Instruction Set Architecture and the SPARC: References

- `iwaki.anu.edu.au`: an UltraSPARC IIICu system
  - system configuration info from the commands `prtconf` and `fpversion`
- Structured Computer Organization, Andrew S. Tanenbaum, Edition 3
- SPARC Architecture, Assembly Language Programming and C, Richard Paul
- The SPARC Architecture Manual, Version 9

(credits: Dr Alistair Rendell, COMP2300, 2005)

Instruction Set Architectures

- early microprocessors were very simple, but in 1964 IBM introduced the 360 series which was microprogrammed
- from then, instruction sets and addressing modes (PeANUt Manual 4.2.1) increased, prompted in part by development of high level languages
- special microcode was added to handle case statements, procedure calling, array indexing etc.
  - led to the CISC concept (Complex Instruction Set Computer)
- in the 70s writing, debugging and maintaining microcode became a major issue
- academics begin to analyse what programs actually did and this resulted in a major rethink of microprocessor design
  - led to the RISC concept (Reduced Instruction Set Computer)

Typical Program Instructions (as of late 1970s)

statement:	SAL	XPL	Fortran	C	Pascal	Average
assignment	47	55	51	38	45	47
if	17	17	10	43	29	23
call	25	17	5	12	15	15
loop	6	5	9	3	5	6
goto	0	1	9	3	0	3
other	5	5	16	1	6	7

- SAL was PASCAL like language used in OS code
- XPL was a PL/1 like language used for system programming
- **conclusion:** most programs consist of assignments, if statements and procedure calls

Typical Assignments and Procedures (as of late 1970s)

assignment

# terms	
0	-
1	80
2	15
3	3
4	2
$\geq$0	80

procedures

# locals	
0	22
1	17
2	20
3	14
4	8
$\geq$ 5	20

# parameters	
0	41
1	19
2	15
3	9
4	7
$\geq$5	8

- 80% of all assignment are `variable = value`
- 21% of all procedures have no local variables
- 41% of all procedures have no parameters

conclusion: although people can write complex code - most is very simple!

Comparison of CISC and Early RISC Machines

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC1	Stanford MIPS
year completed	1973	1978	1978	1980	1981	1983
instructions	208	303	270	120	39	55
microcode size (bytes)	54K	61K	17K	0	0	0
instruction size (bytes)	2-6	2-57	1-3	4	4	4
execution model	reg-reg reg-mem mem-mem	reg-reg reg-mem mem-mem	stack	reg-reg	reg-reg	reg-reg

Characteristics of RISC and CISC Machines

	RISC	CISC
1	fixed format instructions	variable format instructions
2	only loads/stores reference memory	any instruction may reference memory
3	few instructions and modes	many instructions and modes
4	instructions executed by the hardware	instructions interpreted by the microcode
5	simple instructions taking 1 cycle	complex instructions taking multiple cycles
6	highly pipelined	not pipelined or less pipelined
7	complexity is in the compiler	complexity is in the microprogram
8	multiple register sets	single register set

RISC is now the dominant processor architecture

(Pentium implements CISC using RISC!)

RISC Processors

First generation characteristics:

- uniform instruction length
- load/store architecture (simple addressing)
- pipelining
- branching (delayed branching and branch prediction)

Second generation:

- faster clocks
- super-pipelining
- superscalar

Post-RISC

- out-of-order execution

Uniform Instruction Length

- fundamental RISC feature (and also true for PeANUt - 16 bit)
- CISC machine had no apriori knowledge of length of each instruction

original SPARC instruction formats (32-bit)

2	5	6	5	1	8	5	
	DEST	OPCODE	SRC1	0	FP-OP	SRC2	3 register
	DEST	OPCODE	SRC1	1	IMMEDIATE CONSTANT		Immediate

