
CHAPTER 3 RISC Processor

All rights reserved
Copyright ©2006 Nicholas L. Pappas
Copying in any form without the expressed written
permission of Nicholas L. Pappas is prohibited

3.1 RISC Computers

- 3.1.1 RISC ul Set
 - 3.1.1.1 Literals
 - 3.1.1.2 Address Modes
 - 3.1.1.3 Load and Store
 - 3.1.1.4 Operators
 - 3.1.1.5 Program Control
 - 3.1.1.6 ul Codeword Formats
- 3.1.2 The First Step from ul to User Data Path
 - 3.1.2.1 Bus
 - 3.1.2.2 Register File
 - 3.1.2.3 Memory
 - 3.1.2.4 Program Counter and Instruction Register
 - 3.1.2.5 ALU and Shifter
 - 3.1.2.6 Temporary Register
 - 3.1.2.7 Status
 - 3.1.2.8 Other Items
- 3.1.3 The Second Step from ul to User Data Path
 - 3.1.3.1 RISC ml Word Format
 - 3.1.3.2 Branch to fetch_ul
 - 3.1.3.3 Shift Operations
 - 3.1.3.4 Scc
 - 3.1.3.5 CONST
 - 3.1.3.6 Load
 - 3.1.3.7 Store
 - 3.1.3.8 Program Control
 - 3.1.3.9 uData Path
- 3.1.4 Microinstructions
- 3.1.5 Micro Data Path
- 3.1.6 Micro Control
- 3.1.7 Microprograming
 - 3.1.7.1 ml Fields

- 3.1.7.2 ml used as Fetch_ul Control cl
- 3.1.7.3 Arithmetic and Logical Operators
- 3.1.7.4 Shift Operators
- 3.1.7.5 CONST
- 3.1.7.6 Scc
- 3.1.7.7 Load
- 3.1.7.8 Store
- 3.1.7.9 Program Control

Overview

We define a reduced instruction set computer (RISC) by pulling out of thin air a minimum RISC user instruction (ul) set. The ul set is one specification for the RISC computer design. Then the RISC concept is embodied in the main goal of no lists of ml. The one-ml-per-ul goal is another specification for the design.

Guided by the desire to minimize the number of time consuming memory accesses literals and address modes are treated differently as we show how and why memory accesses are limited to load and store ul. Next, new ul types are discussed and ul codeword formats are defined.

Starting from the ul set the preliminary user data path is evolved. We show how the data path is some set of combinational circuit blocks and sequential circuit blocks interconnected by data buses. However this preliminary user data path does not fulfill the main goal. The ul set does not provide the necessary information.

We show how to benefit from what we know by using the CISC ml set to write RISC ul ml lists. Analysis of the lists reveals how special hardware replaces ml in those lists. In this way the lists are reduced to one ml except for load and stores (two ml) and a suitable user data path results. Given the user data path control lines the RISC mControl and associated ASM is created by modifying the CISC mControl and ASM.

As we proceed through the above processes we do extensive microprogramming demonstrating that any ul can be represented by ml lists with only one or two ml in each list.

Introduction

In the beginning computers used simple user instructions (ul) which were directly executed by the computer. After microprogramming was invented by Wilkes, and ml's were born, the computer instruction set quickly became complex. Designers used ml lists to implement complex ul and the CISC evolved. The reduced instruction set computer (RISC) is a return to the original computers no ml lists. At its best each RISC ul executes in one computer cycle.

Key RISC characteristics are:

- ul execution in one computer cycle
- memory access limited to load and store ul
- simple address modes
- arithmetic, logic, and shift ul only have register operands
- large register set and register windows
- fixed word length and simple codeword format for ul
- microcode replaced by hardwired logic (sometimes)

The last RISC characteristic in the list is the preferred implementation for RISC microprocessors. This characteristic is not mandatory for implementing RISC as will be demonstrated. RISC computers that do not use a RISC microprocessor can and do use microcode to great advantage. The programmer thinks reduced instruction set computers are different primarily because only load and store ul access memory with one or two simple address modes. Another difference is that programs need more instructions because the ul are not complex. The designer thinks RISC is different primarily because the ml lists contain only one or two ml.

3.1.1 RISC ul Set

Again we pull out of thin air a minimum RISC set (Table 3.1) we use to drive the design process. This time a three address format is used, and the following changes are made to the CISC format to create a RISC format.

3.1.1.1 Literals

In 2.1 CISC Computers we learned the CISC ul set uses literals to implement some operands. These literal words followed ul words in the program memory space. One important idea underlying the risc philosophy is minimizing the number of memory accesses simply because accesses slow down the program execution process. Therefore, unlike CISC, the RISC ul set uses no separate literal words. Literals are fields in the RISC ul codewords. This is why RISC ul have a constant length of one word. Whereas the length of a CISC ul is one or two or three or four words.

This ul set is a three address set.

rx, ry source registers, rz destination register

sx is a source register rx or a 16 bit constant n

sx', rz' mean their contents are 1's complements

N C Z V status bits are updated if special ul scc bit is 1

TABLE 3.1 RISC ul set

Operators:	
ADD <i>sx ry rz</i>	$rz \leftarrow ry + sx$
ADDC <i>sx ry rz</i>	$rz \leftarrow ry + sx + c$
SUB <i>sx ry rz</i>	$rz \leftarrow ry + sx' + 1 \quad (ry \quad sx)$
SUBC <i>sx ry rz</i>	$rz \leftarrow ry + sx' + c$
SUBR <i>sx ry rz</i>	$rz \leftarrow sx + ry' + 1 \quad (sx \quad ry)$
SUBRC <i>sx ry rz</i>	$rz \leftarrow sx + ry' + c$
AND <i>sx ry rz</i>	$rz \leftarrow sx \text{ AND } ry$
OR <i>sx ry rz</i>	$rz \leftarrow sx \text{ OR } ry$
XOR <i>sx ry rz</i>	$rz \leftarrow sx \text{ XOR } ry$
ROTL <i>sx ry rz</i>	$rz \leftarrow ry \ll sx \text{ (msb to lsb each shift)}$

TABLE 3.1 RISC uI set

SLL	<i>sx ry rz</i>	$rz \leftarrow ry \ll sx$ (zero fill)
SRA	<i>sx ry rz</i>	$rz \leftarrow ry \gg sx$ (sign fill) k in sx
SRL	<i>sx ry rz</i>	$rz \leftarrow ry \gg sx$ (zero fill) shift k bits
CONST+	<i>n rz</i>	$rz \leftarrow 0^{16} \# n$
CONST	<i>n rz</i>	$rz \leftarrow 1^{16} \# n$
CONSTH	<i>n rz</i>	$rz \leftarrow n \# 0^{16}$
Scc	<i>sx ry rz</i>	If cc true then $rz \leftarrow 1$ else $rz \leftarrow 0$ (form ry rx to set status)(cc in Table 2.2)
Load and Store:		
Lp	<i>*ry(n) rz</i>	$rz \leftarrow M[ry+n]$
Sq	<i>ry *rx(n)</i>	$M[rx+n] \leftarrow ry$
where p is B (byte), H (halfword) or W (word) BU (byte unsigned), HU (halfword unsigned) where q is B (byte), H (halfword) or W (word)		
Program control:		
BZ	<i>n ry</i>	if $ry=0$ then $pc \leftarrow pc + n$ else $pc \leftarrow pc+4$
BNZ	<i>n ry</i>	if $ry \neq 0$ then $pc \leftarrow pc + n$ else $pc \leftarrow pc+4$
JMP	<i>m</i>	$pc \leftarrow pc + m$
JMP	<i>ry</i>	$pc \leftarrow ry$
CALL	<i>m</i>	$r31 \leftarrow pc, pc \leftarrow pc + m$
CALL	<i>ry</i>	$r31 \leftarrow pc, pc \leftarrow ry$
TRAP		$int \leftarrow pc, pc \leftarrow 0$
RTE		$pc \leftarrow int$ (return from exception)
HALT		halt
NOP		no operation

src = source data

dst = destination data

N C Z V are status bits

TABLE 3.2 RISC Condition Codes cc

Unconditional cc:		cc equation (note 1)		
		C=carry	C=carry'	code
UN	branch always	none		00hex
Unsigned compare cc:				
LO	dst lower than src	C'	C	01
HS	dst higher or same as src	C	C'	02
LS	dst lower or same as src	C'+ Z	C + Z	03
HI	dst higher than src	C Z'	C' Z'	04
EQ	dst equal to src	Z	Z	05
NE	dst not equal to src	Z'	Z'	06
Signed compare cc:				
LT	dst less than src	NV'+ N'V		07
GE	dst greater than or equal to src	NV + N'V'		08
LE	dst less than or equal to src	NV'+ N'V + Z		09
GT	dst greater than src	(NV + N'V')Z'		0A
EQ	dst equal to src	Z		0B
NE	dst not equal to src	Z'		0C
Compare to zero cc:				
Z	result equal to zero	Z		0D
NZ	result not equal to zero	Z'		0E
P	result is positive, >0	N'Z'		0F
N	result is negative, <0	N (sign)		10
NN	result not neg, >0 or =0	N'		11
Arithmetic cc:				
Z	result equal to zero	Z	Z (zero)	12
NZ	result not equal to zero	Z'	Z'	13

TABLE 3.2 RISC Condition Codes cc

C	result sets carry	C	C (carry)	14
NC	result clears carry	C'	C'	15
V	result overflows	V	V (overf)	16
NV	result does not overflow	V'	V'	17
B	result sets borrow	C'	C	18
NB	result does not borrow	C	C'	19

Note 1: use of C or C' is the designer's choice.

3.1.1.2 Address Modes

The ml lists for CISC ul needed many ml to implement the various address modes. By definition RISC ul are supposed to use only one ml. This is why the only address modes are as follows:

- n signed 16 bit immediate word
- rj register direct where j is the register number
- *rj(n) register indirect with offset n

3.1.1.3 Load and Store

As a practical matter the state of the art always seems to make the ratio of memory access time to logic clock period greater than one. In this sense memory accesses stall the process as mControl waits. And so one feature of RISC ul is that memory access capability is restricted to what are called load and store ul.

Load ul can read one byte, or a halfword (two bytes), or a word (four bytes) on each load. In the ul set the data address is the sum of the base address in a register and the immediate halfword n which is part of the ul codeword.

$$\begin{array}{ll} \text{Lp} & *ry(n) \quad rz \quad rz \leftarrow M[ry+n] \\ \text{Sq} & ry \quad *rx(n) \quad M[rx+n] \leftarrow ry \end{array}$$

where p is B (byte), BU (byte unsigned)
H (halfword), HU (halfword unsigned), or W (word)
where q is B (byte), H (halfword) or W (word)

Memory reads (computer register loads) are full word reads maximizing the data transfer rate. Therefore byte and halfword loads require extraction of the byte and halfword from the full word. Later we show how this is achieved with a one ml solution if special hardware is added to the uData Path.

3.1.1.4 Operators

The operator ul operands are registers or immediate halfwords. The immediate halfwords are incorporated into the ul codeword. The operator ul specify the usual arithmetic, logical, and shift opera-

tions. The **CONST** ul are new.

```

CONST+    n  rz      rz  $\leftarrow$  016 ## n
CONST     n  rz      rz  $\leftarrow$  116 ## n
CONSTH    n  rz      rz  $\leftarrow$  n ## 016

```

The immediate n is concatenated with zeros or ones forming a signed 32 bit word. This is how positive and negative constants are stored in registers. Any 32 bit number is constructed with a two ul sequence. Note that n2 must be a positive number.

```

CONSTH  n1 ry      ry_high  $\leftarrow$  n          ry_low  $\leftarrow$  0
OR      n2 ry rz    rz_high  $\leftarrow$  n1 OR 0    rz_low  $\leftarrow$  0 OR n2

```

The compare ul is Scc (set on condition cc). The true/false report is stored in register rz.

```

Scc    sx ry rz      If cc true then rz  $\leftarrow$  1 else rz  $\leftarrow$  0 (form ry    sx
to set status)(cc in Table 2.4)

```

3.1.1.5 Program Control

RISC branch ul test the flag in a register. This is consistent with the Scc compare ul which stores the true or false test report in register rz. Note the simplification: rz replaces the (CISC) status register.

```

Scc    sx ry rz
followed by
BZ     n ry      if ry=0  then pc  $\leftarrow$  pc + n else pc  $\leftarrow$  pc + 4
or
BNZ    n ry      if ry<>0 then pc  $\leftarrow$  pc + n else pc  $\leftarrow$  pc + 4

```

Unconditional jumps are straightforward.

```

JMP    m          pc  $\leftarrow$  pc + m
or
JMP    ry          pc  $\leftarrow$  ry

```

Calling a subroutine has two flavors:

1. Jump to a compiled address (pc + m)

```

CALL  m          r31  $\leftarrow$  pc, pc  $\leftarrow$  pc + m

```

2. Jump to the address in register ry

```

CALL  ry          r31  $\leftarrow$  pc, pc  $\leftarrow$  ry

```

The CISC **CALL** ul saves the address of the next in line ul (the return address) on a stack. The

RISC **CALL** ul avoids the memory access to a stack by saving the return address in register r31. This calling process is also known as link and jump.

There is no **RET** ul because the return address is in r31. This means **JMP r31** implements a **RET** equivalent.

The int (interrupt) register saves the address of the next-in-line ul when exceptions occur.

```

TRAP      int ← pc, pc ← 0
RTE       pc ← int      (return from exception)
HALT      halt
    
```

3.1.1.6 ul Codeword Formats

All codewords are 32 bit words and a six bit opcode allows for 64 ul. The type3 codeword (Table 3.3) operation field allows for increasing the number of ul beyond 64. This is not obvious at this point.

Type1 format: Program control ul with the offset operand m need a 6 bit opcode field and an offset field m occupying the remaining 26 bits.

Type2 format: Operator ul, constant ul, load and store ul, and branch ul with an immediate operand need a codeword with two register fields and the n field. The 6 bit opcode field and two 5 bit register fields use 16 bits. Sixteen bits remain for the immediate field n.

