MACQUARIE UNIVERSITY

SYDNEY ~ AUSTRALIA

END-OF-YEAR EXAMINATIONS 2009

Unit: **COMP226 – Computer Architecture**

Date: **Monday 23 November 2009 at 1:50 PM**

Time Allowed: **Three (3) hours, plus 10 minutes reading time**

Total Number of Questions: **Nine (9)**

Total Marks: **180**

Instructions:

Answer **all** questions**.**

The questions are *not* of equal value.

The examination is divided into *two* parts: A and B.

The parts are of equal value.

Use a separate book for each part.

Indicate the part clearly on the outside of each book.

Write your name and student number clearly on each book.

The marks allocated to a question are written next to the question.

If you wish to spread your time evenly, you should spend approximately one minute for each mark.

Every attempt has been made to make the questions unambiguous. However, if you are not sure what a question is asking, make some *reasonable* assumption and state it at the beginning of your answer.

Materials Permitted:

No dictionaries permitted.
No calculators permitted.

PART A [90 marks] (Use a separate book)

For this part you will require the ISEM Reference Card which is attached to this examination paper.

1. [30 marks]

For each of the following, work out your solution first on scrap paper and then copy it neatly into your book. Comment your solutions clearly. Do **not** use any synthetic instructions in your answers.

Write a fragment of SPARC (isem) assembler which will print *n* stars '*' on a line, starting at the current cursor position, where *n* is the non-negative number stored in the word at the location labelled `number:`

Your code should work for any non-negative value of *n*. Do not worry that for larger values the line of stars would wrap on your screen.

Now write another fragment of SPARC assembler which will work the same way if the number *n* is positive (viewed as a 2's-complement number stored in a word), but will print the stars to the **left** of the cursor position if the number *n* is negative. How do you access positions to the left of the cursor? Well, the ascii character '`\b`' is the backspace character. If you print '`\b`' in the normal way the cursor moves one space to the left. If you then print another character it will appear there, one space to the left of the normal cursor position.

Of course, this second part adds some complications. For example, if the number *n* is 2000, your code from above will just print 2000 stars to the right of the cursor. But the backspace character can only move back on the **current** line, so there will be a certain number of stars that you can print to the left of the cursor, depending on where the cursor begins. We will suppose that the cursor is positioned after 13 characters on the current line. That means that you could print any number of stars on the left for *n* between -1 and -13. However, I only want you to print the correct number of stars to the left for *n* between -1 and -10. If *n* is less than -10 you should print a line that begins with three dots '.' to indicate that there are more stars not shown to the left, and then the 10 stars that fit to the left of the original cursor position.

So, in summary your second fragment will print the correct number of stars for *n* from -10 up to the maximum positive value of a signed word, with negative *n* indicated by stars to the left of the original cursor position, and positive *n* by stars to the right. Furthermore, if n is less than -10 your code will print 10 stars to the left of the cursor position preceded (further left) by three dots to indicate that there should really be more than 10 of them.

Lastly, what is the maximum positive value of a signed word? Write the number on a separate line at the end of your answer to this question.

2. [30 marks]

Assemble the following fragment of SPARC assembly code into SPARC machine code.

```
        set 1, %r4
        cmp %r1, 1
        be  done
        nop
        dec %r1
        jmpl %r3, %r15
        nop
        add %r4, %r5, %r4
done:   ld [%sp], %r1
        add %sp, 4, %sp
```

Show your working for each line of code, and then present your final answer as ten lines of 32 bit words written in hexadecimal. The SPARC reference card is included at the back of this examination paper.

What do you think the two lines of code which begin at `done` are doing? Why might that need to be done?

3. [30 marks]

(a) [15 marks] Consider each of the following expressions in the Hardware Description Language (HDL) we studied in lectures. In each case, if possible write a single line of SPARC assembler which will have the same effect. If this is not possible explain why.