2	5	3	22	
	DEST	OP	IMMEDIATE CONSTANT	SETHI

2	1	4	3	22	
	A	COND	OP	PC-RELATIVE DISPLACEMENT	BRANCH

2	30	
	PC-RELATIVE DISPLACEMENT	CALL

Load/Store Architecture

- memory reference restricted to load/store
  - NOT true for PeANUt, with direct, indirect, indexed and stack (i.e. `label`, `@label`, `*label`, `!label`)
  - BUT we have many more registers for variables (not just AC)
- operations only occur between data that is in registers

```
                          ! e.g. a = a + b
ld   [%o1], %i1           ! %i1 = memory[%o1]
ld   [%o2], %i2           ! %i2 = memory[%o2]
add %i1,%i2,%i3           ! %i3 = %i1 + %i2
st   %i3, [%o1]           ! memory[%o1] = %i3
```

- motivation:
  - to enable fixed instruction length
  - to ease pipelining (see below)
  - to reduce the number of (slow) memory references

The PeANUt Execution Algorithm

```
REPEAT
    CI := memory[PC];
    PC := PC + 1;
    Evaluate_Operands;
    Execute_Operation;
UNTIL FALSE
```

- one operation completes before the next starts
- RISC wanted fast cycle times, but cycle time reflects complexity of the operation
- RISC issues one operation per cycle, but does NOT require completion of that operation in one cycle
- RISC can achieve one result per cycle via pipelining

Pipelining

- break instr'n execution into k stages; $\Rightarrow$ can get $\leq k$-way ||ism

  (generally, the circuitry for each stage is independent)
- e.g. ($k = 5$): stages `FI` = Fetch Instr'n., `DI` = Decode Instr'n., `FO` = Fetch Operand, `EX` = Execute Instr'n., `WB` = Write Back

(branch):	FI	DI	FO	EX	WB					
(delay slot:)		FI	DI	FO	EX	WB				
(guess)			FI	DI	FO	EX	WB			
(guess)				FI	DI	FO	EX	WB		
(sure)					FI	DI	FO	EX	WB	

  - note: `EX` & `WB` stages may involve memory accesses (and may possibly stall the pipeline)

Branch Delay and Prediction

- branch to a new program address will disrupt the pipeline flow
  - processor doesn't know if instruction is a branch until the decode stage
  - may not know if branch taken until execute stage
  - if taken, "in flight" instructions must be annulled
- SPARC has a 'branch delay slot' instr'n immediately after branch instructions
  - e.g.

```
cmp %1, %2          !
add %3,1,%3         ! n = n + 1
bne endif1          ! if (i != k) ...
 nop                ! branch delay slot; ALWAYS executed
```

    (could move the `add` into the branch delay slot)
  - pipeline can continue for **unconditional branches** (e.g. function calls)
- conditional branches are more difficult, pipeline will stall and require flushing as it can't be recognized before the `DI` stage
  - hardware for branch prediction is very important

Superscalar (2nd Generation Feature)

a small number (w) of instr'ns (a group) are scheduled by the H/W to *execute together*

- groups must have an appropriate 'instruction mix'

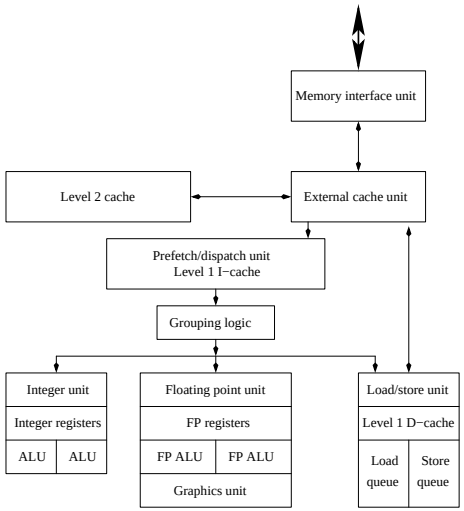
  e.g. UltraSPARC ($w = 4$): $\left\{ \begin{array}{l} \leq 2 \text{ different floating point} \\ \leq 1 \text{ load / store} \;;\; \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right\}$ instr'ns per group

- have $\leq w$-way ||ism over *different* types of instr'ns
- generally requires:
  - multiple ($\geq w$) instr'n fetches
  - extra grouping stage (G) in the pipeline
- problem: amplifies all problems with pipelining
- issue: the instruction mix must be balanced for maximum performance!

