# Summary of Commands

Core Commands

(LOAD|STORE)[NEXT][BIG][RED] var (ACC|TEMP)

UNREAD var

MAP map

JUMP label

BRANCH label0 label1 label2 label3

LOADI (imm2 ACC | imm1 TEMP)

Other Primitive Commands

SLLs vard imm1

SRLs vard imm1

SLL2s vard imm2

SRL2s vard imm2

NOTs vard

COMPs vard

ZEROs vard imm1

ADDIs vard imm2

SUBIs vard imm2

LEQZ vard

Function Control

FUNC name param0 param1 …

PRIM name param0

END

Stack Access

$POP

$TOP

Higher Level Commands

ADD vard var0 var1

SUB vard var0 var1

BOL vard var0 var1 map

ADDs vard var0

SUBs vard var0

NSUBs vard var0

BOLs vard var0 map

SLL vard var0 imm1

SRL vard var0 imm1

SLT var0 var1

MULT vard var0 var1

MULTo vard var0 var1

DIV vard varr var0 var1

MOD vard var0 var1

RECP vard varr var0

PUSH var0

UNREADSP var0

# Core Commands

Load Unred Bit

LOAD var ACC LOADRED var ACC

LOAD var TEMP LOADRED var TEMP

LOADBIG var ACC LOADBIGRED var ACC

LOADBIG var TEMP LOADBIGRED var TEMP

Load the next smallest or largest unred bit or the last red bit of var into ACC or TEMP

If out of bounds, jump to the next *oob* label

Load Next Unred Bit

LOADNEXT var ACC LOADNEXTRED var ACC

LOADNEXT var TEMP LOADNEXTRED var TEMP

LOADNEXTBIG var ACC LOADNEXTBIGRED var ACC

LOADNEXTBIG var TEMP LOADNEXTBIGRED var TEMP

Load the next smallest/largest unred bit or the last red bit of var into ACC/TEMP and toggle whether or not the bit is red

If out of bounds, jump to the next *oob* label

Store Unred Bit

STORE var ACC STORERED var ACC

STORE var TEMP STORERED var TEMP

STOREBIG var ACC STOREBIGRED var ACC

STOREBIG var TEMP STOREBIGRED var TEMP

Store the value of ACC or TEMP in the next smallest or largest unred bit or the last red bit of var

If out of bounds, jump to the next *oob* label

Store Next Unred Bit

STORENEXT var ACC STORENEXTRED var ACC

STORENEXT var TEMP STORENEXTRED var TEMP

STORENEXTBIG var ACC STORENEXTBIGRED var ACC

STORENEXTBIG var TEMP STORENEXTBIGRED var TEMP

Store the value of ACC or TEMP in the next smallest or largest unred bit or the last red bit of var and toggle whether or not the bit is red

If out of bounds, jump to the next *oob* label

Un-Read Variable

UNREAD var

Mark all bits of var as unred

Map Memory

MAP map

Change the value of ACC and/or TEMP based on a map function

A map is a series of a comma-seperated mappings within braces

Each mapping is a colon-seperated pair of states

A state can be an ACC value (0, 1, 2, or 3) or an “x”-seperated ACC value and TEMP value (0 or 1), depending on neighboring commands

{<acc>x<temp>:<acc>, <acc>x<temp>:<acc>, ...} if directly after loading into TEMP

{<acc>:<acc>x<temp>, <acc>:<acc>x<temp>, ...} if directly before storing TEMP

{<acc>x<temp>:<acc>x<temp>, <acc>x<temp>:<acc>x<temp>, ...} if directly between loading a bit into TEMP and storing TEMP into the same bit

{<acc>:<acc>, <acc>:<acc>, …} otherwise

If a possible ACC value or ACC and TEMP value pair isn’t specified, it defaults to not changing the value of the ACC or TEMP

Unconditional Branch

JUMP label

Go to the label

Conditional Branch

BRANCH label0 label1 label2 label3

Go to the label corresponding to the value of ACC

If null can be used in place of a label it will just go to the next line

Load Immediate

LOADI imm2 ACC

LOADI imm1 TEMP

Set ACC or TEMP to imm (0, 1, 2, or 3 for ACC and 0 or 1 for TEMP)

# Other Primitive Commands

Stationary Logical Shift

SLLs vard imm1

SRLs vard imm1

Shift vard 1 bit to the left or right

Replaces the vacated bit with imm (0 or 1)

ACC will contain the displaced bit (0 or 1)