`r3 <-` $(M[r2]_0)^{24}$ `## M[r2]`

`r3 <-` 0^{16} `## M[r3] ## M[r3 + 1]`

`r3 <- M[r2+200] ## M[r2+201] ## M[r2+202] ## M[r2+203]`

(b) [15 marks] Draw a diagram which shows how to construct an `OR` gate using three transistors. Next draw diagrams which show how to construct a half-adder and a multiplexor with 8-bit data input, 3-bit control input and 1-bit data output. You may use `AND`, `OR` and `NOT` gates.

PART B [90 marks] (Use a separate book)

4. [6 marks]

A CPU designer is comparing two different memory hierarchy designs – one with only L1 caches and one with an additional L2 cache. The cache designs have the following characteristics:

L1 cache: Hit time 0.25 ns, miss rate 3%, capacity 64 KB, associativity 2, line size 32 Bytes.

L2 cache: Hit time 2 ns, global miss rate 0.6%, capacity 512 KB, associativity 4, line size 32 Bytes.

The main memory access time is 50 ns.

(a) [3 marks] Compute the Average Memory Access Time for a memory hierarchy comprising only the L1 cache and main memory.

(b) [3 marks] Compute the Average Memory Access Time for a memory hierarchy comprising both the L1 and L2 caches, and main memory.

5. [30 marks]

The diagram on the next page represents the memory hierarchy of the AMD Opteron processor. With reference to this diagram, answer the following questions. Each question is worth 3 marks.

(a) What is the component labelled DTLB? What does it do?

(b) What is the component labelled L2 DTLB? What does it do?

(c) What is the component labelled DCACHE? What does it do? How is it related to the DTLB?

(d) What is the component labelled L2 CACHE? What does it do? How is it related to the other components?

(e) What is the maximum virtual memory address space supported by this processor? Show your working.

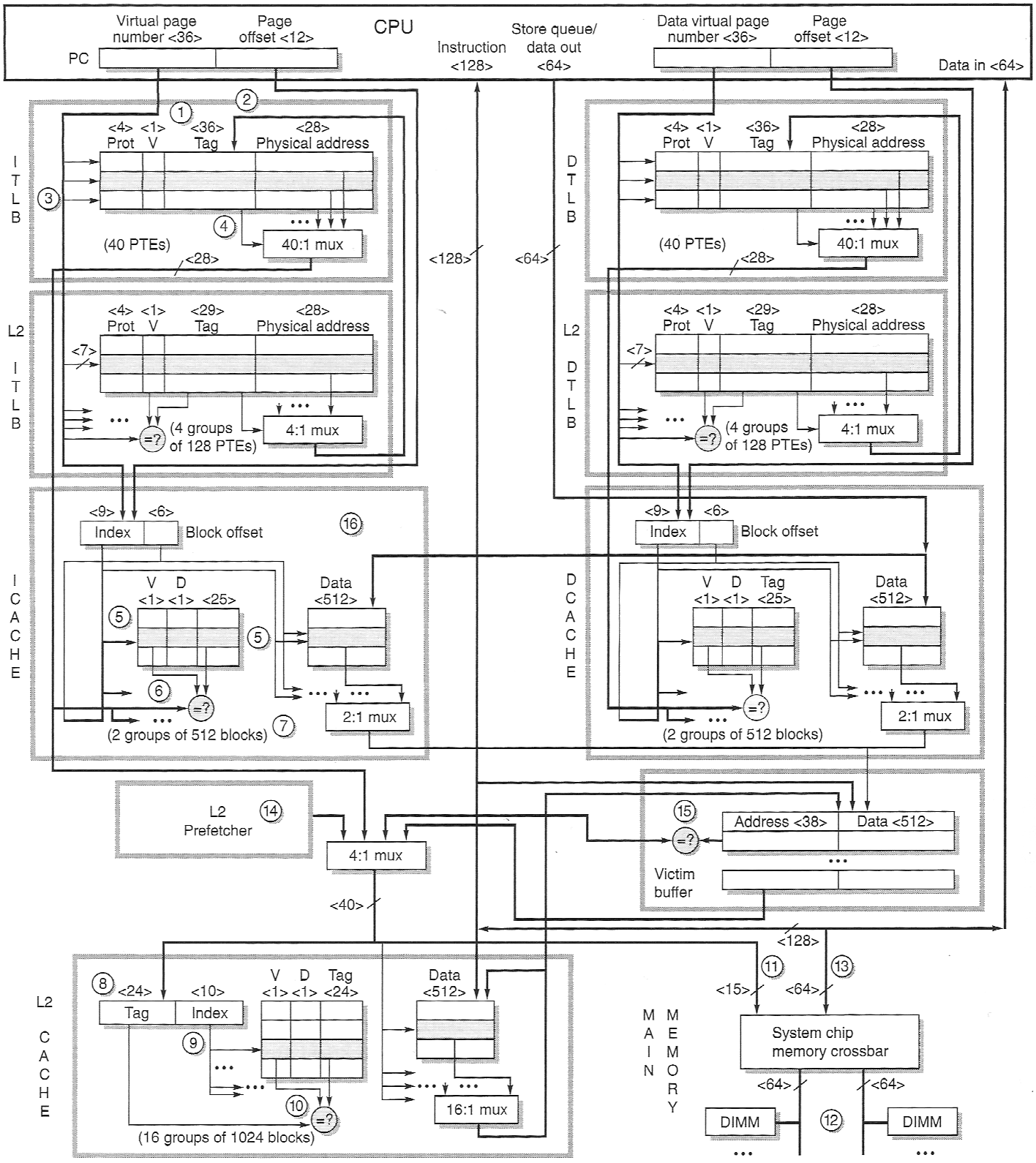
(f) What is the maximum physical memory address space supported by this processor? Show your working.

