ISTD 50.002 Computation Structures

Zhang Yue
Oka Kurniawan
Lu Wei

Original creator: Lozano-Perez, Tomas, using materials originally developed by Christopher J. Terman and Stephen A Ward. Course materials for ISTD 103, Computation Structures. MIT-SUTD Collaboration, 2012.

Modified by: Oka Kurniawan, 2013.

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

ISTD 50.002 Computation Structures

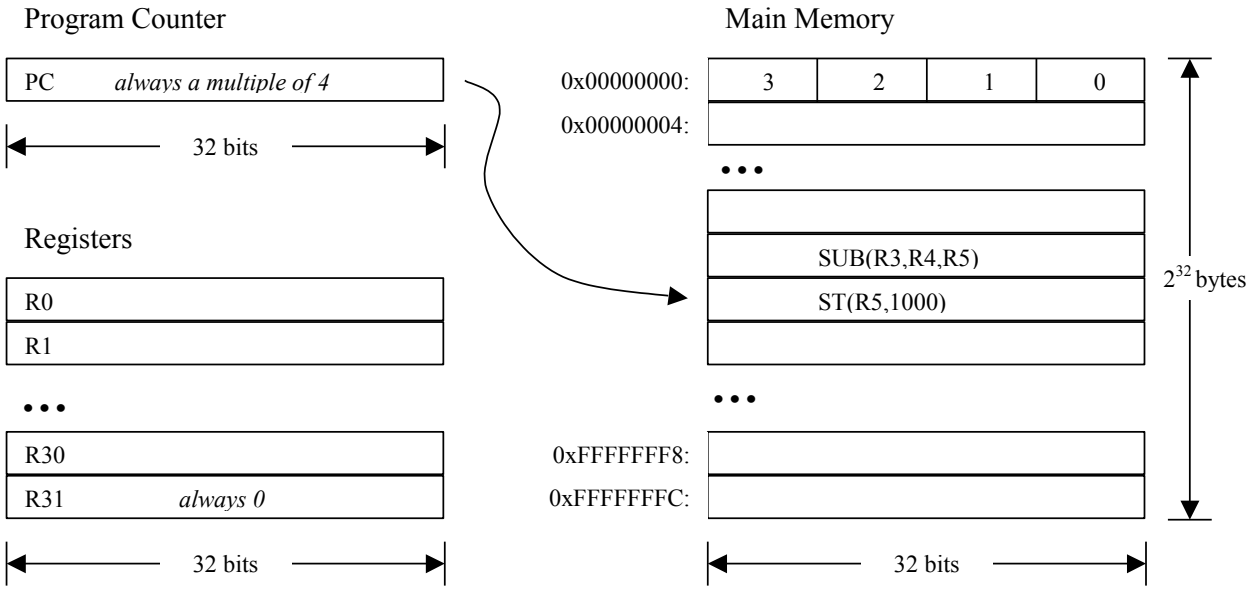
β Documentation

1. Introduction

This handout is a reference guide for the β, the RISC processor design for 50.002. This is intended to be a complete and thorough specification of the programmer-visible state and instruction set.

2. Machine Model

The β is a general-purpose 32-bit architecture: all registers are 32 bits wide and when loaded with an address can point to any location in the byte-addressed memory. When read, register 31 is always 0; when written, the new value is discarded.



3. Instruction Encoding

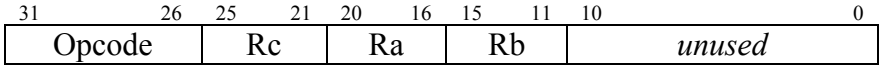
Each β instruction is 32 bits long. All integer manipulation is between registers, with up to two source operands (one may be a sign-extended 16-bit literal), and one destination register. Memory is referenced through load and store instructions that perform no other computation. Conditional branch instructions are separated from comparison instructions: branch instructions test the value of a register that can be the result of a previous compare instruction.

There are only two types of instruction encoding: *Without Literal* and *With Literal*. Instructions without literals include arithmetic and logical operations between two registers

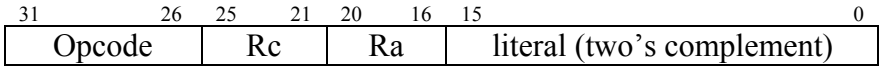
whose result is placed in a third register. Instructions with literals include all other operations.

Like all signed quantities on the β, an instruction's literal is represented in two's complement.

3.1 Without Literal



3.2 With Literal



4. Instruction Summary

Below are listed the 32 β instructions and their 6-bit opcodes. For detailed instruction operations, see the following section.

