UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series B

MPASim — a Simulator for the MPA

Martti Forsell

Report B-1994-3

ACM D.1.3; D.3.2
UDK 681.3.02

MPASim — a Simulator for the MPA

Martti J. Forsell

Department of Computer Science, University of Joensuu, PB111, SF-80101 Joensuu, Finland, email: mforsell@cs.joensuu.fi

December 20, 1994

Abstract

The Minimal Pipeline Architecture (MPA) is an experimental VLIW architecture being developed at University of Joensuu. In this paper we introduce MPASim program, which is a simulator for the MPA. MPASim provides tools for executing and debugging programs written for MPA processors as well as an utility for translating DLX programs to MPA programs. It collects also statistical information on program execution for research purposes. MPASim runs in Macintosh environment and provides a reasonable graphical user interface.

Keywords: VLIW, instruction level parallelism, processor simulators

---

1. The Minimal Pipeline Architecture

The *Minimal Pipeline Architecture* (MPA) is a very simple superscalar load/store architecture using VLIW-style instruction scheduling policy [Forsell94]. The main parts of the MPA are arithmetic and logic units (ALU), arithmetic unit for integer condition codes (AIC), memory interfaces (M), registers (R) and sequencer (S). These parts are called functional units of the processor. Functional units are connected to each others by a large cross-bar switch (see Figure 2.1).

The main features of the MPA are:

- Multiple functional units connected with a cross-bar switch
- Uncoded VLIW-style instruction format with multiple word operands
- Two-level pipeline with general forwarding
- Single addressing mode
- Register-to-result bus operations
- Explicit write back instructions

See authors licentiate thesis [Forsell94] for further information on the MPA.

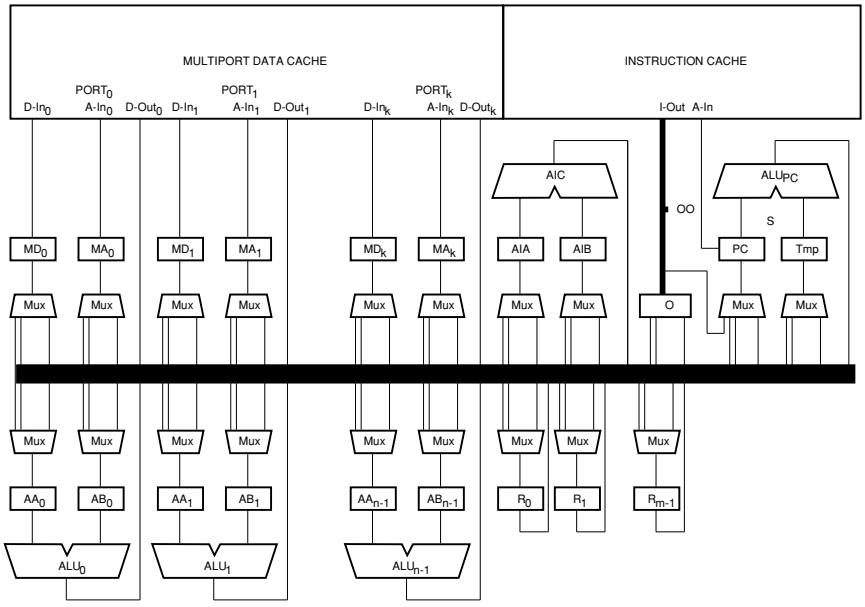

Figure 2.1 The block diagram of the MPA.

2. MPASim

MPASim is a simulator program for MPA processors. It provides a flexible environment with a reasonable graphical user interface to experiment MPA programming: The number of functional units including the number of register is configurable by the user. MacOS system calls are achieveable through traps and some programming, because MPASim data memory is embedded into the memory of host Macintosh.

MPASim provides tools for executing and debugging MPA programs as well as tools for collecting statistical information on program execution. It provides also an utility for translating and parallelizing DLX programs to MPA programs. (DLX is an experimental RISC architecture [Hennessy90]).

In this section we will introduce algorithms, coding conventions, traps and exceptions, and reserved words and error messages of MPASim.