Type3 format: The operator ul need three 5 bit register fields for the operands and a 6 bit opcode field. This leaves 1 bit for the special scc field and 10 bits for the operation field. Status bits NCZV are updated when scc = 1. NCZV status bits are not changed when scc = 0.

TABLE 3.3 ul Codeword Formats

Type1	6	26				
	opcode	m				
Type2	6	5	5	16		
	opcode	rz	ry	n		
Type3	6	5	5	5	1	10
	opcode	rz	ry	rx	scc	operation

3.1.2 The First Step from ul to User Data Path

The process evolving the RISC user data path from the RISC ul set separates into two major steps. First the RISC ul operations, operands, address modes, and register transfer statements create the need for some set of combinational circuit blocks and sequential circuit blocks interconnected by

data buses. We call this the preliminary user data path. This first part of the process evolving the RISC user data path (Figure 3.1 on page 99) is similar to the process used to evolve the CISC user data path in Figure 2.13 on page 29 to Figure 2.13 on page 29.

The second major step modifies the preliminary user data path at a detailed circuit level. The modifications result in execution of RISC ul in one computer cycle. In other words ml lists consist of one ml. Strictly speaking the one computer cycle goal is not met for the RISC ul that access memory. We now proceed to implement the first major step.

3.1.2.1 Bus

We arbitrarily select a 32 bit word size for the RISC computer. Performance dictates that the 32 bits are available simultaneously when a word is read from any source. In hardware terms this means 32 wires, one wire per bit, provide parallel access to all word bits. All the bits in a source word are connected to a destination simultaneously.

Parallel access to two sources requires two source busses x, y. One result requires one destination bus z.

3.1.2.2 Register File

The operands rx, ry are implemented by a set of 32 bit registers. The 32 bit size is consistent with 32 bit data busses, and with our arbitrary decision to design a 32 bit machine.

The number of registers is not defined in Table 3.1 nor do we have a theoretical basis for picking a number. We selected 32 when we sized the rx, ry, rz fields at five bits (Table 3.2).

The ul **ADD rx ry rz** reads two registers rx and ry in parallel (at the same time), calculates the result, and writes one register rz at another time. The 32 32 bit registers are implemented by a register file.

3.1.2.3 Memory

A memory is required to store programs and their data. The computer treats random access memory as a mass of word registers that is accessed only one word at a time.

Reading or writing a memory word requires an address for that word. The address is stored in the memory address register (mar). The mar presents a stable address to the memory for the duration of a memory cycle. Writing a memory requires data to store in addition to the address. The data is placed in the memory buffer register (mbr) prior to initiating a memory access. The mbr presents stable data to the memory data input during a memory write cycle.

The instruction register (ir) is loaded with ul read from memory. This is why the memory q outputs (mem) are connected directly to the ir inputs (Figure 3.1 on page 99). We did not choose to do this in the CISC design.

3.1.2.4 Program Counter and Instruction Register

Any user instruction ul must be fetched from memory, stored somewhere in the data path, and executed. Fetching requires knowing the address of every ul word in a program list. The easiest way to manage the list of ul is to store the ul words sequentially in memory. The management process is as follows. A counting register is loaded with the address of the first ul word in the program. This stand alone counting register is called the (user) program counter or pc. The pc is incremented after each

fetch of a ul from the ul program list. In other words the pc always points to the next item in the program list. Fetched ul are stored in the instruction register (ir).

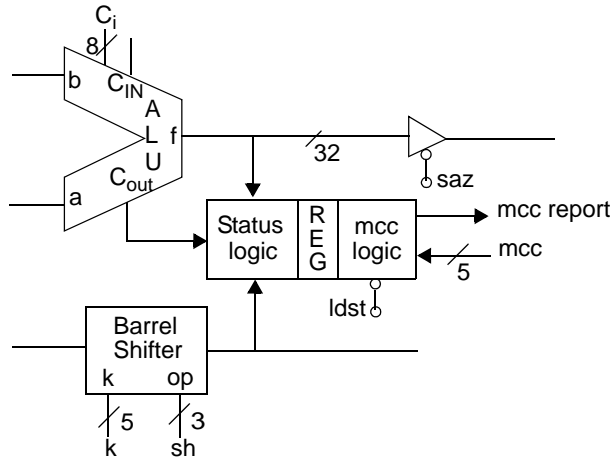
3.1.2.5 ALU and Shifter

Arithmetic and logical operators used by the ul are readily implemented by an arithmetic logic unit (alu). A separate shifter is required because an alu does not implement shift operators. The CISC one-bit-at-a-time shifter solution is not applicable in a RISC data path because execution with one ml implies shifting k bits in one computer cycle. We use two parallel paths from the x, y source busses to the destination bus z in lieu of the single path cascading the alu and shifter. One path is for the alu and the other path is for a barrel shifter. The barrel shifter shifts k bits at a time.

3.1.2.6 Temporary Register

Executing a ul in one computer cycle implies there is no need for a temporary register holding address or data for a subsequent ml in an ml list to use.

FIGURE 3.1 RISC uData Path - First Step (I)



3.1.2.7 Status

The Scc and **BZ**, **BNZ** ul functions do not explicitly require a status register however, as will be shown, the Scc ul needs the mcc_report. So status and mcc logic creating the mcc_report are included in the data path. The mcc logic implements the cc equations (Table 3.2). Selection of an equation outputs the corresponding mcc_report. Five encoded lines from mControl implement mcc_report selection.

The ul **ADDC**, **SUBC**, **SUBRC** require a status register for the carry bit C. Therefore we include a NCZV status register even though the need for NCZV bit storage is not obvious here.

3.1.2.8 Other Items

Transfer: Two source buses x, y deliver the source data in parallel to the alu or other destinations.

Results are delivered via one destination bus z. Direct connections from x to z, and y to z busses allow for faster data transfer for data that does not need processing by the alu or shifter.

Since more than one output may now connect to the z bus a tri state multiplexer is needed.

Zero: The number zero is still needed. We implement this with hardwired register rzero (r0).

Stack Pointer: The stack pointer sp is omitted because a memory stack is not used in RISC.

User data path: The blocks for individual functions assembled together constitute the uData Path (Figure 3.1 on page 99).

Data Path Control Lines: Signal lines shown with each data path block control the user data path (Figure 3.1 on page 99). The signal lines originate in the mControl microcontroller (Figure 3.30 on page 128) which is explained later.

3.1.3 The Second Step from ul to User Data Path

The RISC goal is an ml language that allows one ml to represent any ul. The RISC goal is execution of any ul in one computer cycle. We start with the four CISC ml (**mALU**, **mMOV**, **mBR**, **mNOP**) we used to microprogram the CISC ul set. We find that our experience with the four ml gives us a running start. Our method is to write the ml list for each RISC ul using the CISC ml set and ask a question. What changes do we make to the four CISC ml or to the RISC user data path to achieve the goal of execution in one computer cycle?

3.1.3.1 RISC ml Word Format

We need to track changes made to the CISC ml fields. To this end we merge fields from the four CISC ml **mALU**, **mMOV**, **mBR**, and **mNOP** (Table 2.23) into one ml word. We use this word as a preliminary RISC ml word format which we proceed to modify as we pursue our goal.

FIGURE 3.2 Merged fields from CISC ml **mALU**, **mMOV**, **mBR**, **mNOP**

moff	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	bop
------	-----	-----	-------	----	-----	----	-----	------	------	------	-----

3.1.3.2 Branch to fetch_ul

Every CISC ml list ends with the **mBR** ml.

mBR bop un fetch_ul

The need for using the above **mBR** or any other branch ml is eliminated if we add a jmp_to field in every ml codeword (ml not ul). This is practical to do because the one ml for each ul goal implies a very small RISC mROM. The jmp_to field may only have seven bits for example. This is small compared to the thousands of mls in the CISC ml lists stored in the CISC mROM. The seven bit jmp_to field replaces the 16 bit moff field in the preliminary RISC ml word. The CISC bop are incpc, incsp, decsp, setc, clrc. The RISC ul set omits **CLRC** and **SETC** ul that set and clear the carry. And, there is no stack pointer in the RISC data path. This leaves incpc as the one remaining bop. We change

bop to misc(ellaneous). The RISC mI word is changed as shown below.

TABLE 3.4 Eliminate mBR mI by adding jmp_to field to every mI codeword

Before:											
moff	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	bop
After:											
jmp to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc

incpc

FIGURE 3.3 Program Control Logic

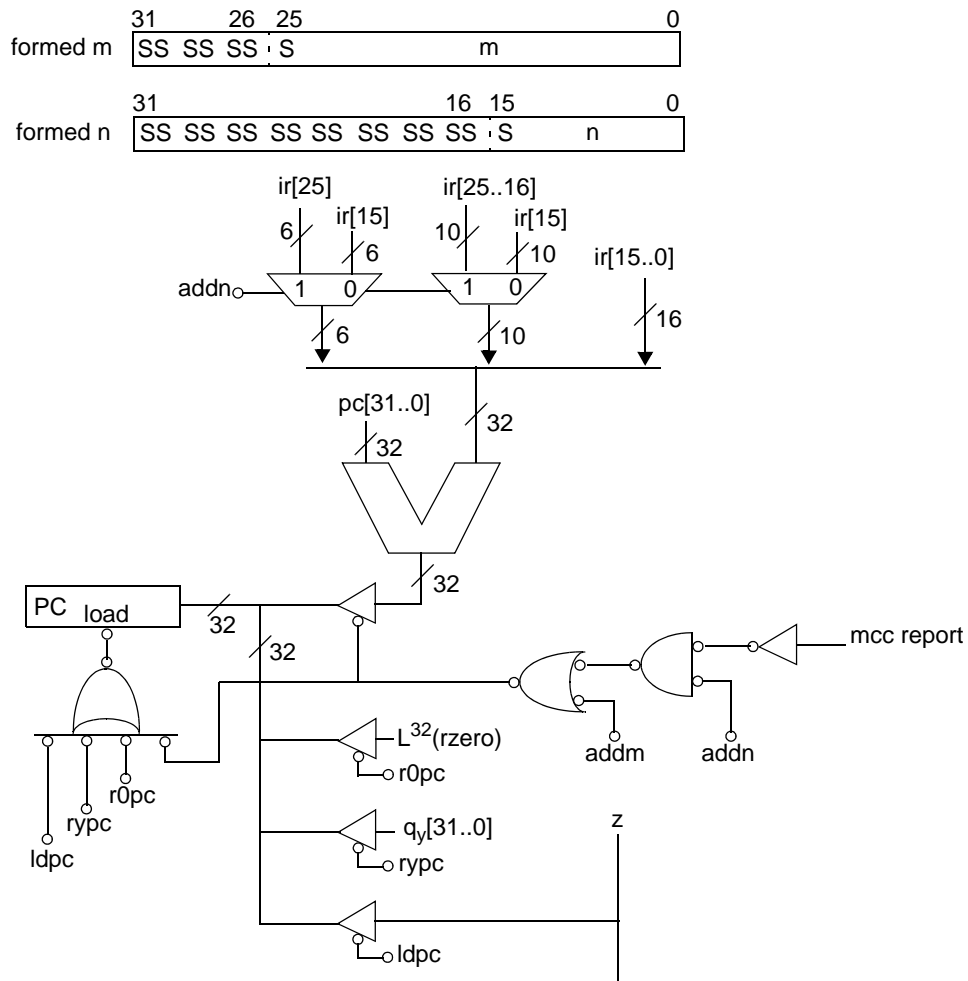


FIGURE 3.4 Program counter (PC)

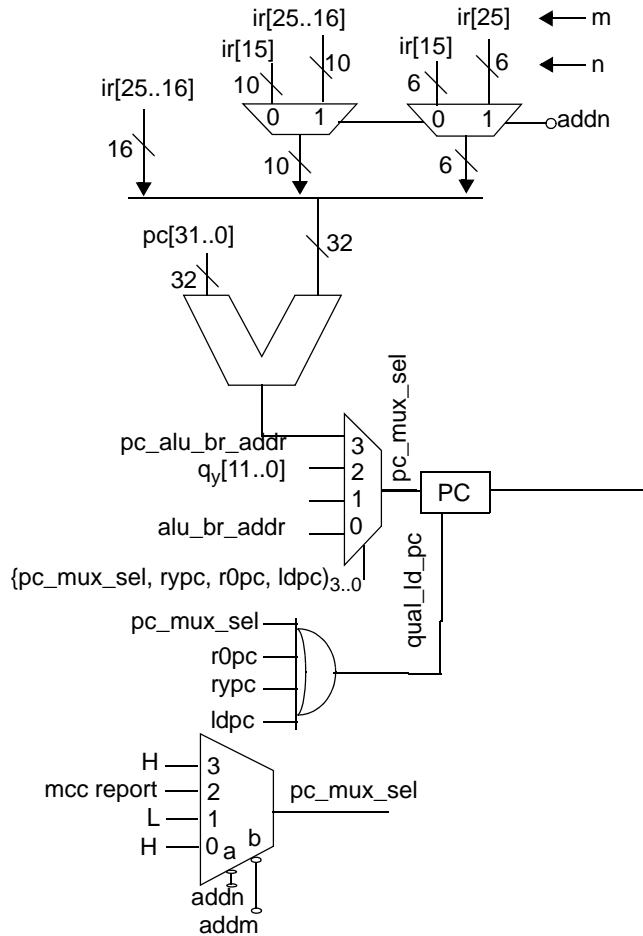


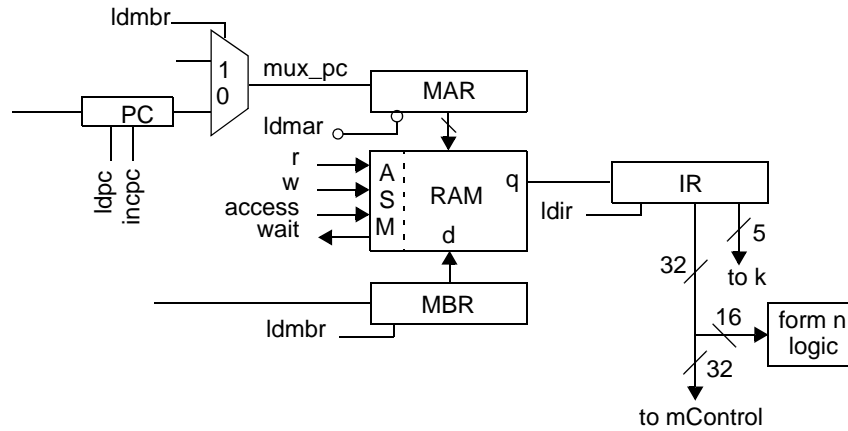
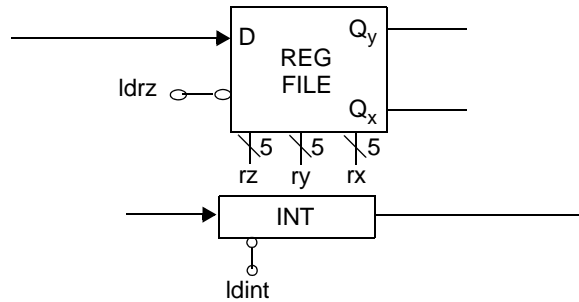
FIGURE 3.5 Instruction memory (IM) and instruction register (IR)**FIGURE 3.6** Register File (RF))