The SPARC Family

- 1982: SUN formed by McNealy, Joy, Khosla and Bechtolsheim
- 1986: SPARC (Scalable Processor ARChitecture) v7 ISA published
- 1987: first SPARC systems sold (Sun-4)
- 1988: SPARC industry consortium formed
- 1990: SPARC v8 ISA published
- 1994: SPARC v9 ISA published
- 95-00: UltraSPARC I (95), II (98), III (00), IIICu (03), IV (05)
- 2006: UltraSPARC T1: 8 CPUs per chip, 4 threads can execute on each CPU

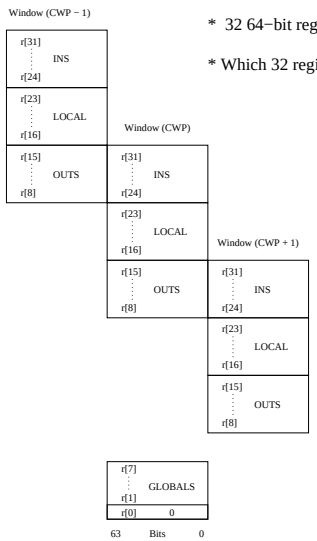
UltraSPARC (IIICu) Architecture



memory hierarchy:

- level 1 D-cache: 64 KB, 4-way S.A., 32-byte blocks
- level 2 E-cache: 512 to 8192 KB, 2-way S.A., 64-byte blocks
- D-TLB: 128-entry, 2-way S.A., 8KB pages

Registers and Register Windows



* 32 64−bit registers are visible at any time

* Which 32 registers is determined via a window register pointer

Function Calls

- historically, registers of caller and parameters are stored on the stack prior to function invocation
  - e.g. in PeANUt, may want to store XR to stack before function call
- modern computers have special instructions, e.g. SPARC provides the `save` instruction:

```
save %sp,-120,%sp
```

  - provides a new register window, and decrements stack pointer by 120
  - this reserves space to save all registers in current window, if needed
- a limited number of parameters are passed via the 8 shared `ins(CWP)` / `outs(CWP-1)` registers
  - recall most procedures have few arguments (≤ 6)
  - stack & frame pointers are two of these registers

SPARC Assembler #1

- examine `sum.s` produced via `gcc -S sum.c`

```
int sum(int n1, int n2)
{
   int tmp;

   tmp = n1+n2;
   return tmp;
}
```

```
.section        ".text"
        .align 4
        .global sum
        .type    sum,#function
sum:
        !#PROLOGUE# 0
        save %sp,-120,%sp
        !#PROLOGUE# 1
        st %i0,[%fp+68]
        st %i1,[%fp+72]
        ld [%fp+68],%o0
        ld [%fp+72],%o1
        add %o0,%o1,%o0
        st %o0,[%fp-20]
        ld [%fp-20],%o0
        mov %o0,%i0
        b .LL1
         nop
.LL1:
        ret
         restore
.LLfe1:
        .size    sum,.LLfe1-sum
        .ident  "GCC: (GNU) 2.8.1"
```


SPARC Assembler #2

- examine `sum.s` produced via `gcc -S -O sum.c` (and for `repeat.c`)

```
int sum(int n1, int n2)
   int tmp;
   tmp = n1+n2;
   return tmp;
}

int repeat(int n)
{

   int tmp,i;
   tmp = 0;

   for (i=0; i<n; i++)
        tmp = tmp + i;

   return tmp;
}
```

```
        ...
sum:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        retl
         add %o0,%o1,%o0
        ...
repeat:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        mov %o0,%g3      !%g3=n
        mov 0,%o0        !%o0=tmp
         bge .LL4
        mov %o0,%g2      !%g2=i
        add %o0,%g2,%o0
.LL8:
        add %g2,1,%g2
        cmp %g2,%g3
        bl,a .LL8
         add %o0,%g2,%o0
.LL4:
        retl
```