2.1 Algorithms

Two main algorithms of MPASim are described here shortly—the *machine cycle algorithm* and the *translation algorithm*. A machine cycle of a MPA processor is simulated by following algorithm:

- Select operands according to a operation code on the instruction memory bus and send them to appropriate functional units
- Commit sequencer operation and system call if necessary
- Commit operations on ALUs, AIC, memory units according to operation code
- Fetch the next instruction from the instruction memory

DLX machine code programs are translated and parallelized into MPA programs by following algorithm:

- Split DLX instructions into MPA subinstructions
  - Fix the execution order of delayed branches
- Find dependencies between subinstructions
- Fill the instructions of a basic block from the beginning of the block
  - Rename functional units if necessary
  - Use forwarding if possible
- Remove empty instructions

2.2 Coding conventions

Program structure

A MPA program consists of three segments—the *code segment*, the *data segment* and the *stack and heap segment*. Assigning instructions and data to appropriate segments is described later in this section.

MPASim places the instructions of a program in the beginning of instruction memory. This area is called code segment. Operation code fetch outside of the code segment causes a bus error exception. Similarly MPASim places static data defined by storage definition directives in the beginning of data memory. This area is called data segment. The rest of data memory is left for dynamic data placement. It usually contains heap and stack, thus the name of the area is the stack and heap segment.

The starting point of a program must be indicated by _MAIN label in the beginning of code segment. The MEMSIZE symbol contains the address of the last word in data memory. This value can be used to initialize the stack.

Source text structure

Each line of MPA assembler source text is either a *machine instruction statement* or *directive statement*.

Machine instruction syntax

Machine instruction statements are written in three fields—the *label field*, the *instruction field* and the *comment field*. These fields must be separated by one or more spaces (ASCII code $20) or tabs (ASCII code $09), and must be written in the order given.

The instruction field consists of arbitrary number of instructions which are written in two fields—the *operation field* and the *operand field*.

The label field

The label field is the first field in a source text line. It may be empty or it may contain an identifier. The syntax rules for identifiers are given later in this section.

If the label field contains an identifier, it must begin in the first character position on the line. An identifier in the label field must be terminated by a colon (:), if the label is to be assigned to the data segment. Otherwise it must not terminate by a colon. In this case the label is assigned to the code segment.

The Operation field

The operation field contains the mnemonic operation code specifying the desired machine instruction. Valid mnemonics are described in appendix A. The assembler ignores uppercase and lowercase distinctions when reading the operation field.

The Operand field

Many instructions require operands as part of their specification. The operand field follows the operation field and must be separated from it by at least one space or tab. The operand field may be empty, or it may be composed of one or more subfields separated by commas. The operands required for MPA instructions are described in appendix A.

There are two types of operands—constants and identifiers. Complex expressions are not allowed in this version of MPASim.

Comments

You can insert a comment into your source text in two ways: as a comment field or as a *comment line*. Comments are ignored by the assembler and may contain any characters except return (ASCII $0D). Comments are intended for your use in documenting your program.

A comment field must begin with a semicolon (;). An entire line may be used for a comment by placing semicolon in the first character position of the line.

Identifiers

Names and labels are *identifiers*. The first character of an identifier must be an uppercase or lowercase letter (A..Z, a..z) or an underscore (_).

Directives

Directives are commands, which control the compilation of the code. There are four directives in MPASim:

ALIGN	Advance the location counter to the next multiple of a value
ASCII	Place a ASCII string constant in data segment
SPACE	Define a storage area in data segment
WORD	Place hexadecimal constants in data segment

MPA directives follow the general format for assembler statements. You write them in three fields, separated by spaces or tabs:

Label field	***Instruction field***		***Comment field***
	Operation field	***Operand field***	
[*identifier*]	*directive name*	*directive parameters*	[*comments*]

A directive name in operation field begins with period (.).

2.3 Traps and exception handling

Traps

MPASim uses same trap assignment convention as DLXSim (DLXSim is a simulator program for DLX. It is available with Hennessy's book [Hennessy90]). In current version only _EXIT (#0) and _PRINTF (#5) are implemented. See the documentation of DLXSim for more information.

Exception handling

Exceptions are situations, in which the processor is not able to continue execution normally. MPASim recognizes four kinds of exceptions—*bus errors*, *resets*, *unimplemented traps* and *halts*.

A bus error occurs when the processor tries to access memory outside the instruction, data or stack and heap segments. This causes MPASim to halt execution of the program. A memory reference outside segments is not allowed, because the data memory of the simulator is embedded into the memory of Machitosh. A random memory reference might crash the MacOS. A bus error normally indicates that there is a bug in the program.