(g) What is the size of a VM page as shown in this diagram? Show your working.

(h) The associativity of the ICACHE is 2-way. How many sets are in the ICACHE? How many cache lines? What is the capacity of the ICACHE? Explain your reasoning.

(i) What is the write policy for the DCACHE? Is it write-through or write-back? Would you expect it to be write-allocate or write-noallocate? Explain your reasoning.

(j) What do you think that the component labelled L2 PREFETCHER does? How does it help the memory hierarchy perform well?


CPU
Virtual page number <36>
Page offset <12>
PC
Instruction <128>
Store queue/ data out <64>
Data virtual page number <36>
Page offset <12>
Data in <64>
ITLB
DTLB
<4> Prot <1> V <36> Tag <28> Physical address
(40 PTEs)
40:1 mux
<28>
<128>
<64>
L2 ITLB
L2 DTLB
<29> Tag
<7>
(4 groups of 128 PTEs)
4:1 mux
ICACHE
DCACHE
<9> Index
<6> Block offset
V <1> D <1> Tag <25>
Data <512>
(2 groups of 512 blocks)
2:1 mux
L2 Prefetcher
Address <38>
Victim buffer
<40>
L2 CACHE
<24> Tag
<10> Index
(16 groups of 1024 blocks)
16:1 mux
MAIN MEMORY
<15>
System chip memory crossbar
DIMM

6. [15 marks]

Below are two program modules ("one" and "two") that perform the same computation on arrays. One of them has been optimised to improve execution speed through making more efficient use of the memory hierarchy of the computer.

You may assume that the programs are to be run on an UltraSPARC processor identical to the processor in "titanic.ics.mq.edu.au". This processor has 32-bit integers. The processor has separate I and D first-level caches, each 64 KB 4-way set associative with a 32 byte cache line. The first-level caches are virtually indexed and physically tagged with a pseudo-random replacement policy. The second level cache is 2MB while the third level cache is 32MB. Both the L2 and L3 caches are 4-way set associative with a pseudo-LRU replacement policy; they are writeback, write-allocate caches.

(a) [10 marks] Compare the two modules. Identify which module you expect to run faster and why.

Identify each of the changes in your preferred module compared to the other module, and explain how and why it affects the execution speed. In your explanation, refer explicitly to the different types of cache misses where appropriate, in addition to considering page faults or other memory hierarchy performance issues.