FIGURE 3.7ALU p

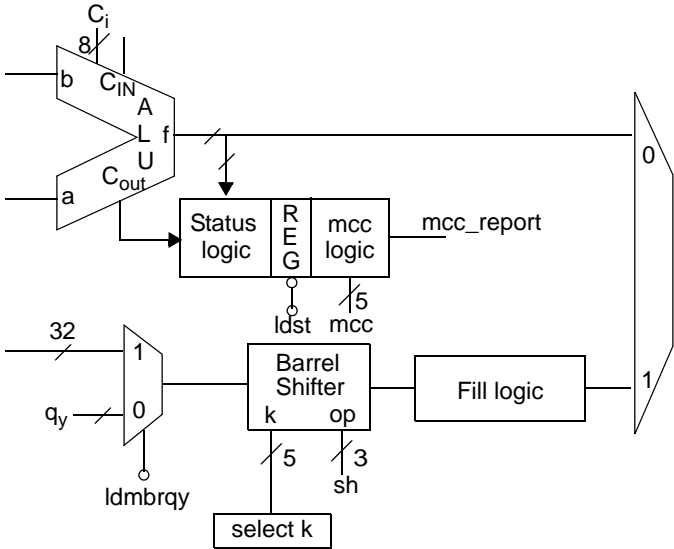


FIGURE 3.8SelectK

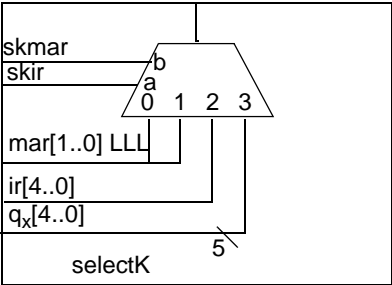


FIGURE 3.9 Constants - from IR to x and y Busses

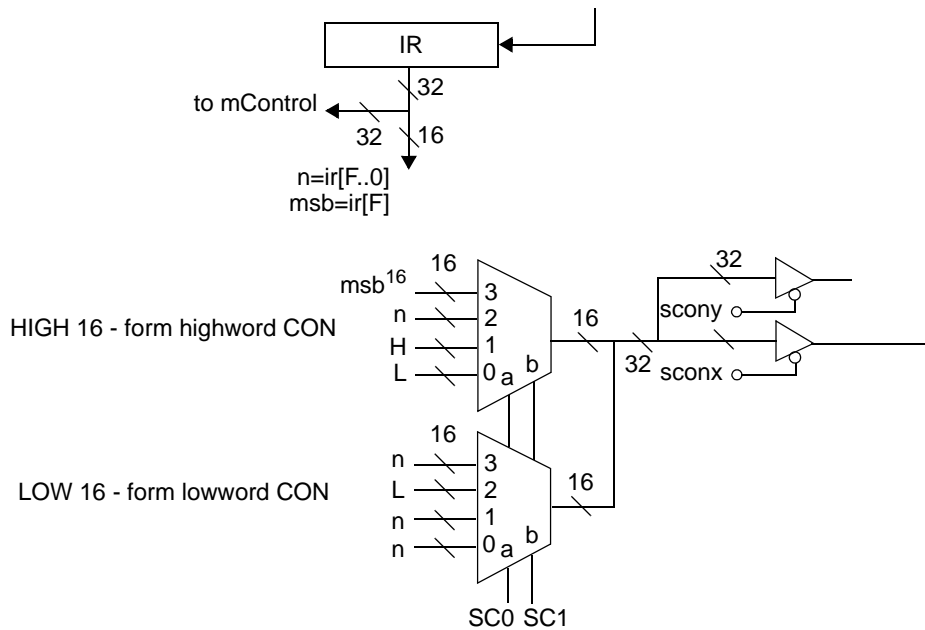
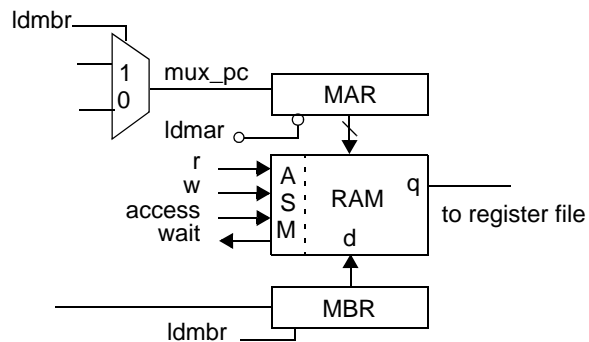


FIGURE 3.10 Data memory (DM)



The diagram illustrates the internal structure of a 32-bit ALU. It features a Barrel Shifter, a 4-to-1 multiplexer, and a 3-to-1 multiplexer. The Barrel Shifter takes a 32-bit input and produces a 32-bit output, with a 5-bit shift amount (sh) and a 3-bit operation code (op). The 4-to-1 multiplexer selects between four 8-bit inputs (a, b, and two others) based on two select lines (L1, L0). The 3-to-1 multiplexer selects between three 8-bit inputs (a, b, and one other) based on one select line (L). The final 32-bit output is zero-extended (sbz) to 32 bits.

A k-bit shift operation is executed by the barrel shifter. The barrel shifter is microprogrammed for the desired shift operation by the sh ml field. The RISC shift ul specify k in the sx operand. Sx is an immediate word n or a register rx. Therefore the five bits representing the number k are in the n ul field stored in the instruction register ir or the five lsb in a source register rx. We show these facts using the ml and RTL languages.

Rx_{4..0} is the five bit field consisting of rx bits 4 to 0. Field n (type2, Table 3.3) is 16 bits wide. However, bits F to 5 are in effect zeros when n represents k because k > 31 serves no purpose. This is why we are only concerned with bits 4 to 0 and why we ignore bits F to 5. When sx is an iw, field n = k, so that with the ul in the ir we have

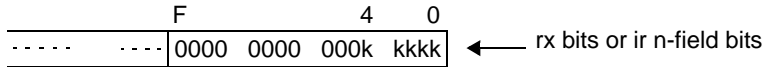
```

SLL  k ry rz:      (k is field n in a Type2 uI)
mALU sll ir4..0 ry rz nop      (mALU op sx sy dz rw)

```

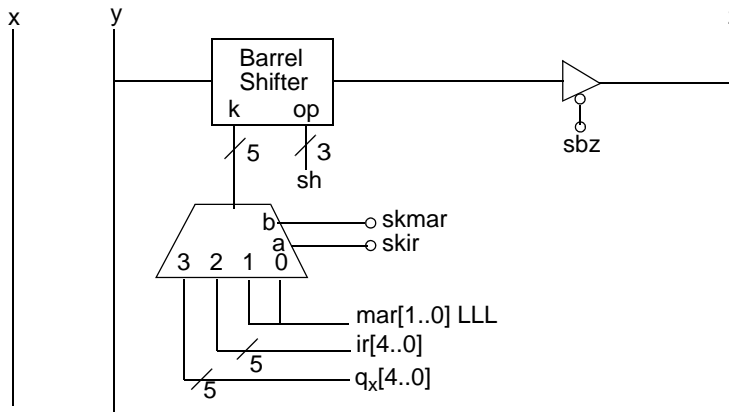
Ir_{4..0} is the five bit field consisting of ir bits 4 to 0.

FIGURE 3.12



In a CISC machine there would be a k-register for the k bits driving the shifter. We would load the k-register from the rx or the ir k fields. However, performing the k-register load requires use of an extra ml which conflicts with our RISC one ml per ul goal.

FIGURE 3.13 Source bits Specify Shift Parameter k



Additional hardware replaces ml. One hardware solution is to wire the rx and ir k fields (field n sub-fields) to a five bit wide 2 x 1 multiplexer. This means a new control line we call **skir** is required to select k from rx or ir (Figure 3.4 on page 102). Skir is a miscellaneous function we assign to the misc field because the bits representing k do not appear on the x or y busses. Skir is derived from the phrase "select k value from the ir."

Figure 3.13 on page 107 also shows a third source for k and the corresponding control line skmar. This is explained in Section 3.1.3.6 Load.

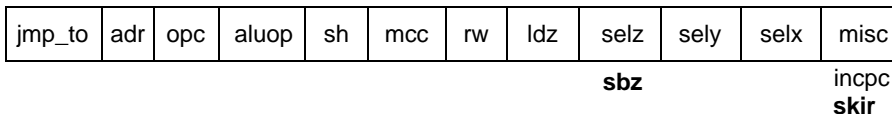
The barrel shifter path which is parallel to the alu path adds a new choice to selz.

```

skir → select k from the ir
sbz  → connect barrel shifter output to the z bus

```

FIGURE 3.14



3.1.3.4 Scc

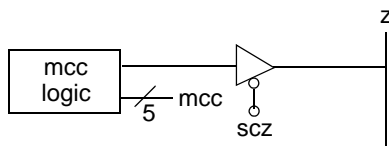
The set on condition ul is more complex. We write one ml list for Scc in pseudo code revealing the functions Scc performs. The sub operation sets up status for decision making.

```

Scc  sx ry rz
      mALU sub sx ry nop nop          (mALU op sx sy dz rw)
If cc is true then
      mMOV 1 rz nop                   (mMOV s dz rw)
else (cc is false)
      mMOV 0 rz nop                   (mMOV s dz rw)

```

FIGURE 3.15Mcc_Report - from Status to z bus



The mcc report resulting from execution of the **mALU** ml has value 1 if cc is true and value 0 if cc is false. The pseudo code shows that the if then else-statement is implemented if the mcc report is loaded into rz. This means the mcc_report is a source that needs to be placed on the z bus via a tri state gate (Figure 3.15 on page 108). We change the pseudo code to match the new hardware. The if then else statement is replaced by the mcc_report supplying 1 or 0.

```

Scc  sx ry rz
      mALU sub sx ry nop nop          (mALU op sx sy dz rw)
      mMOV mcc_report rz nop          (mMOV s dz rw)

```

The new list still has more than one ml. Can we merge the two ml and reduce this to one ml? The answer is yes if we add a third source field to the **mALU** ml. A third source (the mcc_report in this case) available in parallel is necessary because the first two sources feed the alu for the sub operation.

```

mALU op sx sy s3 rz nop              (revised)
mALU sub sx ry nop nop               (two sources for sub)
mMOV  mcc_report rz nop               (third source for result)

```

Now we merge **mALU** and **mMOV** so that the ul Scc executes in one computer cycle. The hardware change placing the mcc_report on the z bus via a tri-state gate requires the scz choice in the selz field.

```

mALU op  sx sy s3  dz  rw
mALU sub sx ry  mcc_report rz  nop      (ry - sx sets status)
                                           (rz ← mcc_report)

scz → connect mcc_report to the z bus
skir → select k from the ir

```

FIGURE 3.16

jmp_to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
								sbz			incpc
								scz			skir

3.1.3.5 CONST

The sixteen bit constant *n* is part of the ul codeword (type2, Table 3.3). This means we need hardware to read the instruction register low halfword (the *n* field) and hardware generating the other halfword of 16 zeros or ones. The 32 bit constants formed from *n* for various ul are as follows.

```

CONST+ n rz
      mMOV    016##n  rz  nop      (mMOV s dz rw)

CONST  n rz
      mMOV    116##n  rz  nop      (mMOV s dz rw)

CONSTH n rz
      mMOV    n##016  rz  nop      (mMOV s dz rw)

```

Immediate word from *n*

```
nsigned  msb16##n
```

Each of the **CONST** ul place one of three formed constants on the *x* bus or the *y* bus. It turns out there is a fourth constant. The operand *sx* used by other ul is *rx* or a signed *n* (Table 3.1). This implies a need for a 4 x 1 mux to select one of the four formed constants (Figure 3.18 on page 110). The mux output feeds tri state gates whose outputs place the selected constant on the *x* bus and, or the *y* bus (Figure 3.18 on page 110). In this way *n*+ or *n* or *n* high or a sign extended *n* are placed on source busses *x* and, or *y*.

The bits *sc*1, *sc*0 (Figure 3.18 on page 110) select the formed *n* required by **CONST+**, **CONST**, **CONSTH**, or *n* signed. Select lines *scon**x*, *scon**y* put the programmed constant on the *x*, *y* busses.