A reset occurs when user selects Boot item in Process menu. This set whole simulator to initial state: Memories are emptied, functional units are initialized and PC is set to zero. The contents of the Command and Output windows are not changed.

An unimplemented trap exception occurs when an unimplemented trap subinstruction is executed. This causes MPASim to halt execution of the program. Calling a trap, which is defined in DLXSim but not in MPASim has no effect.

A halt occurs when user selects Halt item in Process menu. This causes MPASim to halt execution of the program.

2.4 Reserved words and error messages

Reserved words

Some words are reserved in MPA assembler, because they are used to express internal parts of the processor or some vital constants. Among them are:

_MAIN	Starting point of the MPA program
MEMSIZE	Address of the last memory location
A0, A1, ...	ALU outputs
M0, M1, ...	Memory unit outputs
R0, R1, ...	Registers
PC	Program counter

Error messages

Due to limitations of MPASim and MacOS 7.5 an alert containing one of the following messages can appear:

"You must run on 512Ke or later"
"Application Memory Size is too small"
"Not enough memory to run MPASim"
"Not enough memory to do Cut"
"Cannot do Cut"
"Cannot do Copy"
"Cannot exceed 32,000 characters with Paste"
"Not enough memory to do Paste"
"Cannot create window"
"Cannot exceed 32,000 characters"
"Cannot do Paste"
"Symboltable Overflow"
"Unknown Command"
"Illegal Operand"

You should press the OK button in the alert and try to avoid the action, which caused the problem.

Due to errors in DLX source code one of the following messages may appear during the DLX to MPA code translation. Numbers do not carry any specific information currently:

'Illegal mnemonic.'	100
'Label in delay slot.'	107
'Missing directive.'	107
'End of line ignored.'	108
'Illegal label.'	109
'SymbolTable overflow.'	111
'Missing directive.'	111
'Unknown operand.'	112
'End of line ignored.'	112
'Unknown instruction class.'	113

You should fix the DLX source code according to the message and retranslate the code. Note that, only one or two errors are reported at a time.

Due to errors in MPA source code one of the following messages may appear during the MPA assembler compilation. Number do not carry any specific information currently:

'Illegal ALU number.'	101
'Illegal memory bus number.'	102
'Illegal memory bus number.'	103
'Illegal register number.'	104
'Illegal operand number.'	105
'Illegal Mnemonic.'	106
'Missing directive.'	107
'End of line ignored.'	108
'Illegal label.'	109
'Illegal label.'	110
'Missing directive.'	111
'End of line ignored.'	112
'SymbolTable overflow'	114
'Unknown operand.'	115
'Unknown instruction class.'	116
'Program too large to fit in instruction memory.'	117

You should fix the MPA source code according to the message and recompile the code. Note that, only one or two errors are reported at a time.

3. The User Instructions

This section discusses installation, configuration, running and translation issues. Running requirements and a short description of graphical user interface for MPASim are also given.

Requirements

Requirements for running the MPASim program are

- Machintosh with M68020, M68030, M68040 or M68060 processor
- System 7.1 or newer
- 4 MB free RAM
- Large display is strongly recommended

Native PowerPC execution is not currently supported. At least recompilation of the program and rewriting of M32.a file would be needed to run native. The M680LC40 emulation in PowerMacs should, however, work fine with MPASim, because FPU instructions are not implemented in this version.

MPASim is quite demanding application. Apple Macintosh Quadra 700 (25 MHz 68040, 20 MB memory) executes 1800 MPA instructions per second.

Distribution

MPASim is distributed as a high density floppy disk or as a 162k self-extracting Stuffit Lite document *MPASim.sea*. When extracted the Stuffit Lite creates a folder containing following 14 files:

block.opt.s	9k	Example program for DLX
block.opt.M11	9k	Example program for M11
block.opt.M5	9k	Example program for M5
DLX.data	9k	Data file containing information on DLX instructions
MPASim	179k	Precompiled MPA Simulator application
MPASim.c	9k	C source file
MPASim.data	9k	Data file containing information on MPA instructions and configuration
MPASim.data.M11	9k	MPA configuration file for M11 simulation
MPASim.data.M5	9k	MPA configuration file for M5 simulation
MPASim.h	26k	Resource include file
MPASim.make	17k	Makefile of the program
MPASim.p	238k	Pascal source file
MPASim.r	34k	Resource definition source file
MPASimGlue.a	26k	M68020-assembler source file

MPASim was programmed with the Macintosh Programmers Workshop (MPW) programming environment version 3.3.1. The build process (compilation, linking and resource setting) requires 5400k free memory. Thus, you should set the preferred memory amount of MPW Shell to at least 5400k.