(b) [5 marks] Are there any aspects of your preferred module that you think could be further improved? Explain

```
void one (int x[512][512], y[512][512], z[512])
{
   int row, col;
   /* Add arrays x and y into array y */
   for (col = 0; col < 512; col++)
      for (row = 0; row < 512; row++)
         y[row][col] = x[row][col] + y[row][col];
   /* Zero the sums */
   for (col = 0; col < 512; col++)
      z[col] = 0;
   /* Compute sums of the columns of y into z */
   for (col = 0; col < 512; col++)
      for (row = 0; row < 512; row++)
         z[col] = z[col] + y[row][col];
   return;
}

void two (int x[512][512], y[512][512], z[512])
{
   int row, col;
   /* Zero the sums */
   for (col = 0; col < 512; col++)
      z[col] = 0;
   for (row = 0; row < 512; row++)
      for (col = 0; col < 512; col++)
      {
         /* Add arrays x and y into array y */
         y[row][col] = x[row][col] + y[row][col];
         /* Compute sums of the columns of y into z */
         z[col] = z[col] + y[row][col];
      }
   return;
}
```

7. [10 marks]

(a) [2 marks] What is a page fault?

(b) [3 marks] Describe how a page fault is handled – what steps must be taken when a page fault happens?

(c) [2 marks] What is thrashing? How would you recognise thrashing if it was happening on your PC?

(d) [3 marks] What can be done to reduce or prevent thrashing? Describe at least two different ways to reduce or prevent thrashing, and explain. You may consider the hardware, the Operating System, the programmer, and/or the computer user.

8. [15 marks]

The tables below and on the next page contain important information about modern hard disks and modern flash drives. Use the data in this table to compare hard disk storage with flash drive storage.

(a) [10 marks] Summarise the current characteristics of hard drives vs the current characteristics of flash storage, and compare the two technologies.

(b) [5 marks] Evaluate flash storage compared to hard disk storage and suggest at least two application areas where hard drives are likely to be the most appropriate storage medium, and at least two application areas where flash drives could be more appropriate, giving reasons to support your conclusions.

Flash Storage Devices

Parameter	Kingston Data Traveler 150 USB flash drive	Intel X-25M Solid State Disk	Corsair Solid State Disk P256
Capacity	64GB	160GB	256GB
Bytes/sec	not available	512	512
Bus	USB	SATA-300	SATA-300
Data transfer rate R/W	20/10 MB/s	250/70 MB/s	220/180 MB/s
Latency R/W	not available	65/85 µs	not available
Dimensions (mm)	78 x 22 x 12	35 x 43 x 3	9 x 70 x 102
Mass	40 g	80 g	80 g
MTBF	not available	>1 200 000 hr	>1 000 000 hr
Shock tolerance	not available	1500 g	1500 g
Price (2009)	$166	$650	$900

Hard Disk Storage Devices

Parameter	Samsung 2.5” 160GB SATA	Western Digital 3.5” 320GB IDE	Seagate Barracuda 3.5” 1.5TB SATA
Form factor	2.5”	3.5”	3.5”
Capacity	160GB	320GB	1500GB
Bytes/sec	512	512	512
Bus	SATA-150	IDE	SATA-300
External transfer rate (host to/from buffer)	150 MB/s	100 MB/s	300 MB/s
Internal transfer rate (buffer to/from disk)	66 MB/s	not available	120 MB/s
Seek time track-to-track	2 ms	2 ms	not available
Seek time average	12 ms	8.9 ms	8.5 ms
Latency average	5.6 ms	4.2 ms	4.2 ms
Rotation speed	5400 RPM	7200 RPM	7200 RPM
Buffer (cache)	8 MB	8 MB	32 MB
Dimensions (mm)	70 x 100 x 9	102 x 147 x 25	102 x 147 x 25
Mass	100 g	630 g	720 g
MTBF	330 000 hr	not available	750 000 hr
Price (2009)	$82	$132	$219

9. [14 marks]

Parallel architectures are important for the future of computing.

(a) [6 marks] Parallel architectures are sometimes classified using the acronyms “SIMD”, “MIMD”. There is also “SISD” and “MISD”, but one of these does not make sense and the other is not really a parallel architecture.

i) What does SIMD stand for? Explain what a SIMD parallel architecture is like and give an example.

ii) What does MIMD stand for? Explain how a MIMD parallel architecture differs from SIMD, and give an example of an MIMD architecture.

iii) What does SISD stand for? What does MISD stand for? Which of these is sensible and what does it mean? Give an example.

