftarrow Rd$
	SETHI	B	`SETHI simm10,Rd`	$Rd \leftarrow simm10 << 6$
Control Transfer	BZ	B	`BZ Rd,simm10`	$if\ (Rd = 0)\ then\ PC \leftarrow PC + simm10$
	BSR	B	`BSR simm10,Rd`	$Rd \leftarrow PC$; $PC \leftarrow PC + simm10$
	JMPL	A	`JMPL Rs1+Op2,Rd`	$Rd \leftarrow PC$; $PC \leftarrow Rs1 + Op2$

[6]operand order follows SPARC convention *marked commands update the Carry flag (C)

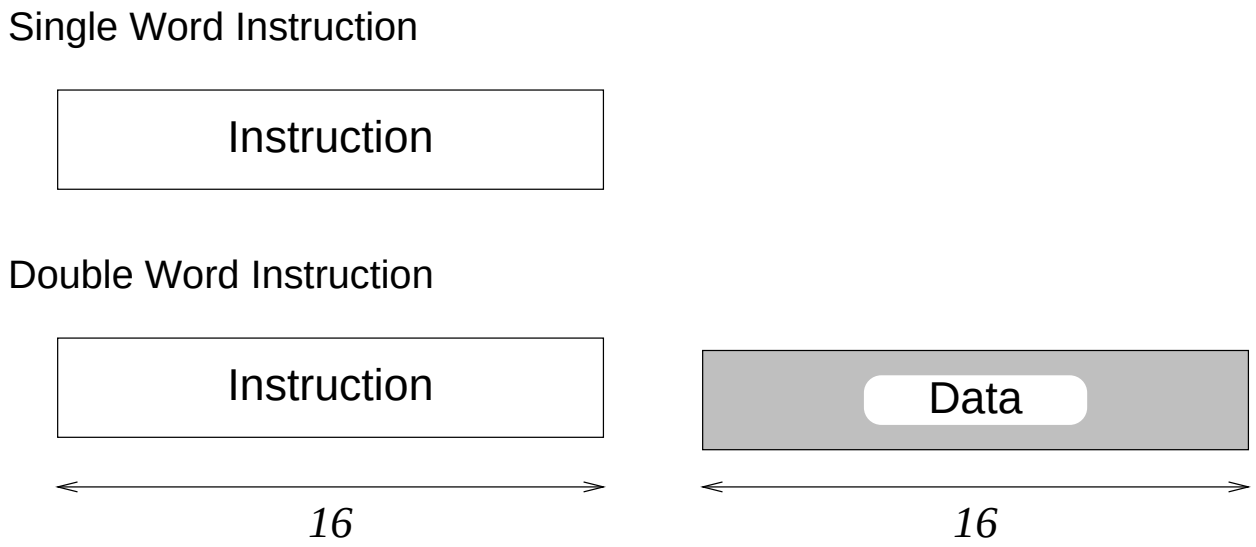
$Q5B_{RISC}$
What Opcodes will be Assigned?

Op[3:2] \ Op[1:0]	00	01	11	10	
00	ADD	ADDX	AND	OR	Format A
01	SUB	SUBX	XOR	LSR	
11	LD	ST	-	JMPL	
10	SETHI	-	BZ	BSR	Format B

Instructions are grouped so that decoding is simple.

Basic Processor Design – CISC

- Instruction Length is Variable
  - instructions with implied operands will typically be 1 word (i.e. 16 bits)
  - instructions with an explicit operand will be 2 words



The data word contains either the operand itself (immediate addressing mode) or a value which will be used to calculate the operand address in memory.

Basic Processor Design – CISC

- Register-Memory Architecture

  A typical arithmetic or logic instruction will take one operand from a register and one from memory. It will return the result to a register.

  e.g. `ADDA 203` $A \leftarrow A + mem(203)$

  A is the implied operand while 203 is the address of the explicit operand.

- A single instruction may give rise to a complex sequence of events

  e.g. `JSR +137` $SP \leftarrow SP - 1$

  $mem(SP) \leftarrow PC$

  $PC \leftarrow PC + 137$

  The old value of the program counter is stored on a stack in memory during a jump to subroutine instruction.

$Q1_{CISC}$
How many Data Registers and Address Registers?

The minimum system for this exercise will have just one data register (Accumulator: A) and one address register (Stack Pointer: SP).

- Extra Address Registers.