Installing the program

MPASim can be installed to a hard disk of a Macintosh by copying the contents of distribution floppy to a folder on the hard disk. The actual MPASim files are compressed into a self-extracting Stuffit Lite document (MPASim.sea). The extraction can be started just by double-clicking the icon of MPASim.sea file.

Configuring the simulator

The configuration of MPASim is described in MPASim.data text file, which is read when the program is started. The configuration file contains mnemonics of MPA instructions, size of data memory and number of functional units.

The contents of MPASim.data file is (only the numbers marked with italic typeface should be changed):

```
ADD                 2       0
SUB                 2       0
MUL                 2       0
.
.
.
INC                 0       8
OP                  1       9
END
InstructionMemorySize           1024
MemorySize                      65536
NumberOfALUs                    4
```

NumberOfMemoryBusses	*4*
NumberOfOperands	*4*
NumberOfRegisters	*33*
NumberOfRegisterWindows	*8*

The distribution of MPASim contains two example configuration files—MPASim.data.M5 and MPASim.data.M11. They represent five and eleven unit implementations of MPA respectively.

The graphical user interface

The graphical user interface of MPASim consists of a menu-bar, seven windows, some dialog boxes and alerts.

The structure of the menu-bar is:

	File	**Edit**	**Process**	**Breakpoint**	**Windows**
About MPASim...	New	*Undo*	Boot	Insert	Commands
	Open	----------	----------	Remove	Listing
	----------	Cut	Run	----------	Dump
	Close	Copy	Go	Clear All	Output
	Save	Paste	Pass		Processor
	Save As...	Clear	Step		Setup
	Revert		Halt		Statistics

	Import				
	Page Setup...				
	Print				

	Quit				

Items marked with *italic typeface* are not in use in this version. Most menu items have keyboard shortcuts.

The state of the processor, the contents of memories as well as commands, output messages and statistics are all presented in seven windows. The windows are:

Commands	Shows the commands that are given to MPASim
Memory	Shows the contents of the Data Memory
Listing	Shows the contents of the Instruction Memory in symbolic form
Output	Shows the output of simulator and possible system calls
Processor	Shows the state of the processor
Statistics	Shows statistical information on execution of the program
Setup	Not currently used

File references are made with standard MacOS dialog boxes that allow you to browse and select files. Some MacOS-related error messages are notified by alerts. The list of those error messages is shown in Section 2.2.

Managing windows

The windows are typical Macintosh environment windows with go-away region, drag region, grow region, full size region and two scrollbars.

They are directly connected to standard TextEdit managers with the exception of Memory window. You can write and delete text as well as copy and paste like in every normal MacOS application. The size of text is limited to 32000 bytes. This may cause problems when dealing with large programs or long simulations. You may wish to delete a part or the entire content of a window if you run out of space.

Importing DLX programs

One can translate DLX programs into MPA programs by Import utility of MPASim.

Translating a DLX program into a MPA program:

> *Select Import item in File menu*
> *Select the file, which should be translated using the standard file open dialog*

MPASim will now translate the DLX program into a MPA program and write it to MPA.OUT.USC and MPA.OUT.MPA files. The former is unscheduled version and the latter is parallelized version. The parallelization method is basic block compression without register renaming.

Executing MPA programs

Loading a MPA program into MPASim:

> *Select Open item in File menu*
> *Select the file, which should be loaded into MPASim using the standard file open dialog*

MPASim will now compile the program to machine language of MPA according to configuration used. If there are errors in the program error messages are shown in Output window, otherwise the compiled program will now be shown in Listing window in symbolic form.

Running a MPA program:

> *Select Run item in Process menu*

The program is executed until the processor is reset, the execution is halted, an _EXIT trap is met, a breakpoint is met or an exception occurs. At the same time statistical information on program execution is collected.

Halting the execution of a program:

> *Select Halt item in Process menu*

Continuing suspended execution:

Select Go item in Process menu

Executing a MPA program stepwise:

Select Step item in Process menu

Only the next instruction of the program is executed. This command will update the statistics.