(b) [8 marks] Another way to think about parallelism is in terms of the implementation of processors. What do each of the following terms mean? How do they relate to the SIMD, MIMD, etc classification? Are there special features required in the processor implementation to make the execution model fit SIMD, MIMD, etc?

i) Pipelined instruction stream.

ii) Superscalar architecture.

iii) Hyper-threading.

iv) Multi-core architecture.

-- That’s all! Now check your answers --

ISEM Reference Card (Integer Unit)

• Notation

Notation	Meaning
reg	integer register (%r0–%r31); subscript denotes instruction field (e.g., reg_{rd})
const	unsigned integer value; subscript denotes size in bits (e.g., $const_{22}$)
disp	signed integer displacement; subscript denotes size in bits (e.g., $disp_{22}$)
siconst	2's complement signed integer value; subscript denotes size in bits (e.g., $siconst_{30}$)
address	reg_{rs1}
	$reg_{rs1} + reg_{rs2}$
	$reg_{rs1} + siconst_{13}$
	$reg_{rs1} - siconst_{13}$
	$siconst_{13}$
	$siconst_{13} + reg_{rs1}$
regaddr	reg_{rs1}
	$reg_{rs1} + reg_{rs2}$
label	an instruction label
asi	address space identifier (0 to 255)

• Registers

Name	Meaning
%r0–%r31	integer registers
%g0–%g7	global regs (%r0–%r7)
%o0–%o7	output regs (%r8–%r15)
%l0–%l7	local regs (%r16–%r23)
%i0–%i7	input regs (%r24–%r31)
%fp	frame pointer (%i6)
%sp	stack pointer (%o6)
%asr1–%ars31	ancillary state registers
%psr	processor status register
%tbr	trap base register
%y	multiply/divide register

%r0 always holds the 32-bit integer value 0.

%psr has the following structure:

31:28	27:24	23	22	21	20	19:14	13	12	11:8	7	6	5	4:0
impl	ver	n	z	v	c	reserved	EC	EF	PIL	S	PS	ET	CWP

%tbr has the following structure:

31:12	11:4	3:0
Trap base address	trap type	0000

• Store Instructions

Assembler Syntax

operation reg_{rd}, [*address*]
operationa reg_{rd}, [*regaddr*]*asi*

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
11	rd	op3	rs1	0	asi	rs2
11	rd	op3	rs1	1	$siconst_{13}$	

Operations

operation	op3	operation (alt. space)	op3
stb	000101	stba	010101
sth	000110	stha	010110
st	000100	sta	010100
std	000111	stda	010111

• Load Instructions

Assembler Syntax

operation [*address*], reg_{rd}
operationa [*regaddr*]*asi*, reg_{rd}

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
11	rd	op3	rs1	0	asi	rs2
11	rd	op3	rs1	1	$siconst_{13}$	

Operations

operation	op3	operation (alt. space)	op3
ldsb	001001	ldsba	011001
ldsh	001010	ldsha	011010
ldub	000001	lduba	010001
lduh	000010	lduha	010010
ld	000000	lda	010000
ldd	000011	ldda	010011
ldstub	001101	ldstuba	011101
swap	001111	swapa	011111

• SETHI Instructions

Assembler Syntax

sethi $const_{22}$, reg_{rd}
sethi %hi(*const*), reg_{rd}

Instruction Format

31:30	29:25	24:22	21:0
00	rd	100	$const_{22}$

• Arithmetic and Logical Instructions

Assembler Syntax

operation reg_{rs1}, reg_{rs2}, reg_{rd}
operation reg_{rs1}, $siconst_{13}$, reg_{rd}

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	rd	op3	rs1	0	unused (zero)	rs2
10	rd	op3	rs1	1	$siconst_{13}$	