TABLE 3.5

uI	Entry in mI field	Select		Bit	Output to source bus
		sc	sc	hi16	lo16(hex)
CONST+	scon+	L	L	0000	n

TABLE 3.5

CONST	scon-	L	H	FFFF	n
CONSTH	sconh	H	L	n	0000
nsigned	sn	H	H	msb^16	n

FIGURE 3.17

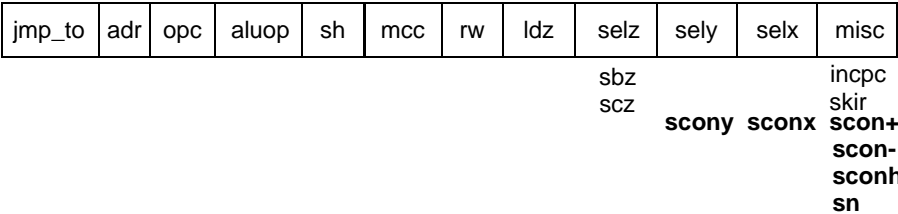


FIGURE 3.18 Constants - from IR to x and y Busses

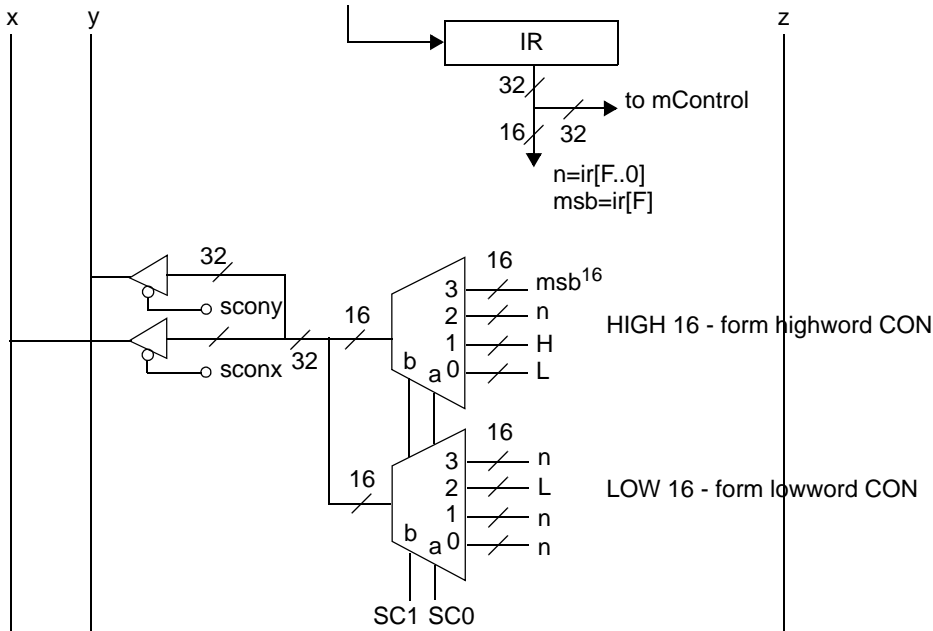


TABLE 3.6

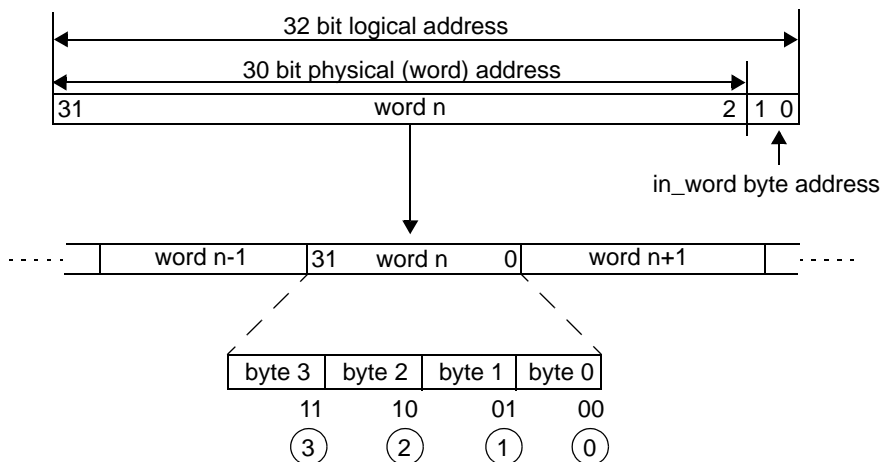
INPUT		SC1	SC0	HIGH 16	LOW 16
scon+	CONST+	L	L	0 ¹⁶	n
scon-	CONST-	L	H	1 ¹⁶	n
sconh	CONSTH	H	L	n	0 ¹⁶
sn	n_SIGNED	H	H	msb ¹⁶	n
lmbreq	n_SIGNED	H	H	msb ¹⁶	n

3.1.3.6 Load

Load executes in two computer cycles. The second cycle is a memory access reading data (Figure 2.23 on page 44). Loads suffer a performance penalty unless a pipeline is used. [Pipelines are explained in Chapter 4.] When load executes, the address in the mar is the address of the word holding the data (Figure 3.19 on page 111). This word becomes available at the memory ram q output. Two of the three data types (halfwords and bytes) require alignment to the destination register least significant bit (lsb). The shifter aligns the word fetched from memory to bit 0 with 24, 16, or 8 bit shifts. The number of bits k to shift is the in_word byte address represented by mar[1..0] (Figure 3.19 on page 111) multiplied by 8. Why?

$$k_align = mar[1..0]LLL = a1a0LLL$$

FIGURE 3.19 Physical Memory Address



Two bytes of a halfword data type, and three bytes of a byte data type need to be replaced (filled)

with zeros or copies of the msb after alignment is implemented by shifting. Fill is required because those bytes are parts of other halfword data and byte data. The aligned data is correctly filled by adding a fill-with-zero-or-msb logic circuit at the barrel shifter output (Figure 3.21 on page 114). Word loads do not need alignment or filling. This is why the second ml in the **LW** ml list is a straightforward move.

```

LW *ry(n) rz
      mALU add n ry mar r          (mALU op sx sy dz rw)
      mMOV nop mem rz nop          (mMOV mop s dz rw)

```

The second ml in the ml lists for **LH** and **LB** ul is an mALU whose op operand is the desired shift command and whose misc field specifies the shift and the fill. The special operators snop, srlb, srab, srlh, and srah specify the correct shift and fill operations.

Halfword and byte loads are aligned to the lsb of the destination register. Shifted unsigned halfwords and bytes are filled with zeros. Shifted signed halfwords and bytes are filled with the most significant bit (msb). Note: the second ml executes in (time) parallel with the memory read access (Figure 2.23 on page 44). This avoids increasing the ul execution time to three computer cycles.

```

LHU *ry(n) rz
      mALU add n ry mar r          (mALU op sx sy dz rw)
      mALU srlh mem nop rz nop    (mALU op sx sy dz rw)

LH  *ry(n) rz
      mALU add n ry mar r          (mALU op sx sy dz rw)
      mALU srah mem nop rz nop    (mALU op sx sy dz rw)

LBU *ry(n) rz
      mALU add n ry mar r          (mALU op sx sy dz rw)
      mALU srlb mem nop rz nop    (mALU op sx sy dz rw)

LB  *ry(n) rz
      mALU add n ry mar r          (mALU op sx sy dz rw)
      mALU srab mem nop rz nop    (mALU op sx sy dz rw)

```

```

skmar  → select k from the mar to shift the data (Figure 3.13 on
fillb0 → fill bytes with zeros
fillbm → fill bytes with msb
fillh0 → fill halfword with zeros
fillhm → fill halfword with msb

```


FIGURE 3.20

jmp_to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
								sbz			incpc
								scz			skir
									scony	sconx	scon+
											scon-
											sconh
											sn
											fillb0
											fillbm
											fillh0
											fillhm

srlb assembles srl, sbz, fillb0
 srab assembles sra, sbz, fillbm
 srlh assembles srl, sbz, fillh0
 srah assembles sra, sbz, fillhm

Important Note: each fill-- asserts skmar.

TABLE 3.7

INPUT	
fillhm	fill high word with MSB
fillh0	fill high word with 0's
fillbm	fill high 3 bytes with MSB
fillb0	fill high 3 bytes with 0's

FIGURE 3.21 Load Word, Halfword, and Byte

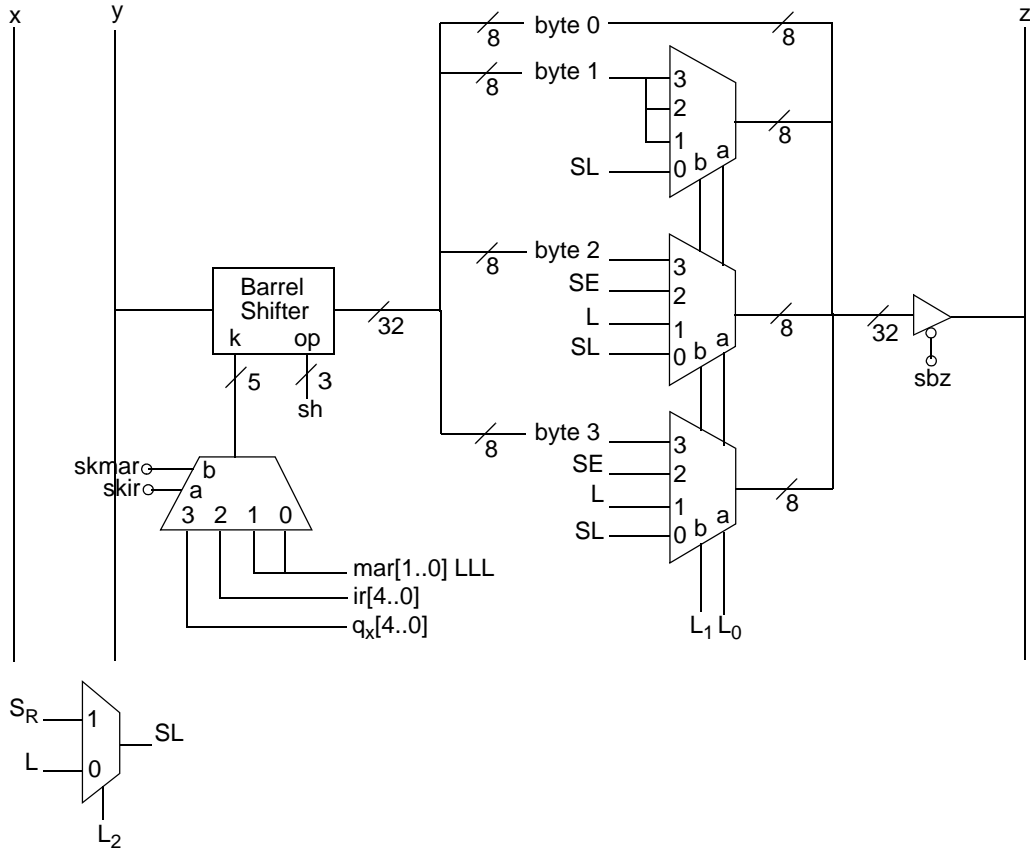


FIGURE 3.22 Store Word, Halfword, and Byte

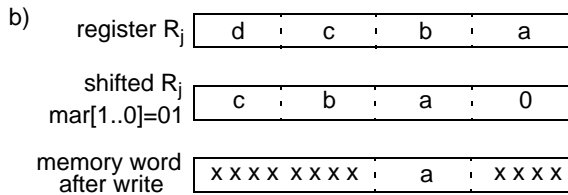
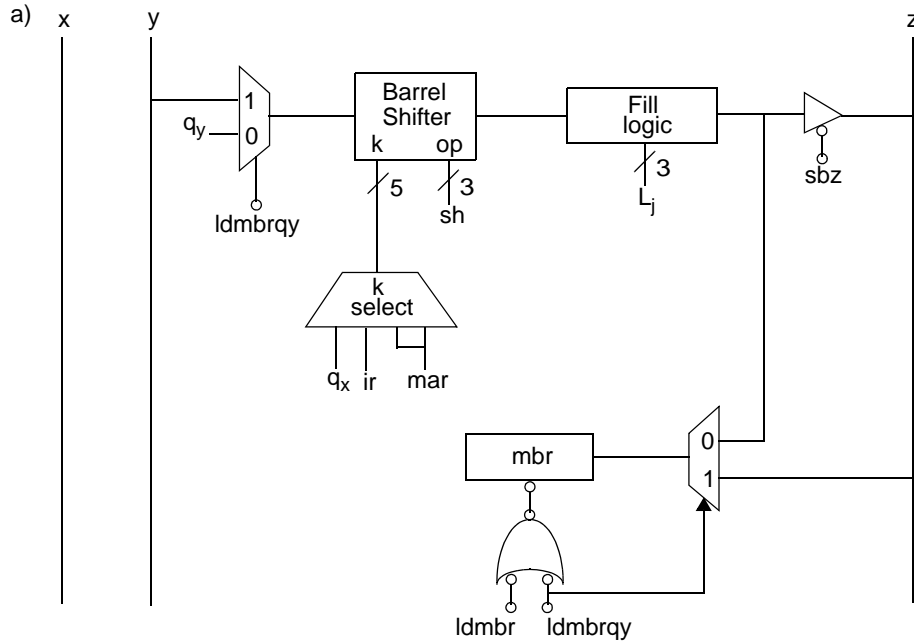


TABLE 3.8

byte	3	2	1	0	
word					3
$b_F \dots b_F b_F$ halfword					2
$0 \dots 01$ halfword					1
$b \dots b \quad b$ byte					0
$0 \dots 01$ byte					0

TABLE 3.9

L ₂	L ₁	L ₀	INPUT
x	H	H	nop
x	H	L	fillhm
x	L	H	fillh0
H	L	L	fillbm
L	L	L	fillb0

3.1.3.7 Store

Executing a store requires calculating and loading the address in the mar, loading the data in the mbr, and accessing the memory to write. This implies that store executes in three computer cycles. The third cycle, accessing memory and writing data (Figure 2.23 on page 44), stalls the cpu during this third cycle. The immediate question is whether or not the first two cycles can be merged into one cycle. Can we merge the load-the-address-in-the-mar and load-the-data-in-the-mbr functions into one ml?