Resetting the processor:

Select Boot item in Process menu

Executing a program without collecting statistical information:

Select Pass item in Process menu

Passing can be applied to a block of code by setting breakpoints into the beginning and the end of the block. The execution before the block should be done normally. The block should be executed with Pass command and the rest of the code should be executed normally with Go command.

Setting and removing breakpoints

You can insert up to 10 breakpoints into MPA code. The execution of a MPA program is halted if a breakpoint is met. This simplifies debugging and measuring of MPA programs.

Inserting a breakpoint into a MPA code:

Select the row in which the breakpoint should be inserted in Listing window
Select Insert item in Breakpoint menu

Symbol -> should now appear between address and instructions indicating the presence of a breakpoint.

Removing a breakpoint from a MPA code:

Select the row from which the breakpoint should be removed in Listing window
Select Remove item in Breakpoint menu

Symbol -> indicating the presence of the breakpoint should now disappear.

All breakpoints could be removed by selecting Clear All item in Breakpoint menu.

Collecting statistical information

MPASim collects statistical information on execution of the programs. The information is shown on statistics window.

The information is shown according to following format:

MPA - AN EXPERIMENTAL SCALABLE VLIW PROCESSOR SIMULATOR

© Copyright Martti Forsell, 1994
Department of Computer Science, University of Joensuu

STATISTICAL INFORMATION ON PROGRAM EXECUTION

PROCESSOR SETUP:

Number of ALUs	*: 4*
Number of Memory Busses	*: 4*
Number of Operands	*: 4*
Number of Registers	*: 32*
Start of Data Memory	*: 0x00953B60*
Size of Data Segment	*: 2000*
Size of Program Memory	*: 14*

GENERAL INFORMATION:

Clock Cycles	*: 394*

INSTRUCTION CLASS FREQUENCIES:

tALUI	*2692*
tAICI	*387*
tLDI	*99*
tSTI	*368*
tBRAI	*287*
tJMPI	*1*
tREGI	*1361*
tWINI	*0*
tINTI	*106*
tOPI	*0*

INSTRUCTION FREQUENCIES:

ADD	*1360*	
SUB	*2*	
NOP	*1330*	
SEQ	*99*	
SLT	*288*	
LD	*99*	
ST	*368*	
BEQZ	*98*	*73*
BNEZ	*189*	*162*
TRAP	*1 0*	
CMD	*106*	
WB	*1361*	

Statistical information produced by MPASim can be transferred to worksheet programs like Microsoft Excel for further processing and drawing diagrams by copying the content of statistics window to clipboard and pasting it to the worksheet.

Bugs and limitations

Being a four week project resulting to 11000 lines of program code MPASim has several limitations and bugs.

- Internal data structures are static
- Instruction memory is small due to unefficient coding of instruction words
- DLX to MPA translation does not rename registers
- The content of a window is limited to 32000 characters
- Floating point instructions are not implemented
- Reconfiguration must be done by hand editing a text file and rerunning the program
- Only two traps are actually implemented
- Assemblers have poor error recovery process

References

[Forsell94] M. Forsell, Design and Analysis of Some Chip Level Parallel Architectures, *Licentiate thesis*, Department of Computer Science, University of Joensuu, Finland, 1994.

[Hennessy90] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

Appendices

A. The complete instruction set of the MPA

The symbols used for describing MPA operations and instructions

Ax	Result of ALU x
AAx	Input register 1 of ALU x
ABx	Input register 2 of ALU x
AIA	Input register 1 of ALU for integer condition codes
AIB	Input register 2 of ALU for integer condition codes
AIC	Result of ALU for integer condition codes
d	32-bit immediate value
I[X]	Location x of instruction memory
Mx	Result of memory x operation
Mx[y]	Location y of memory x
MAx	Address register for memory x
MDx	Data register for memory x
MuxY	Multiplexer for Y
O	Operation register
Ox	Operand x of instruction register
OO	Result of operation code fetch
PC	Program counter
RA	Return address
Rx	Contents of register x
Xx	Ax, Mx, Ox or Rx