  Having another address register (e.g. Index Register: X) should help to reduce the number of memory accesses to speed up the processor. It should also help to simplify the programming.

- Extra Data Registers.

  Having another data register (e.g. Accumulator: B) is likely to have less effect unless you add support for Register-Register arithmetic/logic instructions.

- Multi-Purpose Registers.

  For this exercise it may be beneficial to allow any register to be used as either a data register or an address register, with just one of the registers acting also as the stack pointer.

$Q2_{CISC}$
What Addressing Modes should we support for Arithmetic/Logic/Load & Store?

Basic addressing modes[7]:

Addressing Mode	Assembly Language Syntax	Semantics
Immediate	`ADDA #imm`	$A \leftarrow A + imm$
	—	—
Direct	`ADDA addr`	$A \leftarrow A + mem(addr)$
	`STA addr`	$mem(addr) \leftarrow A$
Indexed with X	`ADDA X,offset`	$A \leftarrow A + mem(X + offset)$
	`STA X,offset`	$mem(X + offset) \leftarrow A$
Indexed with SP	`ADDA SP,offset`	$A \leftarrow A + mem(SP + offset)$
	`STA SP,offset`	$mem(SP + offset) \leftarrow A$
Stack (pop)	`ADDA SP++`	$A \leftarrow A + mem(SP); SP \leftarrow SP - 1$
(push)	`STA --SP`	$SP \leftarrow SP - 1; mem(SP) \leftarrow A$

[7] It is not necessary to implement all of these modes

Basic Processor Design – CISC

Assembly language conventions used here:

`#nnn`	The literal value nnn is the operand
`nnn`	nnn is the address of the operand
`Reg,nnn`	$Reg + nnn$ is the address of the operand
`(nnn)`	nnn is the address of a pointer to the operand
`--Reg`	Reg is pre-decremented before the address is calculated
`Reg++`	Reg is post-incremented after the address is calculated

Less useful addressing modes:

Addressing Mode	Assembly Language Syntax	Semantics
Indirect	`ADDA (point_addr)`	$A \leftarrow A + mem(mem(point\_addr))$
(Pre-)Indexed Indirect	`ADDA (X,offset)`	$A \leftarrow A + mem(mem(X + offset))$
Indirect (Post-)Indexed	`ADDA X,(point_addr)`	$A \leftarrow A + mem(X + mem(point\_addr))$

These modes all use multiple memory accesses to reach data in complex data structures. Since such data structures are unlikely to be required for this processor, the extra complexity involved in implementing them is not justified by the benefit gained.

$Q3_{CISC}$
What Instructions will we support?

Because we do not need to include an immediate field in the 16 bit instruction word, we can have more instructions[8].
This is one possible set based on the CISC example processor model in Verilog:

Type	Function	Mnemonic	Function	Mnemonic
Arithmetic	Add	ADDA	Subtract	SUBA
	Add with Carry	ADCA	Subtract with Borrow	SBCA
Logic	Bitwise NOT	COMA	Bitwise AND	ANDA
	Bitwise OR	ORA	Bitwise Exclusive OR	XORA
	Logical Shift Right	LSRA	Logical Shift Left	LSLA
Data Movement	Load	LDA,LDX,LDS	Store	STA,STX,STS
Control Transfer	Branch Always	BA		
	Branch if Equal	BEQ	Branch if Not Equal	BNE
	Branch to Subroutine	BSR	Return from Subroutine	RTS

[8]this lack of constraint is all the more reason to choose the instruction set carefully

$Q4_{CISC}$
Which Addressing Modes are Supported for each Instruction?

- Inherent

  For certain instructions (LSRA, LSLA, COMA, RTS) the addressing mode is implied/inherent in the instruction. These are always single word instructions.

- PC Relative

  For Branch instructions the addressing mode is PC-relative (some processors use PC-absolute instead). The branch destination is calculated by adding the branch offset to the current program counter (PC).

- Other Instructions

  The remaining instructions can support any of the addressing modes described in $Q2_{CISC}$[9].

[9] with the exception that a store instruction in immediate mode is meaningless

$Q5_{CISC}$
How is the Instruction Coded?

Normally CISC instructions are compactly coded leading to complex decode logic. The most one might expect is an instruction field that specifies the addressing mode.

With a word length of 16 bits and no immediate to include we can produce an orthogonal coding with many sub-fields for easy decoding (and potentially more power).

Note: just a few of the potential sub-fields are shown here.