Mnemonic	*Opcode*
ADD	0x20
ADDC	0x30
AND	0x28
ANDC	0x38
BEQ	0x1D
BNE	0x1E
CMPEQ	0x24
CMPEQC	0x34

Mnemonic	*Opcode*
CMPLE	0x26
CMPLEC	0x36
CMPLT	0x25
CMPLTC	0x35
DIV	0x23
DIVC	0x33
JMP	0x1B
LD	0x18

Mnemonic	*Opcode*
LDR	0x1F
MUL	0x22
MULC	0x32
OR	0x29
ORC	0x39
SHL	0x2C
SHLC	0x3C
SHR	0x2D

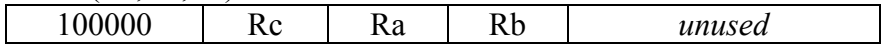
Mnemonic	*Opcode*
SHRC	0x3D
SRA	0x2E
SRAC	0x3E
SUB	0x21
SUBC	0x31
ST	0x19
XOR	0x2A
XORC	0x3A

5. Instruction Specifications

This section contains the specifications for the β instructions, listed alphabetically by mnemonic. No timing-dependent information is given: it is specifically assumed that there are no pathological timing interactions between instructions in this specification. Each instruction is considered atomic and is presumed to complete before the next instruction is executed. No assumptions are made about branch prediction, instruction prefetch, or memory caching.

5.1 ADD

Usage: ADD(Ra,Rb,Rc)



Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] + Reg[Rb]

The contents of register Ra are added to the contents of register Rb and the 32-bit sum is written to Rc. This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.2 ADDC

Usage: ADDC(Ra,literal,Rc)

Opcode:	110000	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] + SEXT(literal)

The contents of register Ra are added to *literal* and the 32-bit sum is written to Rc. This instruction computes no carry or overflow information. If desired, this can be computed through explicit compare instructions.

5.3 AND

Usage: AND(Ra,Rb,Rc)

Opcode:	101000	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] & Reg[Rb]

This performs the bitwise boolean AND function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.4 ANDC

Usage: ANDC(Ra,*literal*,Rc)

Opcode:	111000	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] & SEXT(literal)

This performs the bitwise boolean AND function between the contents of register Ra and *literal*. The result is written to register Rc.

5.5 BEQ/BF

Usage: BEQ(Ra,label,Rc)
BF(Ra,label,Rc)

Opcode:	011101	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) − OFFSET(current instruction)) / 4)−1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP = 0 then PC ← EA

The PC of the instruction following the BEQ instruction (the updated PC) is written to register Rc. If the contents of register Ra are zero, the PC is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

5.6 BNE/BT

Usage: BNE(Ra,label,Rc)
BT(Ra,label,Rc)

Opcode:	011110	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) − OFFSET(current instruction)) ÷ 4)−1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP ≠ 0 then PC ← EA

The PC of the instruction following the BNE instruction (the updated PC) is written to register Rc. If the contents of register Ra are non-zero, the PC is loaded with the target address; otherwise, execution continues with the next sequential instruction.

The displacement *literal* is treated as a signed word offset. This means it is multiplied by 4 to convert it to a byte offset, sign extended to 32 bits, and added to the updated PC to form the target address.

5.7 CMPEQ

Usage: CMPEQ(Ra,Rb,Rc)

Opcode:	100100	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] = Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are equal to the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.8 CMPEQC

Usage: CMPEQC(Ra,literal,Rc)

Opcode:	110100	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] = SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are equal to *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.9 CMPLE

Usage: CMPLE(Ra,Rb,Rc)

Opcode:	100110	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] ≤ Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or equal to the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.10 CMPLEC

Usage: CMPLEC(Ra,literal,Rc)

Opcode:	110110	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] ≤ SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than or eqaul to *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.11 CMPLT

Usage: CMPLT(Ra,Rb,Rc)

100101	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] < Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than the contents of register Rb, the value one is written to register Rc; otherwise zero is written to Rc.

5.12 CMPLTC

Usage: CMPLTC(Ra,literal,Rc)

110101	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] < SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

If the contents of register Ra are less than *literal*, the value one is written to register Rc; otherwise zero is written to Rc.

5.13 DIV

Usage: DIV(Ra,Rb,Rc)

100011	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] / Reg[Rb]

The contents of register Ra are divided by the contents of register Rb and the low-order 32 bits of the quotient are written to Rc.

5.14 DIVC

Usage: DIVC(Ra,literal,Rc)

110011	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] / SEXT(literal)

The contents of register Ra are divided by *literal* and the low-order 32 bits of the quotient is written to Rc.