Operations

operation	op3	operation	op3
add	000000	addcc	010000
addx	001000	addxcc	011000
and	000001	andcc	010001
andn	000101	andncc	010101
		mulscc	100100
or	000010	orcc	010010
orn	000110	orncc	010110
udiv	001110	udivcc	011110
umul	001010	umulcc	011010
sdiv	001111	sdivcc	011111
smul	001011	smulcc	011011
sub	000100	subcc	010100
subx	001100	subxcc	011100
taddcc	100000	taddcctv	100010
tsubcc	100001	tsubcctv	100011
xor	000011	xorcc	010011
xnor	000111	xnorcc	010111

• Shift Instructions

Assembler Syntax

operation reg_{rs1}, reg_{rs2}, reg_{rd}
operation reg_{rs1}, $const_5$, reg_{rd}

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	rd	op3	rs1	0	unused (zero)	rs2
10	rd	op3	rs1	1	unused (zero)	$const_5$

Operations

operation	op3
sll	100101
sra	100111
srl	100110

• Branch Instructions

Assembler Syntax

operation *label*
operation,a *label*

Instruction Format

31:30	29	28:25	24:22	21:0
00	a	cond	010	$disp_{22}$

Operations

operation	cond	operation	cond
bn	0000	ba	1000
be (bz)	0001	bne (bnz)	1001
ble	0010	bg	1010
bl	0011	bge	1011
bleu	0100	bgu	1100
bcs (blu)	0101	bcc (bgeu)	1101
bneg	0110	bpos	1110
bvs	0111	bvc	1111

• Jump and Link Instructions

Assembler Syntax

jmpl *address*, reg_{rd}

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	rd	111000	rs1	0	unused (zero)	rs2
10	rd	111000	rs1	1	$siconst_{13}$	

• Return from Trap Instructions

Assembler Syntax

rett *address*

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	00000	111001	rs1	0	unused (zero)	rs2
10	00000	111001	rs1	1	$siconst_{13}$	

• Call and Link Instructions

Assembler Syntax

call *label*

Instruction Format

31:30	29:0
01	$disp_{30}$

• Restore and Save Instructions

Assembler Syntax

operation reg_{rs1}, reg_{rs2}, reg_{rd}
operation reg_{rs1}, $siconst_{13}$, reg_{rd}

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	rd	op3	rs1	0	unused (zero)	rs2
10	rd	op3	rs1	1	$siconst_{13}$	

Operations

operation	op3
restore	111101
save	111100

• Trap Instructions

Assembler Syntax

operation reg_{rs1}, reg_{rs2}, reg_{rd}
operation reg_{rs1}, $siconst_{13}$, reg_{rd}

Instruction Formats

31:30	29	28:25	24:19	18:14	13	12:5	4:0
10	r	cond	111010	rs1	0	unused (zero)	rs2
10	r	cond	111010	rs1	1	$siconst_{13}$	

Operations

operation	cond	operation	cond
tn	0000	ta	1000
te (tz)	0001	tne (tnz)	1001
tle	0010	tg	1010
tl	0011	tge	1011
tleu	0100	tgu	1100
tcs (tlu)	0101	tcc (tgeu)	1101
tneg	0110	tpos	1110
tvs	0111	tvc	1111

• The NOP Instruction

Assembler Syntax

nop

Instruction Format

31:30	29:25	24:22	21:0
00	00000	100	0000000000000000000000

• The UNIMP Instruction

Assembler Syntax

unimp *const*$_{22}$

Instruction Format

31:30	29:25	24:22	21:0
00	reserved	000	$const_{22}$

• The STBAR Instruction

Assembler Syntax

stbar

Instruction Format

31:30	29:25	24:19	18:14	13:0
10	00000	101000	01111	unused (zero)

• Read State Register Instructions

Assembler Syntax

rd *statereg, rd*

Instruction Format

31:30	29:25	24:19	18:14	13:0
10	rd	op3	rs1	unused (zero)

Register Encodings

state register	op3	rs1
%y	101000	0
%asr1–%asr31	101000	1–31
%psr	101001	reserved
%wim	101010	reserved
%tbr	101011	reserved

• Write State Register Instructions

Assembler Syntax

wr reg_{rs1}, reg_{rs2}, *statereg*
wr reg_{rs1}, $siconst_{13}$, *statereg*

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	rd	op3	rs1	0	unused (zero)	rs2
10	rd	op3	rs1	1	$siconst_{13}$	