Store copies data from a register to memory. Halfword and byte data need to be aligned to bit 0, or 8, or 16, or 24 of the mbr register. (Word data is already aligned to bit 0.) The uData Path barrel shifter can align the data to bit 0, 8, 16, or 24 with 0, 8, 16, or 24 bit left shifts before loading it into the mbr. The shift is specified by the type of ul and the in_word byte address (Figure 3.13 on page 107 and Figure 3.19 on page 111).

The aligned word is loaded into the mbr. The bytes actually written into memory are specified by the ul and the in_word address. This information is translated into byte access lines. Four byte access lines wj specify the memory bank(s) to write (Figure 3.24 on page 118). This is why filling is not needed on store.

```

Sq ry *rx(n)                                (q is W or H or B)
  mALU add rx n mar nop
  mALU sllq nop ry mbr w

```

In this ml scheme the special operand sllq would have to assemble the shift op sll and skmar, and w requires a third computer cycle.

The two ml are merged into one ml when we add special hardware (Figure 3.24 on page 118). The first special hardware mux connects the ry output qy to the shifter to shift qy by 8, 16, or 24 bits according to the in_word byte address (selected by skmar). The second special hardware mux connects shifted ry to the mbr. The mnemonic ldmbry is a new misc field operator. Ldmbry controls the two multiplexers and mbr loading well as specifying sn to form a 32 bit signed word from ir field n.

At the same time a parallel action takes place. The ALU calculates the address ry+n and loads the mar. The two destinations are specified by the marmbr operand in the **mALU** ml as is shown below. The ldmar field operator loads the mar.

ldmbrqy → load shifted ry into mbr bypassing the busses in parallel with normal data path action. Also forms 32 bit signed word from ir field n and asserts skmar.

marmbr → activates ldmar and ldmbrqy

Sq ry *rx(n) (q is W or H or B)
mALU add rx n marmbr w (**mALU** op sx sy dz rw)

FIGURE 3.23

jmp_to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
								sbz			incpc
								scz			skir
									scony	sconx	scon+
											scon-
											sconh
											sn
											fillb0
											fillbm
											fillh0
											fillhm
											ldmbrqy

marmbr assembles sll, ldmar, ldmbrqy

Important Note: ldmbrqy asserts sn and skmar.

FIGURE 3.24 Store Word, Halfword, and Byte

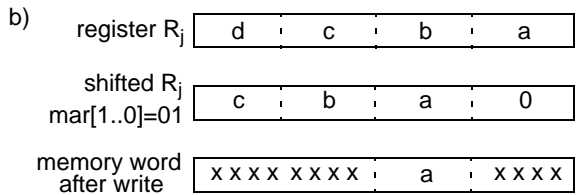
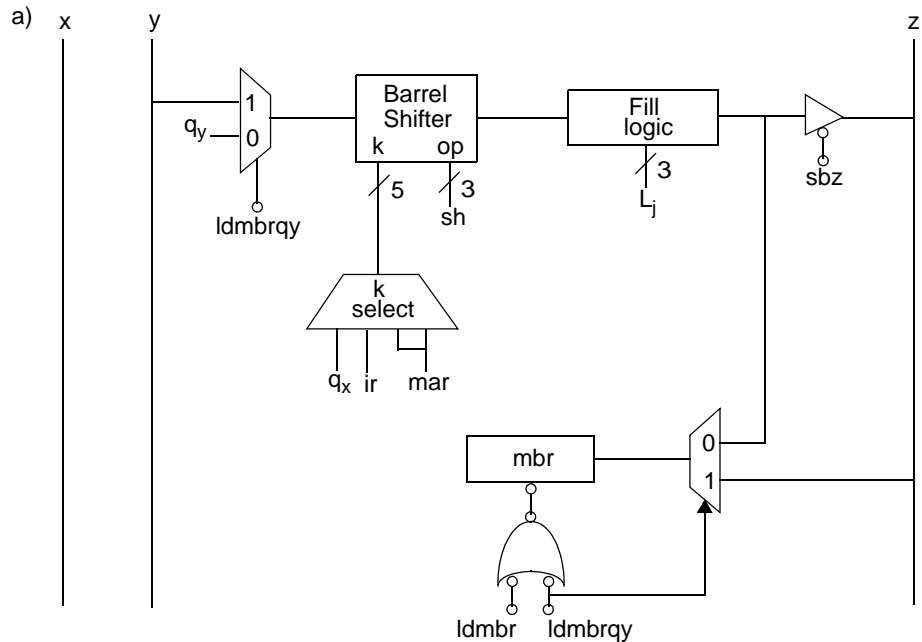


TABLE 3.10

uI	mar[1..0]	W ₃	W ₂	W ₁	W ₀
sw	00	L	L	L	L
sh	00	H	H	L	L
	10	L	L	H	H
sb	00	H	H	H	L
	01	H	H	L	H

TABLE 3.10

	10	H	L	H	H
	11	L	H	H	H

(L = write)

3.1.3.8 Program Control

Program control ul test a register's contents and execute a branch if the test result is true. The test requires passing the register word through the ALU to set status. The branch requires the ALU to form pc+n. Therefore execution in one computer cycle implies use of two ALUs. One ALU processes the test subtraction; at the same time a second ALU performs the addition forming the address.

The branch ul (Table 3.1) imply forming a 32 bit signed word from the 16 bit n field prior to addition . Branch ul pseudo code makes the following points.

```

BZ n ry    if ry=0 then pc ← pc + n else pc ← pc + 4
             mALU sub r0 ry nop nop
             if     Z= 1    (ry = 0)
             then  mALU add msb^16##n pc pc nop      (mALU op sx sy dz rw)
             else  mNOP

```

Call and jump ul imply forming a 32 bit signed word from the 26 bit m field prior to addition. Call and jump ul pseudo code makes the following points.

```

JMP m
             mALU add msb^6##m pc pc nop              pc ← pc + m
                                                         (mALU op sx sy dz rw)

CALL m
             mMOV pc r31 nop                             r31 ← pc, pc ← pc + m
             mALU add msb^6##m pc pc nop              (mMOV s dz rw)
                                                         (mALU op sx sy dz rw)

CALL ry
             mMOV pc r31 nop                             r31 ← pc, pc ← ry
             mMOV ry pc  nop                          (mMOV s dz rw)
                                                         (mMOV s dz rw)

```

These ul add n or m to the pc, or store zero or ry in the pc. This needs to be done in one computer cycle. CALL also needs to load the pc into r31 before taking a new value into the pc. The user data path is free to load the pc into r31 via the busses when we add special addition hardware implementing 32 bit word formations from the m or n ir fields as well as the address additions pc+n, pc+m. And so we need hardware (Figure 3.26 on page 121) to form pc+n, pc+m and to execute pc ← ry, pc ← r0, pc ← pc+n, pc ← pc+n.

The **CALL** ml lists with two mMOV's must merge into one ml. Merging implies adding an operand to the mMOV ml. We call this operand mop for reasons that will become clear in a moment.

```

mMOV s dz rw
becomes
mMOV mop s dz rw

```

The four operations loading new values into the pc imply a need for four mops in the **mMOV** oper-

and. We name the mops addn, addm, rypc, and r0pc. The revised microcode follows.

```

BZ  n ry
      (mALU  op  sx sy s3  rz nop)
      mALU  sub r0 ry addn pc nop

JMP  ry
      (mMOV   mop s  dz rw)
      mMOV   nop ry pc  nop

JMP  m
      (mMOV   mop s  dz rw)
      mMOV   addm nop nop nop           (26_bit_offset selected)

CALL  m
      (mMOV   mop s  dz rw)
      mMOV   addm pc r31 nop

CALL  ry
      (mMOV   mop s  dz rw)
      mMOV   rypc pc r31 nop

TRAP
      (mMOV   mop s  dz rw)
      mMOV   r0pc pc  int  nop

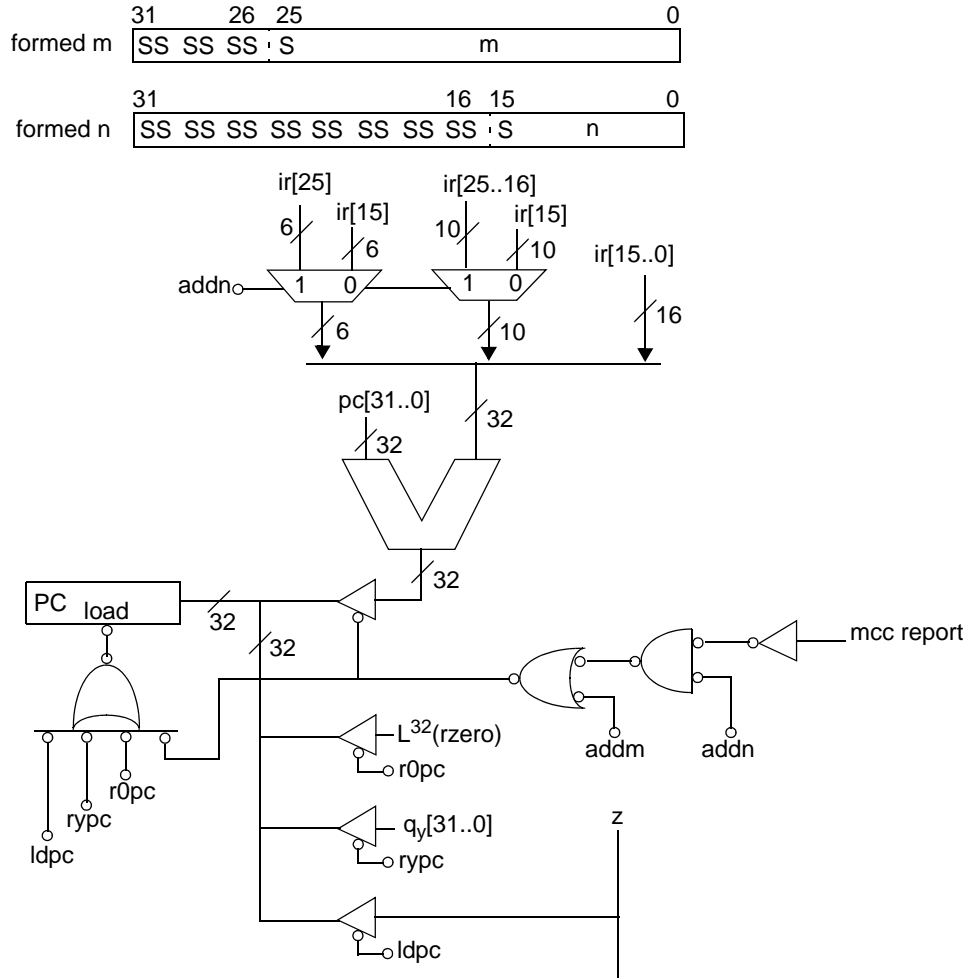
RTE
      (mMOV   mop s  dz rw)
      mMOV   nop  int pc  nop

addn  →   if mcc_report=1 then pc ← pc + n
addm  →   pc ← pc + m
rypc  →   pc ← ry
r0pc  →   pc ← 0

```

FIGURE 3.25

jmp_to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
								sbz			incpc
								scz			skir
									scony	sconx	scon+
											scon-
											sconh
											sn
											fillb0
											fillbm
											fillh0
											fillhm
											ldmbrqy
											addn
											addm
											rypc
											r0pc

FIGURE 3.26 Program Control Logic**3.1.3.9 uData Path**

The RISC uData Path is set up to implement an ml by asserting the group of control lines specified by the bits in the ml_word representing the ul. The RISC ml_word is partitioned into fields as shown below.

FIGURE 3.27

jmp_to	adr	opc	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
--------	-----	-----	-------	----	-----	----	-----	------	------	------	------

Each ml_word field activates a subset of user data path control lines. The lines activated by the ml_word fields set up the user data path. After setup is complete the ml functions are executed in one computer cycle when the mCtrl ASM steps through its states (Figure 3.30 on page 128).

[Reminder: the ASM executes one ml at a time.]

uData Path control lines form groups by function. For example one functional group is the set of lines that activate tri state gates connecting register outputs to the x bus. This typical group of encoded control lines is represented by a field of bits we name selx (select x). The rest of the control lines in Figure 3.4 on page 102 form other functional groups. Each of these other groups are assigned to fields (Table 3.11).

In section 2.1.5 Microinstructions under operators we explained why the CISC rx and ry fields were omitted from the ml_word. Omission allows us to store only generic ml lists in the mROM. For the same reason the rx, ry, and rz fields are also omitted from the RISC ml_word. The ul bit lines representing rx, ry, and rz are wired directly to register file inputs.

The ir bit lines, representing operands m and n, are wired directly to the program control logic (Figure 3.26 on page 121). Furthermore, the scc bit in the type3 RISC ul (Table 3.3) is wired into the status logic to activate ldst when the scc bit is one and a type3 ul is in the ir.

The same ir bit lines represent parts of operands m, n as well as the scc bit and the three operands rx, ry, and rz. Since no ul uses more than one set of operands this is not a problem.

Next, we build the 44 bit ml_word shown in Table 3.12. The field bits total to 42. Two spares increase the word width to 44 bits which is a multiple of four (chips are 1, 4, or 8 bits wide). If 8 bit wide chips are more practical the resulting 48 bit ml_word has 6 spares.

This ml_word is consistent with the ml_word derived from the fields of the four CISC ml. The opc field is deleted because the four ml are merged into one ml_word. The merger makes this ml_word wider than the CISC ml_word.

Table 3.13 provides a recapitulation of the source figures for fields and the new RISC control lines.