5.15 JMP

Usage: JMP(Ra,Rc)

Opcode:	011011	Rc	Ra	0000000000000000

Operation:
PC ← PC+4
EA ← Reg[Ra] & 0xFFFFFFFC
Reg[Rc] ← PC
PC ← EA

The PC of the instruction following the JMP instruction (the updated PC) is written to register Rc, then the PC is loaded with the contents of register Ra. The low two bits of Ra are masked to ensure that the target address is aligned on a 4-byte boundary. Ra and Rc may specify the same register; the target calculation using the old value is done before the assignment of the new value. The unused literal field should be filled with zeroes. Note that JMP can clear the supervisor bit (bit 31 of the PC) but not set it – see section 6.3 for details.

5.16 LD

Usage: LD(Ra,literal,Rc)

Opcode:	011000	Rc	Ra	literal

Operation:
PC ← PC+4
EA ← Reg[Ra] + SEXT(literal)
Reg[Rc] ← Mem[EA]

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The location in memory specified by EA is read into register Rc.

5.17 LDR

Usage: LDR(label,Rc)

Opcode:	011111	Rc	11111	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
Reg[Rc] ← Mem[EA]

The effective address EA is computed by multiplying the sign-extended *literal* by 4 (to convert it to a byte offset) and adding it to the updated PC. The location in memory specified by EA is read into register Rc. The Ra field is ignored and should be 11111 (R31). The supervisor bit (bit 31 of the PC) is ignored (i.e., treated as zero) when computing EA.

5.18 MUL

Usage: MUL(Ra,Rb,Rc)

Opcode:	100010	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] * Reg[Rb]

The contents of register Ra are multiplied by the contents of register Rb and the low-order 32 bits of the product are written to Rc.

5.19 MULC

Usage: MULC(Ra,literal,Rc)

Opcode:	110010	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] * SEXT(literal)

The contents of register Ra are multiplied by *literal* and the low-order 32 bits of the product are written to Rc.

5.20 OR

Usage: OR(Ra,Rb,Rc)

Opcode:	101001	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] | Reg[Rb]

This performs the bitwise boolean OR function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.21 ORC

Usage: ORC(Ra,*literal*,Rc)

Opcode:	111001	Rc	Ra	literal

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] | SEXT(literal)

This performs the bitwise boolean OR function between the contents of register Ra and *literal*. The result is written to register Rc.

5.22 SHL

Usage: SHL(Ra,Rb,Rc)

101100	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] << $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted left 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.23 SHLC

Usage: SHLC(Ra,*literal*,Rc)

111100	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] << $\text{literal}_{4:0}$

The contents of register Ra are shifted left 0 to 31 bits as specified by the five-bit count in *literal*. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.24 SHR

Usage: SHR(Ra,Rb,Rc)

101101	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted right 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.25 SHRC

Usage: SHRC(Ra,*literal*,Rc)

111101	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{literal}_{4:0}$

The contents of register Ra are shifted right 0 to 31 bits as specified by the five-bit count in *literal*. The result result is written to register Rc. Zeroes are propagated into the vacated bit positions.

5.26 SRA

Usage: SRA(Ra,Rb,Rc)

Opcode:	101110	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in register Rb. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.25 SRAC

Usage: SRAC(Ra,*literal*,Rc)

Opcode:	111110	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $\text{literal}_{4:0}$

The contents of register Ra are shifted arithmetically right 0 to 31 bits as specified by the five-bit count in *literal*. The result is written to register Rc. The sign bit (Reg[Ra]_{31}) is propagated into the vacated bit positions.

5.28 ST

Usage: ST(Rc,literal,Ra)

Opcode:	011001	Rc	Ra	literal

Operation: PC ← PC+4

EA ← Reg[Ra] + SEXT(literal)

Mem[EA] ← Reg[Rc]

The effective address EA is computed by adding the contents of register Ra to the sign-extended 16-bit displacement *literal*. The contents of register Rc are then written to the location in memory specified by EA.

5.29 SUB

Usage: SUB(Ra,Rb,Rc)

Opcode:	100001	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] – Reg[Rb]

The contents of register Rb are subtracted from the contents of register Ra and the 32-bit difference is written to Rc. This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.30 SUBC

Usage: SUBC(Ra,literal,Rc)

Opcode:	110001	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] – SEXT(literal)

The constant *literal* is subtracted from the contents of register Ra and the 32-bit difference is written to Rc. This instruction computes no borrow or overflow information. If desired, this can be computed through explicit compare instructions.