The instructions of the MPA are

Load		
LDBn	Xx	Load byte from memory n address Xx
LDBUn	Xx	Load byte from memory n address Xx unsigned
LDHn	Xx	Load halfword from memory n address Xx
LDHUn	Xx	Load halfword from memory n address Xx unsigned
LDn	Xx	Load word from memory n address Xx
Store		
STBn	Xx,Xy	Store byte Xx to memory n address Xy
STHn	Xx,Xy	Store halfword Xx to memory n address Xy
STn	Xx,Xy	Store word Xx to memory n address Xy
Write Back		
WBn	Xx	Write Xx to register Rn. Old value of Rn is preserved if x is r-1.
Operand Input		
OPn	d	Input value d into operand n
Align		
ALBn	Xx,Xy	Align load byte Xx by Xy in ALU n
ALBUn	Xx,Xy	Align load byte Xx by Xy in ALU n unsigned
ALHn	Xx,Xy	Align load halfword Xx by Xy in ALU n
ALHUn	Xx,Xy	Align load halfword Xx by Xy in ALU n unsigned
ASSBn	Xx,Xy	Align store source byte Xx by Xy in ALU n
ASSHn	Xx,Xy	Align store source halfword Xx by Xy in ALU n
ASDBn	Xx,Xy	Align store destination byte Xx by Xy in ALU n
ASDHn	Xx,Xy	Align store destination halfword Xx by Xy in ALU n
Arithmetic		
ADDn	Xx,Xy	Add Xx and Xy in ALU n
SUBn	Xx,Xy	Subtract Xy from Xx in ALU n
MULn	Xx,Xy	Multiply Xx by Xy in ALU n
MULUn	Xx,Xy	Multiply Xx by Xy in ALU n unsigned
DIVn	Xx,Xy	Divide Xx by Xy in ALU n
DIVUn	Xx,Xy	Divide Xx by Xy in ALU n unsigned
MODn	Xx,Xy	Determine Xx modulo Xy in ALU n
MODUn	Xx,Xy	Determine Xx modulo Xy in ALU n unsigned
Logical		
ANDn	Xx,Xy	And of Xx and Xy in ALU n
ORn	Xx,Xy	Or of Xx and Xy in ALU n
XORn	Xx,Xy	Exclusive or of Xx and Xy in ALU n
ANDNn	Xx,Xy	And not of Xx and Xy in ALU n
ORNn	Xx,Xy	Or not of Xx and Xy in ALU n
XNORn	Xx,Xy	Exclusive nor of Xx and Xy in ALU n
SHRn	Xx,Xy	Shift right Xx by Xy in ALU n
SHLn	Xx,Xy	Shift left Xx by Xy in ALU n
SHRAn	Xx,Xy	Shift right Xx by Xy in ALU n arithmetic
NOPn		No operation in ALU n

RORn	Xx,Xy	Rotate right Xx by Xy in ALU n	
ROLn	Xx,Xy	Rotate left Xx by Xy in ALU n	
Compare			
SEQ	Xx,Xy	Set IC if Xx equals Xy	
SNE	Xx,Xy	Set IC if Xx not equals Xy	
SLT	Xx,Xy	Set IC if Xx is less than Xy	
SLE	Xx,Xy	Set IC if Xx is less than or equals Xy	
SGT	Xx,Xy	Set IC if Xx is greater than Xy	
SGE	Xx,Xy	Set IC if Xx is greater than or equals Xy	
ADDCC	*Xx,Xy*	*Add Xx and Xy and set CC*	
SUBCC	*Xx,Xy*	*Subtract Xy from Xx and set CC*	
MULCC	*Xx,Xy*	*Multiply Xx by Xy and set CC*	
MULUCC	*Xx,Xy*	*Multiply Xx by Xy unsigned and set CC*	
DIVCC	*Xx,Xy*	*Divide Xx by Xy and set CC*	
DIVUCC	*Xx,Xy*	*Divide Xx by Xy unsigned and set CC*	
MODCC	*Xx,Xy*	*Determine Xx modulo Xy and set CC*	
MODUCC	*Xx,Xy*	*Determine Xx modulo Xy unsigned and set CC*	
ADDXn	*Xx,Xy*	*Add with carry Xx and Xy in ALU n*	
SUBXn	*Xx,Xy*	*Subtract with carry Xy from Xx in ALU n*	
ADDXCC	*Xx,Xy*	*Add with carry Xx and Xy and set CC*	
SUBXCC	*Xx,Xy*	*Subtract with carry Xy from Xx and set CC*	
TADDn	*Xx,Xy*	*Tagged add Xx and Xy in ALU n*	
TSUBn	*Xx,Xy*	*Tagged subtract Xy from Xx in ALU n*	
TADDCC	*Xx,Xy*	*Tagged add Xx and Xy and set CC*	
TSUBCC	*Xx,Xy*	*Tagged subtract Xy from Xx and set CC*	
ANDCC	*Xx,Xy*	*And of Xx and Xy in ALU n and set CC*	
ORCC	*Xx,Xy*	*Or of Xx and Xy in ALU n and set CC*	
XORCC	*Xx,Xy*	*Exclusive or of Xx and Xy in ALU n and set CC*	
ANDNCC	*Xx,Xy*	*And not of Xx and Xy in ALU n and set CC*	
ORNCC	*Xx,Xy*	*Or not of Xx and Xy in ALU n and set CC*	
XNORCC	*Xx,Xy*	*Exclusive nor of Xx and Xy in ALU n and set CC*	
Control			
BEQZ	Ox	Branch to Ox if IC equals zero	
BNEZ	Ox	Branch to Ox if IC not equals zero	
FBT	Ox	Floating Branch to Ox if FC equals zero	
FBF	Ox	Floating Branch to Ox if FC not equals zero	
JMP	Xx	Jump to Xx	
JMPL	Xx	Jump and link PC+1 to register RA	
TRAP	Xx	Trap	
BA	*Ox*	*Branch always to Ox*	
BN	*Ox*	*Branch newer to Ox*	
BNE	*Ox*	*Branch on not equal to Ox*	*(BNZ)*
BE	*Ox*	*Branch on equal to Ox*	*(BZ)*
BG	*Ox*	*Branch on greater to Ox*	
BLE	*Ox*	*Branch on less than or equal to Ox*	