TABLE 3.11 RISC User Data Path Fields

Reference	Figure 3.4 on page 102
Field	Resources
rx	5 encoded lines select 1 of 32 registers
ry	5 encoded lines select 1 of 32 registers
rz	5 encoded lines select 1 of 32 registers
	Register outputs to x,y busses
selx	3 encoded lines select 1 of 6 register outputs to the x bus (nop, int, rx, pc, r0, con, mem)
sely	3 encoded lines select 1 of 6 register outputs to the y bus (nop, int, ry, pc, r0, con, mem)
	Result to z bus

TABLE 3.11 RISC User Data Path Fields

selz	3 encoded lines select 1 of 3 function outputs or 1 of 2 transfer outputs (nop, saz, sbz, scz, sxz, syz)
	Load result on z bus into register
ldz	3 encoded lines select 1 of 6 registers to load from the z bus or mem (nop, int, rz, pc, ir, mbr, mar)
	Shift data
sh	3 encoded lines select shift functions (nop, sll, sra, srl)
	Process data
aluop	8 lines select alu functions
	Branch
mcc	5 lines select mcc_report cc equation
jmp_to	7 lines provide the jump to address
adr	1 line selects next address
	Other fields
rw	2 encoded lines select read or write (nop, r, w)
misc	4 encoded lines select various actions (nop, incpc, skir, scon+, scon, sconh, sn, fillb0, fillbm, fillh0, fillhm, ldmbrrq, addn, addm, rypc, r0pc)

TABLE 3.12 RISC User Data Path Control Line Field Encoding

2	2	22111111	1	1	0	0
BA98765	43	2	10FEDCBA	98765432	10	FEDCBA 987 654 3210
jmp_to	adr	spare	aluop	sh	mcc	rw ldz selz sely selx misc
next mI	0	jmp_to	add	nop	un	nop nop nop nop nop
address	1	mI_start	addc	rotl	see cc	r ldmbrr saz incpc
0 to 127			+1	sll	in	w ldmar sxz smemy smemx skir
			-1	sra	Table 3.6	ldrz syz sr0y sr0x scon+
			neg	srl		ldir sbz sryy srxx scon-

TABLE 3.12 RISC User Data Path Control Line Field Encoding

			sub					scz	scony	sconx	sconh
			subc				ldint		sinty	sintx	sn
			subr				ldpc		spcy	spcx	fillb0
			subrc								fillbm
											fillh0
			not								fillhm
			-one								
			zero								ldmbrqy

srlb assembles srl, sbz, fillb0

srab assembles sra, sbz, fillbm

srlh assembles srl, sbz, fillh0

srah assembles sra, sbz, fillhm

Important Note: each fill-- asserts skmar.

marmbr assembles sll, ldmar, ldmbrqy

Important Note: ldmbrqy asserts sn and skmar.

TABLE 3.13 uData Path Fields and new Control lines

ul analysis gives rise to user data path control fields				
Figure 3.4 on page 102	rx	ry	rz	
	selx	sely	selz	ldz
	aluop	sh		
	rw	mcc	misc	
Figure 3.29 on page 127	adr	jmp_to		
Note: k has three sources. This is why k cannot be a mI field.				
mI execution in one computer cycle adds control lines:				
Figure 3.13 on page 107	skir, skmar			
Figure 3.15 on page 108	scz			

TABLE 3.13 uData Path Fields and new Control lines

Figure 3.18 on page 110	scony, sconx, sc1, sc0 Note: sc1, sc0 encoded from scon+, scon , sconh, sn.
Figure 3.21 on page 114	smemy, smemx, L2, L1, L0 Note: Lj encoded from fillb0, fillbm, fillh0, fillhm.
Figure 3.24 on page 118	ldmbrqy, w3 w2 w1 w0 (to memory) Note: wj encoded from mar[1..0], SW, SH, SB
Figure 3.26 on page 121	addn, addm, rypc, r0pc

3.1.4 Microinstructions

The process that created the RISC uData Path was dominated by the RISC goal: one ml per RISC ul. As the process evolved the four CISC ml were modified to suit that goal. The **mBR** ml was eliminated by adding the adr and jmp_to fields to the RISC ml_word. A third source operand, s3, was added to the **mALU** ml. And, the mop operand was added to the **mMOV** ml. The following set of three ml and one ml_word resulted. The ml_word details are found in Table 3.12.

```

mALU  op  sx sy s3 dz rw
mMOV  mop s  dz rw
mNOP

```

FIGURE 3.28

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
--------	-----	--	-------	----	-----	----	-----	------	------	------	------

The number of bits in the CISC ml word was reduced to 32 because we chose to have different ml_words for each type of ml (Table 2.23). This time we choose to use the same ml_word for all of the RISC ml. The price paid is an increased number of RISC ml_word bits. The gain is in a simplified mDecoder.

3.1.5 Micro Data Path

The micro data path is the source of the control lines that set up the user data path to execute ul. The micro data path we choose for the RISC computer is shown in Figure 3.29 on page 127. This is a modified copy of the CISC micro data path (Figure 2.19 on page 39). The modifications primarily stem from the use of the absolute address jmp_to field instead of the relative address moff field.

In Figure 3.29 on page 127 note how adr selects the next address source from jmp_to or ml_start

bits. Observe how the jmp_to source replaces the CISC mpc+1 and mpc+moff sources shown in Figure 2.19 on page 39. Also observe that ir bits from the uData Path are mControl inputs. Like CISC the RISC opcode field is the source for the ml_start address. The RISC ul fields for rx, ry, and rz are routed from the ir outputs to the register file inputs. The RISC ul m, which includes n, field is routed to the program control logic.

3.1.6 Micro Control

One ml per ul implies in a not very obvious way that hard wired logic can be used in lieu of microcode. This other form of microcontroller does not use microprograms. The flexible, changeable microcode is replaced by hardwired logic circuits. This important and complex form is considered in a separate chapter. In this chapter we use the mPC mROM mIR microcontroller form (Figure 3.29 on page 127). As before, each ml field activates a subset of user data path control lines. The lines activated by the ml_word bits set up the user data path. The ml executes its functions when the mCtrl ASM steps through its states (Figure 3.30 on page 128). The ASM executes one ml at a time.

FIGURE 3.29 RISC mControl - mData Path and mCTRL

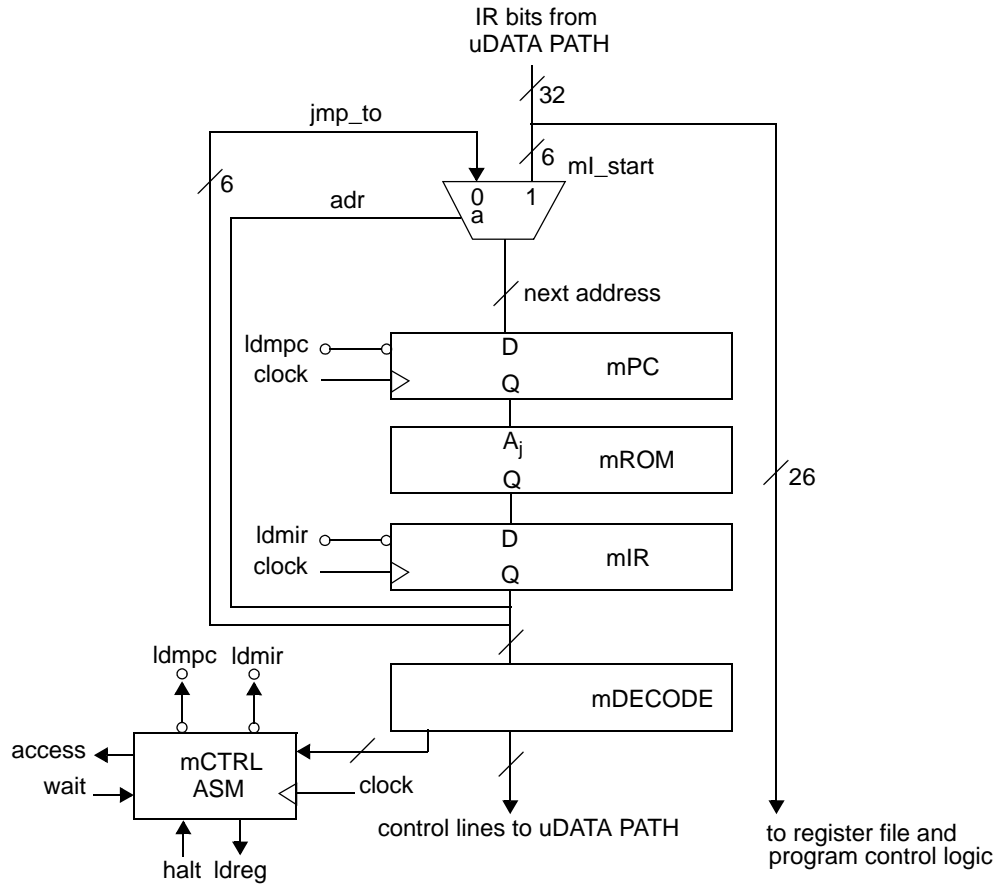
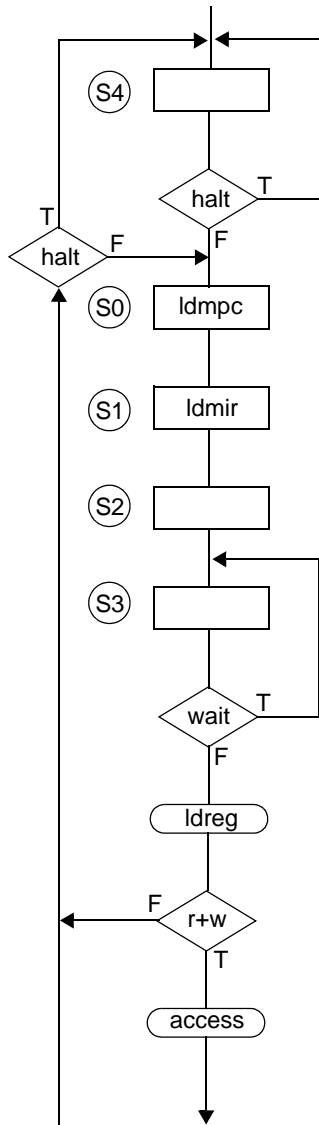


FIGURE 3.30 RISC mControl - Elementary ASM Chart for mCTRL



3.1.7 Microprogramming

RISC computer microassembly language programming is easier than CISC computer microassembly language programming because ml lists for RISC ul reduce to one or two ml. A RISC microassembler translates ul into ml. On the other hand the RISC microprogramming process is more

difficult because the hardware is more complex.

A microassembler translates ml into ml_word. In what follows both translations are shown: ul to ml, ml to ml_word.

Fetching the next ul requires two ml (the fetch_ul ml list). This unavoidable overhead in effect represents a serious degradation of performance. With most RISC ul implemented by one ml the overhead is about two-thirds of the user's program elapsed time for execution. However, pipelining reduces the elapsed time occupied by overhead ml. [Pipelining is discussed in another chapter.]

3.1.7.1 ml Fields

The microinstruction word uses jmp_to and adr fields to set up and control the RISC mControl mData Path in Figure 3.29 on page 127 for the next ml address. These mControl fields are not found in Table 3.11 because they are not part of the uData Path. These mControl fields are additions to the ml_word. The need for an **mBR** ml is eliminated when the jmp_to and adr fields are part of all ml words.

Each ml field activates a subset of user data path control lines. The lines activated by the ml_word bits set up the user data path. The ml executes its functions when the mCtrl ASM steps through its states (Figure 3.30 on page 128). The ASM executes one ml at a time.

Microprogramming is the business of filling in the bit patterns of 1's and 0's in the fields of ml-words stored in the mROM (Figure 3.29 on page 127). When a field's bit pattern is not encoded each bit corresponds to one user data path control line. Clearly no encoding results in the largest number of bits in the ml word. Also this choice results in a minimum number of ml in the mROM. This code is wide and short. This code is called horizontal microcode.

Encoding narrows the ml word by reducing the number of bits. The price paid is the need for mDecoder circuits with their propagation delays. Cost reduction is one motivation for encoding because mROM bits generate more system costs than decoders. The bits of a field encode or do not encode the set of control lines they control. Field encoding reduces the number of field bits.

Furthermore, we can reduce the ml word bit count by encoding the fields themselves. This tall and narrow ml coding is known as vertical microcode. We choose the middle road encoding only the fields of bits, and not encoding the fields into master-fields.

TABLE 3.14 Micro Assembly Language for Micro Instructions

mALU	op	sx	sy	s3dz	rw
	nop	nop	nop	nop	nop
	add	k		mcc	r
	addc	mem	mem	addn	mar
	+1	r0	r0		mbr
	-1	rx	ry		rz
	neg	con	con		
	sub	int	int		int
	subc	pc	pc		pc
	subr				

TABLE 3.14 Micro Assembly Language for Micro Instructions

	subrc				ir
	not				
	-one (all ones)				
	zero (all zeros)				
	and				
	or				
	xor				
	subz (special operator for BZ . See 3.1.7.9 Program Control)				
	rotr (shifts)				
	sll				
	sra				
	srl				
mMOV	mop	s	dz	rw	
	nop	nop	nop	nop	
	incpc			r	
	const+	mem	mar	w	
	const-	r0	mbr		
	consth	rx	rz		
	rypc	ry			
	r0pc	int	int		
	addm	pc	pc		
			ir		

TABLE 3.15 mI_word Fields Specified by mI Operands

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
			nop	nop		nop	nop	nop	nop	nop	nop
			add	rotr		r					incpc
			addc	sll		w	mar		mem	mem	const+
			+1	sra			mbr		r0	r0	const-
			-1	srl			rz		ry	rx	consth
			neg						con	con	rypc

TABLE 3.15 mI_word Fields Specified by mI Operands

		sub				int		int	int	r0pc
		subc				pc		pc	pc	addm
		subr								addn
		subrc				ir			mcc_report	
		not								
		-one (all ones)								
		zero (al zeros)								
		and								
		or								
		xor								
		subz								

TABLE 3.16 Control Lines for mI_Word Fields

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
(1)	(2)		(3)	(4)	(5)	nop	nop	nop	nop	nop	nop
						r	ldir	saz			incpc
						w	ldmar	sxz	smemx	smemy	skir
							ldmbr	syx	sr0y	sr0x	scon+
							ldrz	sbz	sryy	srxx	scon-
								scz	scony	sconx	sconh
							ldint		sinty	sintx	sn
							ldpc		spcy	spcx	
											fillb0
											fillbm
											fillh0
											fillhm
											ldmbrqy
											addn
											addm

TABLE 3.16 Control Lines for mI_Word Fields

											rypc
											r0pc

(1)→ Enter absolute address to jump to.