5.31 XOR

Usage: XOR(Ra,Rb,Rc)

Opcode:	101010	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ Reg[Rb]

This performs the bitwise boolean XOR function between the contents of register Ra and the contents of register Rb. The result is written to register Rc.

5.32 XORC

Usage: XORC(Ra,*literal*,Rc)

Opcode:	111010	Rc	Ra	literal

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ SEXT(literal)

This performs the bitwise boolean XOR function between the contents of register Ra and *literal*. The result is written to register Rc.

6. Extensions for Exception Handling

The standard β architecture described above is modified as follows to support exceptions and privileged instructions.

6.1 Exceptions

β exceptions come in three flavors: traps, faults, and interrupts.

Traps and faults are both the direct outcome of an instruction (e.g., an attempt to execute an illegal opcode) and are distinguished by the programmer's intentions. Traps are intentional and are normally used to request service from the operating system. Faults are unintentional and often signify error conditions.

Interrupts are asynchronous with respect to the instruction stream, and are usually caused by external events (e.g., a character appearing on an input device).

6.2 The XP Register

Register 30 is dedicated as the ``Exception Pointer'' (XP) register. When an exception occurs, the updated PC is written to the XP. For traps and faults, this will be the PC of the instruction following the one which caused the fault; for interrupts, this will be the PC of the instruction following the one which was about to be executed when the interrupt occurred. The instruction pointed to by XP–4 has *not* been executed.

Since the XP can be overwritten at unpredictable times as the result of an interrupt, it should not be used by user-mode programs while interrupts are enabled.

6.3 Supervisor Mode

The high bit of the PC is dedicated as the “Supervisor” bit. The instruction fetch and LDR instruction ignore this bit, treating it as if it were zero. The JMP instruction is allowed to clear the Supervisor bit but not set it, and no other instructions may have any effect on it. Only exceptions cause the Supervisor bit to become set.

When the Supervisor bit is clear, the processor is said to be in “user mode”. Interrupts are enabled while in user mode.

When the Supervisor bit is set, the processor is said to be in “supervisor mode”. While in supervisor mode, interrupts are disabled and privileged instructions (see below) may be used. Traps and faults while in supervisor mode have implementation-defined (probably fatal) effects.

Since the JMP instruction can clear the Supervisor bit, it is possible to load the PC with a new value and enter user mode in a single atomic action. This provides a safe mechanism for returning from a trap to the Operating System, even if an interrupt is pending at the time.

6.4 Exception Handling

When an exception occurs and the processor is in user mode, the updated PC is written to the XP, the Supervisor bit is set, the PC is loaded with an implementation-defined value, and the processor begins executing instructions from that point. This value is called the “exception vector”, and may depend on the kind of exception which occurred.

The only exception which must be supported by all implementations is the “reset” exception (also called the “power up” exception), which occurs immediately before any instructions are executed by the processor. The exception vector for power up is always 0. Thus, at power up time, the Supervisor bit is set, the XP is undefined, and execution begins at location 0 of memory.

6.5 Privileged Instructions

Some instructions may be available while in supervisor mode which are not available in user mode (e.g., instructions which interface directly with I/O devices). These are called "privileged instructions". These instructions always have an opcode of 0x00; otherwise, their form and semantics are implementation-defined. Attempts to use privileged instructions while in user mode will result in an illegal instruction exception.

7. Software Conventions

This section describes our software programming conventions that supplement the basic architecture.

7.1 Reserved Registers

It is convenient to reserve a number of registers for pervasive standard uses. The hardware itself reserves R31 and R30; in addition, our software conventions reserve R29, R28, and R27.

These are summarized in the following table and are described more fully below.

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

7.2 Convenience Macros

We augment the basic β instruction set with the following macros, making it easier to express certain common operations:

Macro	Definition
BEQ(Ra, label)	BEQ(Ra, label, R31)
BF(Ra, label)	BF(Ra, label, R31)
BNE(Ra, label)	BNE(Ra, label, R31)
BT(Ra, label)	BT(Ra, label, R31)
BR(label, Rc)	BEQ(R31, label, Rc)
BR(label)	BR(label, R31)
JMP(Ra)	JMP(Ra, R31)
LD(label, Rc)	LD(R31, label, Rc)
ST(Rc, label)	ST(Rc, label, R31)
MOVE(Ra, Rc)	ADD(Ra, R31, Rc)
CMOVE(c, Rc)	ADDC(R31, c, Rc)
PUSH(Ra)	ADDC(SP, 4, SP) ST(Ra, -4, SP)

POP(Rc)	LD(SP, -4, Rc) SUBC(SP, 4, SP)
ALLOCATE(k)	ADDC(SP, 4*k, SP)
DEALLOCATE(k)	SUBC(SP, 4*k, SP)