BGE	*Ox*	*Branch on greater than or equal to Ox*	
BL	*Ox*	*Branch on less than to Ox*	
BGU	*Ox*	*Branch on greater to Ox unsigned*	
BLEU	*Ox*	*Branch on less than or equal to Ox unsigned*	
BCC	*Ox*	*Branch on carry clear to Ox*	*(BGEU)*
BCS	*Ox*	*Branch on carry set to Ox*	*(BLU)*
BPOS	*Ox*	*Branch on positive to Ox*	
BNEG	*Ox*	*Branch on negative to Ox*	
BVC	*Ox*	*Branch on overflow clear to Ox*	
BVS	*Ox*	*Branch on overflow set to Ox*	

Register window

SAVE		*Save register window*
REST		*Restore register window*

Floating Point

FADDn	Xx,Xy	Floating add Xx and Xy in ALU n
FSUBn	Xx,Xy	Floating subtract Xy from Xx in ALU n
FMULn	Xx,Xy	Floating multiply Xx by Xy in ALU n
FDIVn	Xx,Xy	Floating divide Xy by Xx in ALU n
FMODn	Xx,Xy	Floating Xx modulo Xy in ALU n
FSEQ	Xx,Xy	Floating set FC if Xx equals Xy
FSNE	Xx,Xy	Floating set FC if Xx not equals Xy
FSLT	Xx,Xy	Floating set FC if Xx is less than Xy
FSLE	Xx,Xy	Floating set FC if Xx is less than or equals Xy
FSGT	Xx,Xy	Floating set FC if Xx is greater than Xy
FSGE	Xx,Xy	Floating set FC if Xx is greater than or equals Xy
DADDn	Xx,Xy	Double add Xx and Xy in ALU n
DSUBn	Xx,Xy	Double subtract Xy from Xx in ALU n
DMULn	Xx,Xy	Double multiply Xx by Xy in ALU n
DDIVn	Xx,Xy	Double divide Xy by Xx in ALU n
DMODn	Xx,Xy	Double Xx modulo Xy in ALU n
DSEQ	Xx,Xy	Double set FC if Xx equals Xy
DSNE	Xx,Xy	Double set FC if Xx not equals Xy
DSLT	Xx,Xy	Double set FC if Xx is less than Xy
DSLE	Xx,Xy	Double set FC if Xx is less than or equals Xy
DSGT	Xx,Xy	Double set FC if Xx is greater than Xy
DSGE	Xx,Xy	Double set FC if Xx is greater than or equals Xy
CVTFIn	Xx	Convert floating Xx to integer in ALU n
CVTFDn	Xx	Convert floating Xx to double in ALU n
CVTDIn	Xx	Convert double Xx to integer in ALU n
CVTDFn	Xx	Convert double Xx to floating in ALU n
CVTIFn	Xx	Convert integer Xx to floating in ALU n
CVTIDn	Xx	Convert integer Xx to double in ALU n