Register Encodings

state register	op3	rd
%y	110000	0
%asr1–%asr31	110000	1–31
%psr	110001	reserved
%wim	110010	reserved
%tbr	110011	reserved

• FLUSH Instructions

Assembler Syntax

flush *address*

Instruction Formats

31:30	29:25	24:19	18:14	13	12:5	4:0
10	unused	111011	rs1	0	unused (zero)	rs2
10	unused	111011	rs1	1	$siconst_{13}$	

• Assembler Directives

.align	*n*
.ascii	"*string*" [, "*string*"]*
.asciz	"*string*" [, "*string*"]*
.byte	*8-bitval* [, *8-bitval*]*
.data	
.global	*label* [, *label*]*
.half	*16-bitval* [, *16-bitval*]*
.include	"*filename*"
.skip	*n*
.text	
.word	*32-bitval* [, *32-bitval*]*

• ISEM Traps

0 terminate program
1 putchar(%r8)
2 blocking getchar(%r8)
3 nonblocking getchar(%r8)
4 print integer (%r8) as hexadecimal
6 print zero-byte-terminated string pointed to by %r8

• Synthetic Instructions

Synthetic instruction		Implementation	
bclr	*rs, rd*	andn	*rd, rs, rd*
bclr	*rs,* $siconst_{13}$	andn	*rs,* $siconst_{13}$*, rd*
bset	*rs, rd*	or	*rd, rs, rd*
bset	$siconst_{13}$*, rd*	or	*rd,* $siconst_{13}$*, rd*
btst	rs_1, rs_2	andcc	rs_1, rs_2, %g0
btst	*rs,* $siconst_{13}$	andcc	*rs,* $siconst_{13}$, %g0
btog	*rs, rd*	xor	*rd, rs, rd*
btog	*rs,* $siconst_{13}$	xor	*rs,* $siconst_{13}$*, rd*
call	*address*	jmpl	*address*, %g0
clr	*rd*	or	%g0, %g0, *rd*
clrb	[*address*]	stb	%g0, [*address*]
clrh	[*address*]	sth	%g0, [*address*]
clr	[*address*]	st	%g0, [*address*]
cmp	rs_1, rs_2	subcc	rs_1, rs_2, %g0
cmp	*rs,* $siconst_{13}$	subcc	*rs,* $siconst_{13}$, %g0
dec	*rd*	sub	*rd*, 1, *rd*
dec	$siconst_{13}$*, rd*	sub	*rd,* $siconst_{13}$*, rd*
deccc	*rd*	subcc	*rd*, 1, *rd*
deccc	$siconst_{13}$*, rd*	subcc	*rd,* $siconst_{13}$*, rd*
inc	*rd*	add	*rd*, 1, *rd*
inc	$siconst_{13}$*, rd*	add	*rd,* $siconst_{13}$*, rd*
inccc	*rd*	addcc	*rd*, 1, *rd*
inccc	$siconst_{13}$*, rd*	addcc	*rd,* $siconst_{13}$*, rd*
jmp	*address*	jmpl	*address*, %g0
mov	*rs, rd*	or	%g0, *rs, rd*
mov	$siconst_{13}$*, rd*	or	%g0, $siconst_{13}$*, rd*
mov	*statereg, rd*	rd	*statereg, rd*
mov	*rs, statereg*	wr	%g0, *rs, statereg*
mov	$siconst_{13}$*, statereg*	wr	%g0, $siconst_{13}$*, statereg*
neg	*rs, rd*	sub	%g0, *rs, rd*
neg	*rd*	sub	%g0, *rd, rd*
not	*rd*	xnor	*rd*, %g0, *rd*
not	*rs, rd*	xnor	*rs*, %g0, *rd*
restore		restore	%g0, %g0, %g0
save		save	%g0, %g0, %g0
ret		jmpl	%i7+8, %g0
retl		jmpl	%o7+8, %g0
set	*iconst, rd*	or	%g0, *iconst, rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
		or	*rd*, %lo(*iconst*), *rd*
tst	*rs*	orcc	%g0, *rs*, %g0