(2) → adr_selected address source

0 jmp_to
1 ml_start

(3) → See Table 3.17

(4) → See Table 3.2

subz assembles sub and the z mcc.

srlb assembles srl, sbz, fillb0

srab assembles sra, sbz, fillbm

srh assembles srl, sbz, fillh0

srah assembles sra, sbz, fillhm

Important Note: each fill-- asserts skmar.

marmbr assembles sll, ldmar, ldmbry

Important Note: ldmbry asserts sn and skmar.

TABLE 3.17 mALU operation codes

code	aluop	definition
01001101	add	y plus x (y = 181 a, x = 181 b)
01001111	addc	y plus x plus c (carry)
00000011	+1	y plus 1
01111101	-1	y minus 1
00000010	neg	y = y'+ 1 (2's complement)
00110011	sub	y minus x = y + x'+ 1
00110111	subc	y minus x with borrow = y + x'+ c
01001010	subr	x minus y = x + y'+ 1
01001110	subrc	x minus y with borrow = x + y'+ c
10000101	not	y' (1's complement of y input)
11100101	one	all 1's out, ignore alu inputs
10011101	zero	all 0's out, ignore alu inputs
11011101	and	y and x
11110101	or	y or x

TABLE 3.17 mALU operation codes

10110101	xor	y xor x
When shifting x: aluop = 11010001.		
When shifting y: aluop = 11111001.		
000	nop	when performing non shifting aluop
001	rotl	shift left 1 bit, msb to lsb, msb to c
010	sll	shift left 1 bit, 0 to lsb, msb to c
011	sra	shift right 1 bit, msb to msb, lsb to c
111	srl	shift right 1 bit, 0 to msb, lsb to c

3.1.7.2 ml used as Fetch_ul Control cl

Control instructions fetch the next ul in the program. The fetch ul memory access is implemented with two RISC ml at the fetch_ul mROM (arbitrarily chosen) addresses 00000002 and 00000012. The mControl multiplexer (Figure 3.29 on page 127) selects the jmp_to field as the next ml address when the adr field is 0 (Table 3.12). When adr is 1 the ml_start address from the ul code in the ir is selected as the next ml address and the jmp_to field becomes a don't care. We will put the ml_start label in the jmp_to field when adr is 1 to emphasize a jump to a new ul ml list. This is done because we do not know the ml_start address. We do not know the ml_start address because we cannot predict which ul is next in a user program. Furthermore, we will put the fetch_ul label (instead of address 00000002) in the jmp_to field when adr is 0 to emphasize a jump to the fetch_ul ml list. The first ml (Figure 3.31 on page 134) in the fetch_ul list has three active operands: pc, mar, and r.

```
mMOV nop pc mar r
```

Operand pc assembles spcy, and syz.
 Operand mar assembles ldmar.
 Operand r assembles r.

When the ml is executed by the ASM the pc moves to the mar and r activates a read access. The jmp_to address is 00000012. The next ml executed is the second ml in the fetch_ul list. The second ml has three active operands: incpc, mem, and ir.

```
mMOV incpc mem ir nop
```

Operand mem assembles nothing.
 Operand ir assembles ldირ.
 Operand incpc assembles incpc.

The special bus from the memory q-output to the ir d-input eliminates the need for mem to assemble smemy and syz. When the second ml is executed by the ASM the memory data representing the ul most recently fetched moves to the ir. The next ml executed is the first ml in the ml list for the most recently fetched ul.

FIGURE 3.31 Fetch_ul Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
00000010			nop	nop	nop	r	ldmar	spcy	syz	nop	nop
ml_start	1		nop	nop	nop	nop	ldir	nop	nop	nop	incpc

```

addr 0      (mMOV mop s dz rw)
          mMOV nop pc mar r
addr 1      mMOV incpc mem ir nop

```

3.1.7.3 Arithmetic and Logical Operators

Microcoding arithmetic and logical operators is straightforward except, perhaps, for the immediate word operand *n* (Figure 3.32 on page 135).

The first *ml* (Figure 3.32 on page 135) has four active operands: *add*, *rx*, *ry*, and *rz*.

```
ADD rx ry rz → mALU add rx ry nop rz nop
```

```

Operand rx assembles srxx.
Operand ry assembles sryy.
Operand rz assembles ldrz.
Operand add assembles add and saz.

```

When the **mALU** *ml* is executed by the ASM the sum of the contents of *rx* and *ry* is loaded into *rz*. The second *ml* has four active operands: *add*, *con*, *ry*, and *rz*.

```
ADD n ry rz → mALU add con ry nop rz nop
```

```

Operand con assembles sn, sconx, and sxz
Operand ry assembles sryy.
Operand rz assembles ldrz.
Operand add assembles add and saz. (saz replaces sxz)

```

The *sn* (Figure 3.18 on page 110) forms a signed number whose lower 16 bits are represented by *n* in the *ul* code. The signed number has copies of *n*'s msb in the upper 16 bits (Figure 3.18 on page 110). Therefore when the **mALU** *ml* is executed by the ASM the sum of *n* and the contents of *ry* is loaded into *rz*.

The *jmp_to* address is *fetch_ul* (= 00000002) in both *ml*. The next *ml* executed is the first *ml* in the *fetch_ul* list.

FIGURE 3.32 Arithmetic and Logical Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
fetch_ul 0			add	nop	nop	nop	ldrz	saz	sryy	srxx	nop
fetch_ul 0			add	nop	nop	nop	ldrz	saz	sryy	sconx	sn

```

uI          → mALU op  sx  sy s3  dz rw
ADD rx ry rz → mALU add rx  ry nop rz nop
ADD n  ry rz → mALU add con ry nop rz nop

```

3.1.7.4 Shift Operators

Microcoding shifts (Figure 3.33 on page 136) is not obvious because the rx or k operands specify the five bit number k. The five lsb of n in register rx represent k (Figure 3.13 on page 107). The number k programs the barrel shifter to shift k bits. The skir control line selects k from the ir's five lsb. When skir is not active the k source is rx's five lsb. This is why a field entry is not required when rx is the operand. SLL is the example shown in Figure 3.33 on page 136.

The first ml (Figure 3.33 on page 136) has four active operands: sll, rx, ry, and rz.

```
SLL rx ry rz → mALU sll rx ry nop rz nop
```

```

Operand rx  assembles srxx.
(srxx puts rx on the x bus which not used.)
(rx[4..0] provides the k number.)
Operand ry  assembles sryy.
Operand rz  assembles ldrz.
Operand sll assembles sll and sbz.

```

When the **mALU** ml is executed by the ASM the contents of ry shifted k bits by the k number in rx is loaded into rz.

The second ml has four active operands: sll, k, ry, and rz.

```
SLL k  ry rz → mALU sll k  ry nop rz nop
```

```

Operand k   assembles skir.
Operand ry  assembles sryy.
Operand rz  assembles ldrz.
Operand sll assembles sll and sbz.

```

When the **mALU** ml is executed by the ASM the contents of ry shifted k bits by the k number in the ir is loaded into rz.

The jmp_to address is fetch_ul (= 00000002) in both forms of **SLL**. The next ml executed is the first ml in the fetch_ul list.

FIGURE 3.33Shift Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
fetch_ul 0			nop	sll	nop	nop	ldrz	sbz	sryy	srxx	nop
fetch_ul 0			nop	sll	nop	nop	ldrz	sbz	sryy	nop	skir

```

uI          → mALU op  sx sy s3  dz rw
SLL rx ry rz → mALU sll rx ry nop rz nop
SLL k  ry rz → mALU sll k  ry nop rz nop

```

3.1.7.5 CONST

Microcoding **CONST+**, **CONST-**, or **CONSTH** is a matter of selecting the correct misc field code (Figure 3.18 on page 110 and Figure 3.34 on page 136). The ml for **CONST-** and **CONSTH** assemble microcode just like **CONST+** does. The only difference is found in the misc field entry. The first ml (Figure 3.34 on page 136) has three active operands: scon+, con, and rz.

```
CONST+ n rz → mMOV  const+ con rz nop      Constant = 0^16##n
```

Operand con assembles sn, sconx, and sxz

Operand rz assembles ldz.

Operand const+ assembles scon+. (scon+ replaces sn)

Scon+ replaces the previously assembled sn. Scon+ forms the constant.

When the **mMOV** ml is executed by the ASM the positive constant formed from n is loaded into rz.

The jmp_to address is fetch_ul (= 00000002) in all **CONST** ul. The next ml executed is the first ml in the fetch_ul list.

FIGURE 3.34CONST Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
fetch_ul 0			nop	nop	nop	nop	ldrz	szx	nop	sconx	scon+
fetch_ul 0			nop	nop	nop	nop	ldrz	szx	nop	sconx	scon-
fetch_ul 0			nop	nop	nop	nop	ldrz	szx	nop	sconx	sconh

```

uI          → mMOV  mop  s  dz rw
CONST+ n rz → mMOV  const+ con rz nop      Constant = 0^16##n
CONST  n rz → mMOV  const  con rz nop      Constant = 1^16##n
CONSTH n rz → mMOV  consth con rz nop      Constant = n##0^16

```

3.1.7.6 Scc

Microcoding Scc is complicated by two processes proceeding in parallel. One process performs subtraction and sets condition codes implementing a comparison function. The parallel process

places the `mcc_report` on the z bus and stores it in rz.

The first ml (Figure 3.35 on page 137) has five active operands: `sub`, `rx`, `ry`, `mcc_report`, and `rz`. `SZ` is `Sc` where `cc = Z`.

```
SZ rx ry rz → mALU sub rx ry mcc_report rz nop
```

Operand `sub` assembles `sub` and `saz`.

Operand `rx` assembles `srx`.

Operand `ry` assembles `sry`.

Operand `mcc_report` assembles the `cc z`, and `scz`.

Operand `rz` assembles `ldr`. (`scz` replaces `saz`)

When the ml is executed by the ASM the difference `ry - rx` sets status bits. The `mcc_report` is placed on the z bus and then it is loaded into `rz`.

The second ml has five active operands: `sub`, `con`, `ry`, `mcc_report`, and `rz`.

```
SZ n ry rz → mALU sub con ry mcc_report rz nop
```

Operand `sub` assembles `sub` and `saz`.

Operand `con` assembles `sn`, `scon`, and `sxz`. (`sxz` replaces `saz`)

Operand `ry` assembles `sry`.

Operand `mcc_report` assembles the `cc z`, and `scz`.

Operand `rz` assembles `ldr`. (`scz` replaces `sxz`)

When the ml is executed by the ASM the difference `ry - n` sets status bits. The `mcc_report` is placed on the z bus and then it is loaded into `rz`.

The `jmp_to` address is `fetch_ul` (= 00000002). The next ml executed is the first ml in the `fetch_ul` list.

FIGURE 3.35 `Sc` Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
fetch_ul	0	0	sub	nop	z	nop	ldr	scz	sry	srx	nop
fetch_ul	0	0	sub	nop	z	nop	ldr	scz	sry	scon	sn

```
uI          → mALU op sx sy s3 dz rw
SZ rx ry rz → mALU sub rx ry mcc_report rz nop
SZ n ry rz  → mALU sub con ry mcc_report rz nop
```

3.1.7.7 Load

Load is jargon for data transfers from memory to registers. The `ul LB`, `LH`, and `LW` transfer 1, 2, and 4 bytes respectively. In all cases the bytes transferred fill the destination register starting from the lsb bit 00. Word transfers are straightforward because register and memory words are 32 bits wide. However byte and halfword transfers need alignment to register bit 00 when the two address lsb are not 00. We say the two address lsb represent the `in_word` byte address (Figure 3.19 on page 111). Two bit `in_word` byte addresses 01, 10, 11 require 8, 16, and 24 bit shift right logical (`srl`) operations to align the byte(s) read from memory. And, register bits not receiving transferred bits are

set to 0. In this way the data transferred is filled as well as shifted.

Special shift ops srlh and srlb (Table 3.12) specify operations to shift and fill the unsigned data word with zeros (Figure 3.21 on page 114 and Figure 3.36 on page 138). Special shift ops srah and srab (Table 3.12) specify operations on the signed data to shift and fill the signed data word with msb's (Figure 3.21 on page 114 and Figure 3.36 on page 138). In turn the fill commands also assert skmar. The first ml (Figure 3.36 on page 138) has five active operands: add, con, ry, mar, and r. The first ml fetches the data from memory. [A review of the *ry(n) address mode definition may be in order.]

LB *ry(n) rz → **mALU** add con ry nop mar r

Operand con assembles sn, sconx, and sxz.

Operand ry assembles sryy.

Operand mar assembles ldmar.

Operand r assembles r.

Operand add assembles add and saz.(saz replaces sxz)

The first ml jmp_to entry is mPC+1. This points to the second ml in the list. The second ml (Figure 3.36 on page 138) has three active operands: srab, mem, and rz. The second ml stores the aligned data in rz.

Operand mem assembles smemx.

Operand rz assembles ldrz.

Operand srab assembles sra, sbz, and fillbm.

The second ml jmp_to entry is fetch_ul (= 00000002). When the two ml are executed by the ASM the data stored at address ry+n is loaded into rz. The next ml executed is the first ml in the fetch_ul list. The only assembled field differences are as follows.

TABLE 3.18

	sh	misc
LB *ry(n) rz	sra	fillbm
LBU *ry(n) rz	srl	fillb0
LH *ry(n) rz	sra	fillhm
LHU *ry(n) rz	srl	fillh0
LW *ry(n) rz		

The data read by the LW ul does not need alignment. This is why **mMOV** is used to transfer data from mem to rz (Figure 3.38 on page 139).