Stationary Double Logical Shift

SLL2s vard imm2

SRL2s vard imm2

Shift vard 2 bits to the left or right

Replaces the vacated bits with imm (0, 1, 2, or 3)

ACC will contain the displaced bits (0, 1, 2, or 3)

Stationary Logical Negation

NOTs vard

Set vard to its logical negation

The ACC is preserved

Stationary Two’s Complement

COMPs vard

Set vard to its logical negation

ACC will contain 1 if vard was zero; or 0 otherwise

Stationary Logical Set

ZEROs vard imm1

Set all bits vard to imm (0 or 1)

The ACC is preserved

Stationary Add and Subtract Immediate

ADDIs vard imm2

SUBIs vard imm2

Increase or decrease vard by imm (0, 1, 2, or 3)

ACC will contain 1 if the operation overflowed; or 0 otherwise

Load Equal to Zero

LEQZ vard

ACC will contain 1 if vard is equal to zero; or 0 otherwise

# Higher Level Commands

Add and Subtract

ADD vard var0 var1

SUB vard var0 var1

Add or subtract var0 and var1 and store the result in vard

ACC will contain 1 if the operation overflowed; or 0 otherwise

Boolean Operation

BOL vard var0 var1 map

Apply a bitwise operation to var0 and var1 using a map function and store the result in vard

{<bit0>x<bit1>:<bitd>, <bit0>x<bit1>:<bitd>, ...}

Stationary Add and Subtract

ADDs vard var0

SUBs vard var0

NSUBs vard var0

Add var0 to vard , subtract var0 from vard or vard from var0 and store the result in vard

ACC will contain 1 if the operation overflowed; or 0 otherwise

Stationary Boolean Operation

BOLs vard var0 map

Apply a bitwise operation to var0 and vard using a map function and store the result in vard

{<bit0>x<bitd>:<bitd>, <bit0>x<bitd>:<bitd>, ...}

Using an empty map will copy the value of var0 to vard

Logical Shift

SLL vard var0 imm1

SRL vard var0 imm1

Shift var0 1 bit to the left or right and store the result in vard

Replaces the vacated bit with imm (0 or 1)

Store Less Than

SLT var0 var1

Test if var0 is less than var1

ACC will contain 1 if var0 is less than var1, 0 if they’re equal, or 3 if var0 is greater than var1

Multiply

MULT vard var0 var1

Multiply var0 by var1 and add the product to vard

var1 is set to zero

Multiply with Overflow Detection

MULTo vard var0 var1

Multiply var0 by var1 and add the product to vard, aborting if it overflows

var1 and vard are changed

ACC will contain 1 if the operation overflowed; or 0 otherwise

Divide

DIV vard varr var0 var1

Divide var0 by var1 and store the quotient in vard and the remainder in varr

Modulo

MOD vard var0 var1

Divide var0 by var1 and store the remainder in varr

Reciprocal

RECP vard varr var0

Find the reciprocal or var0 and store the fractional part of the quotient in vard and the fractional part of the remainder in varr

Push

PUSH var0

Push the value of var0 onto the stack

# Function Control

FUNC name param0 param1 …

END

All commands must be inside a function. A function starts with a function header “FUNC name param0 param1 …” giving the function’s name and a list of its parameters and ends with the command “END”. It starts execution on the line after the function header and returns when it reaches the end. FUNC and END may not be used within the function body. Within the function, the names of parameters can be used in place of variable names, maps, or immediates. When the function is called called with “name arg0 arg1 …”, the parameters will be filled in with the values of the corresponding arguments, which can be variable names, maps, or immediates. Functions can call other functions, but not cyclically or recursively. Jumps and branches within a function can only go to labels within that function.

PRIM name param0

A primitive is a special type of function. It can only contain LOAD[NEXT], STORE[NEXT], MAP, JUMP, BRANCH, and LOADI instructions, and can only use the one variable from the parameter. It always treats the variable as completely unred, but will not affect how much of the variable has been red.

# Stack Access

Up to one source variable in ADD, SUB, BOL, ADDs, SUBs, or BOLs can be replaced with $POP or $TOP, and the command will use the top value in the stack, removing it from the stack for $POP or leaving it on the stack for $TOP.

The command PUSH can be used to add a variable to the stack.

Linked jumping can be accomplished by placing one more bits of a pseudo-address on the stack with STORENEXTRED before the JUMP. To return, read the bit(s) from the stack with LOADNEXT and BRANCH to the respective label immediately after the JUMP.

# Future

JAL and JR instructions so the compiler can handle linked jumping