7.3 Stack Implementation

SP is a reserved register that points to the top of the stack. The stack is an arbitrary contiguous region of memory. The contents of SP are always a multiple of 4 and each stack slot is 4 bytes. SP points to the location just beyond the topmost element on the stack. The stack grows upward in memory (i.e., towards higher addresses). Four macros are defined for manipulating the stack:

PUSH(Ra) - Push the contents of register Ra onto the stack

POP(Rc) - Pop the top element of the stack into Rc

ALLOCATE(k) - Push k words of uninitialized data onto the stack

DEALLOCATE(k) - Pop k words off of the stack and throw them away

7.4 Procedure Linkage

A procedure's arguments are passed on the stack. Specifically, when a procedure is entered, the topmost element on the stack is the first argument to the procedure; the next element on the stack is the second argument to the procedure, and so on. A procedure's return address is passed in LP, which is a register reserved for this purpose. A procedure returns its value (if any) in R0 and must leave all other registers, including the reserved registers, unaltered.

Thus, a typical call to a procedure named F looks like:

(push arg_{n-1})
...
(push arg_1)
(push arg_0)
BR (F, LP)
DEALLOCATE (n)
(use R0, which is now F(arg_0, arg_1, ... , arg_{n-1}))

7.5 Stack Frames

The preceeding section describes the rules which procedures must follow to interoperate properly. This section describes our conventional means of writing a procedure which follows those rules.

A procedure invocation requires storage for its arguments, its local variables, and any registers it needs to save and restore. All of this storage is allocated in a contiguous region of the stack called a "stack frame". Procedures "activate" stack frames on entry and "deactivate" them on exit. BP is a reserved register which points to a fixed location within the currently active stack frame. Procedures use a standard prologue and epilogue to activate and deactivate the stack frame.

The standard prologue is:

```
PUSH (LP)
PUSH (BP)
MOVE (SP, BP)
ALLOCATE (k)          | allocate space for locals
(push registers which are used by procedure)
```

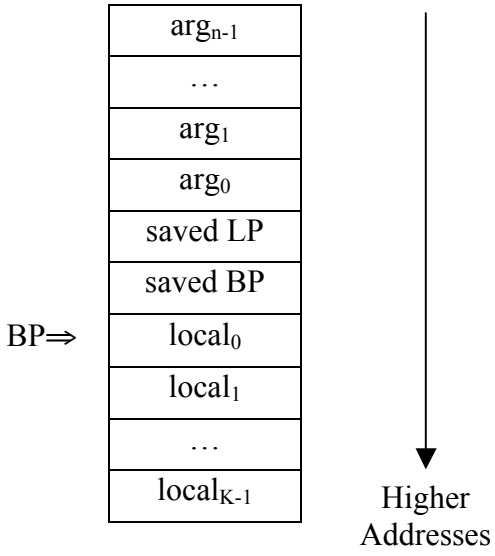
Note that either of the last two steps may be omitted if there are no local variables or if there are no registers to save.

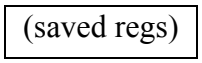
The standard epilogue is:

```
(pop registers which are used by procedure)
MOVE (BP, SP)         | deallocate space for locals
POP (BP)
POP (LP)
JMP (LP)
```

Note that the epilogue assumes that the body of the procedure has no net effect on SP. Also note that either or both of the first two steps may be omitted if there are no registers to restore or if there are no local variables.

The standard prologue and epilogue together with the argument passing conventions imply the following layout for a stack frame:



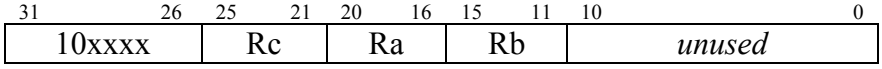


Note that BP always points to the first stack slot above the saved BP, which is the same as the first local variable (if any). It also points to the third stack slot above the first argument (if any). So within the procedure's body, its arguments and locals may be accessed via constant offsets from BP.

Summary of β Instruction Formats

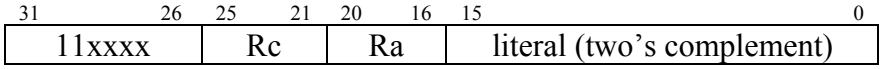
Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

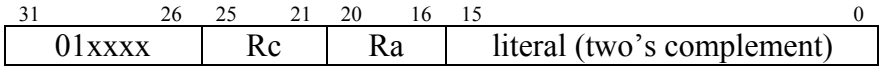
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/**BF**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/**BT**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(Ra,label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	