FIGURE 3.36 LB (Load Byte) Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
mPC+1	0		add	nop	nop	r	ldmar	saz	sryy	sconx	sn
0	0		nop	sra	nop	nop	ldrz	sbz	smemy	nop	fillbm

```

        mALU op  sx sy s3 dz rw
LB *ry(n) rz  → mALU add con ry nop mar r
               mALU srab nop mem nop rz nop

```

FIGURE 3.37LH (Load Halfword) Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
mPC+1	0		add	nop	nop	r	ldmar	saz	sryy	sconx	sn
0	0		nop	sra	nop	nop	ldrz	sbz	smemy	nop	fillhm

```

        mALU op  sx sy s3 dz rw
LH *ry(n) rz  → mALU add con ry nop mar r
               mALU srah nop mem nop rz nop

```

FIGURE 3.38LW (Load Word) Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
mPC+1	0		add	nop	nop	r	ldmar	saz	sryy	sconx	sn
0	0		nop	nop	nop	nop	ldrz	syz	smemy	nop	nop

```

        mALU op  sx sy s3 dz rw
        mMOV mop s  dz rw
LW *ry(n) rz  → mALU add con ry nop mar r
               mMOV nop mem rz nop

```

3.1.7.8 Store

Store is an alias for writing data from registers to memory. The ul **SB**, **SH**, and **SW** transfer 1, 2, and 4 bytes respectively. In all cases bytes transferred fill the destination memory bytes according to the in_word byte address (Figure 3.19 on page 111). Word transfers are straightforward because register and memory words are 32 bits wide. However byte and halfword transfers need alignment to the in_word byte address when the two address lsb are not 00. In_word byte addresses of binary 01, 10, 11 require 8, 16, and 24 bit shift left logical (sll) operations to align and mask the transferred byte(s).

If SB (Figure 3.39 on page 140) is executing and in_word address is 01 an 8 bit shift left logical is implemented. The shift operation drops off 8 high bits and shifts in 8 zeros. Now only the byte at the in_word address has meaning. The entire word is put on the data bus but the zeros shifted in and the high byte bits are ignored because they are not written to memory. This is so because only w1 is activated by the ul **SB** in combination with in_word address 01 (Figure 3.24 on page 118).

Emphasis: When storing a byte the left shift is 0, 8, 16, or 24 bits according to the byte position in the word.

The ml implementing **SB** (Figure 3.39 on page 140) has five active operands: add, rx, con, marmbr,

and w.

SB *ry *rx(n)* → **mALU** *add rx con nop marmbr w*

Operand con assembles sn, scony, and syz.

Operand rx assembles srxx.

Operand marmbr assembles sll, ldmar, ldmbrrqy.

Operand w assembles w.(ldmbrrqy replaces sn)

Operand add assembles add and saz.(saz replaces syz)

The jmp_to address is fetch_ul (= 00000002) in all store ul. The next ml executed is the first ml in the fetch_ul list.

When the ul is executed by the ASM the data in ry is stored at address rx+n.

The **SH** and **SW** ul are executed by the same microcode that executes **SB** (Figure 3.40 on page 140 and Figure 3.41 on page 140). The correct value of mar[1..0]LLL implements **SH** or **SW** instead of **SB** (Figure 3.24 on page 118).

FIGURE 3.39SB (Store Byte) Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		add	sll	nop	w	ldmar	saz	scony	srxx	ldmbrrqy

SB *ry *rx(n)* → **mALU** *op sx sy s3 dz rw*
mALU *add rx con nop marmbr w*

FIGURE 3.40SH (Store Halfword) Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		add	sll	nop	w	ldmar	saz	scony	srxx	ldmbrrqy

SH *ry *rx(n)* → **mALU** *op sx sy s3 dz rw*
mALU *add rx con nop marmbr w*

FIGURE 3.41 Store Word Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		add	sll	nop	w	ldmar	saz	scony	srxx	ldmbrrqy

1 mI code:
SW *ry *rx(n)* → **mALU** *op sx sy s3 dz rw*
mALU *add rx con nop marmbr w*

3.1.7.9 Program Control

Program control uses the special operators `addn`, `addm`, `rypc`, `r0pc` to activate the program control adder (Figure 3.26 on page 121). This adder forms the appropriate `branch_to` address in parallel with the data path `alu` executing the sub operation and setting the status bits.

Depending on which `ml` is executed the `pc` is loaded with the `branch_to` address `ry`, or zero unconditionally. Or the addresses `pc+n` or `pc+m` if the `mcc_report` is true (Figure 3.26 on page 121).

BZ *n ry*: The `ml` implementing BZ (Figure 3.42 on page 141) has four active operands: `sub`, `r0`, `ry`, and `addn`.

BZ *n ry* \rightarrow **mALU** `sub r0 ry addn nop nop`

Operand `r0` assembles `sr0x`.
 Operand `ry` assembles `sryy`.
 Operand `addn` assembles `addn`.
 Operand `subz` assembles `sub` and `z`.

The `jmp_to` address is `fetch_ul` (= 00000002). The next `ml` executed is the first `ml` in the `fetch_ul` list. If the `cc` is false the next `ul` fetched is the `ul` addressed by `pc+4`. If the `cc` is true the next `ul` fetched is the `ul` addressed by `pc+n`.

FIGURE 3.42 Branch Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		sub	nop	z	nop	nop	nop	sryy	sr0x	addn

BZ *n ry* \rightarrow **mALU** `op sx sy s3 dz rw`
 mALU `sub r0 ry addn nop nop`

JMP *ry*: The unconditional **JMP *ry*** (Figure 3.43 on page 142) puts a copy of `ry` in the `pc`.

The first `ml` implementing **JMP *ry*** (Figure 3.43 on page 142) has two active operands: `ry` and `pc`.

Operand `ry` assembles `sryy` and `syz`.
 Operand `pc` assembles `ldpc`.

The second `ml` implementing **JMP *ry*** (Figure 3.43 on page 142) has one active operand: `rypc`. See Figure 3.26 on page 121.

Operand `rypc` assembles `rypc`.

In both `ml` the `jmp_to` address is `fetch_ul` (= 00000002). The next `ml` executed is the first `ml` in the `fetch_ul` list.

JMP *m*: In the unconditional **JMP *m*** microcode (Figure 3.44 on page 142) special operator `addm` (Figure 3.26 on page 121) forms `pc+m` and loads it into the `pc`.

Operand addm assembles addm.

FIGURE 3.43Jump *ry* Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	ldpc	syz	sryy	nop	nop
or the alternate :											
0	0		nop	nop	un	nop	nop	nop	nop	nop	rypc

JMP *ry* → **mMOV** *mop* *s* *dz* *rw*
mMOV *nop* *ry* *pc* *nop*

FIGURE 3.44Jump *m* Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	nop	nop	nop	nop	addm

JMP *m* → **mMOV** *mop* *s* *dz* *rw*
mMOV *addm* *nop* *nop* *nop*

CALL *m* and **CALL** *ry*: In the unconditional **CALL** *m* microcode (Figure 3.45 on page 142) special operator addm (Figure 3.26 on page 121) forms pc+m and loads it into the pc in parallel with the move from the pc to r31.

In the unconditional **CALL** *ry* microcode (Figure 3.45 on page 142) special operator rypc (Figure 3.26 on page 121) loads ry into the pc in parallel with the move from the pc to r31.

FIGURE 3.45Call *m* Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	ldrz	syz	spcy	nop	addm

CALL *m* → **mMOV** *mop* *s* *dz* *rw*
mMOV *addm* *pc* *r31* *nop*

FIGURE 3.46Call *ry* Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	ldrz	syz	spcy	nop	rypc

CALL *ry* → **mMOV** *mop s dz rw*
mMOV *rypc pc r31 nop*

TRAP: For **TRAP** (Figure 3.47 on page 143) special operator *r0pc* loads zero into the *pc* (Figure 3.24 on page 118) in parallel with the move from the *pc* to *int*. The **RET** microcode is straightforward (Figure 3.48 on page 143).

FIGURE 3.47Trap Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	ldint	syx	spcy	nop	r0pc

TRAP → **mMOV** *mop s dz rw*
mMOV *r0pc pc int nop*

FIGURE 3.48Return Microcode

jmp_to	adr		aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
0	0		nop	nop	un	nop	ldpc	szx	nop	sintx	nop

RTE → **mMOV** *mop s dz rw*
mMOV *nop int pc nop*

3.1.8 CD RISC

FIGURE 3.49 Behavioral view

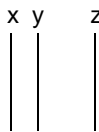
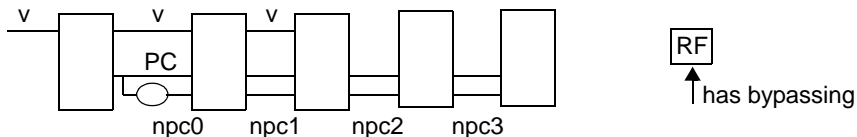


FIGURE 3.50 Structural view



Add ASM gen flow to ISA/CD

Exercises 3:

Use RISC ul (Table 3.1 p302) to write user programs implementing the CISC ul (Table 2.1 p211) in

5/22/08

Confidential Copyright © 2006 Nicholas L. Pappas Copying in any form without the expressed written permission of Nicholas L. Pappas is prohibited

143

the following exercises.

- 3.1 ABS ry
- 3.2 ADDC *rx ry
- 3.3 BTST rx * ry
- 3.4 CLR *ry+
- 3.5 CMP *rx *ry
- 3.6 DEC *ry(n)
- 3.7 DIVS rx ry
- 3.8 DIVU rx ry
- 3.9 MULS rx ry
- 3.10 MULU rx ry
- 3.11 NEG *ry(n)
- 3.12 Scc *rx *ry
- 3.13 SUB * rx *ry
- 3.14 AND *rx ry
- 3.14 OR *rx(n) *ry
- 3.15 XOR addr_abs rx
- 3.16 SLL addr_abs *ry
- 3.17 ADD *rx addr_abs
- 3.18 DEC addr_abs
- 3.19 EXCH ry addr_abs
- 3.20 MOV *rx ry
- 3.21 MOV *rx *ry
- 3.22 MOV rx ry

3.23 PUSH ry

3.24 POP ry

3.25 JMP addr_abs

3.26 JRcc n

3.27 CALL target

3.28 CALL ry

3.29 RET

3.30 GETPC ry

3.31 EXCH pc *ry(n)

3.32 Reference fig 3.4. Design a circuit with outputs sc1, sc0.

Asserted Low Inputs: scon+ scon sconh sn

Asserted High Outputs: sc1 sc0

3.33 Reference fig 3.5. Explain why byte and halfword data loaded from memory must be shifted right so that data bit 0 is placed in destination register bit 0.

3.34 Reference fig 3.6. Design a circuit with outputs L2, L1, L0.

Asserted Low Inputs: snop srlh srah srlb srab

Asserted High Outputs: L2 L1 L0

3.35 Reference fig 3.10. Design the mCTRL state machine and output circuits.

In problems 3.40 through 3.84 use mROM addresses 3 and up. Figure 3.11 places the cl at addresses 0 and 1. In each problem show a two line answer. Line 1 is microcode using names as in figure 3.11. Line 2 is microcode using code numbers from Table 3.6. Use copies of Form 1 which is found below.

3.40 Microcode RISC ul ADD rx ry rz

3.41 Microcode RISC ul ADD n ry rz

3.42 Microcode RISC ul ADDC rx ry rz

- 3.43 Microcode RISC ul ADDC n ry rz
- 3.44 Microcode RISC ul SUB rx ry rz
- 3.45 Microcode RISC ul SUB n ry rz
- 3.46 Microcode RISC ul SUBC rx ry rz
- 3.47 Microcode RISC ul SUBC n ry rz
- 3.48 Microcode RISC ul SUBR rx ry rz
- 3.49 Microcode RISC ul SUBR n ry rz
- 3.50 Microcode RISC ul AND rx ry rz
- 3.51 Microcode RISC ul AND n ry rz
- 3.52 Microcode RISC ul OR rx ry rz
- 3.53 Microcode RISC ul OR n ry rz
- 3.54 Microcode RISC ul XOR rx ry rz
- 3.55 Microcode RISC ul XOR n ry rz
- 3.56 Microcode RISC ul SLL rx ry rz
- 3.57 Microcode RISC ul SLL n ry rz
- 3.58 Microcode RISC ul SRA rx ry rz
- 3.59 Microcode RISC ul SRA n ry rz
- 3.60 Microcode RISC ul SRL rx ry rz
- 3.61 Microcode RISC ul SRL n ry rz

- 3.62 Microcode RISC ul CONST+ n rz
- 3.63 Microcode RISC ul CONST n rz
- 3.64 Microcode RISC ul CONSTH n rz
- 3.65 Microcode RISC ul Scc rx ry rz

- 3.66 Microcode RISC ul Scc n ry rz
- 3.67 Microcode RISC ul LBU *ry(n) rz
- 3.68 Microcode RISC ul LB *ry(n) rz
- 3.69 Microcode RISC ul LHU *ry(n) rz
- 3.70 Microcode RISC ul LH *ry(n) rz
- 3.71 Microcode RISC ul LW *ry(n) rz
- 3.72 Microcode RISC ul SB rx *ry(n)
- 3.73 Microcode RISC ul SH rx *ry(n)
- 3.74 Microcode RISC ul SW rx *ry(n)
- 3.75 Microcode RISC ul BZ n ry
- 3.76 Microcode RISC ul BNZ n ry
- 3.77 Microcode RISC ul JMP m
- 3.78 Microcode RISC ul JMP ry
- 3.79 Microcode RISC ul CALL m
- 3.80 Microcode RISC ul CALL ry
- 3.81 Microcode RISC ul TRAP
- 3.82 Microcode RISC ul RTE
- 3.83 Microcode RISC ul HALT
- 3.84 Microcode RISC ul NOP

Form 1 for microcoding exercises 3.40 to 3.84.

TABLE 3.19

	jmp_to	adr	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
names											
codes											

TABLE 3.20

	jmp_to	adr	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
names											
codes											

TABLE 3.21

	jmp_to	adr	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
names											
codes											

TABLE 3.22

	jmp_to	adr	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
names											
codes											

TABLE 3.23

	jmp_to	adr	aluop	sh	mcc	rw	ldz	selz	sely	selx	misc
names											
codes											