CPE300: Digital System Architecture and Design

Fall 2011
MW 17:30-18:45 CBC C316

RISC: The SPARC
09282011

http://www.egr.unlv.edu/~b1morris/cpe300/

Outline

- Recap
- Finish Motorola MC68000
- The SPARC Architecture

RISC vs. CISC Designs

- CISC: Complex Instruction Set Computer
  - Many complex instructions and addressing modes
  - Some instructions take many steps to execute
  - Not always easy to find best instruction for a task
- RISC: Reduced Instruction Set Computer
  - Few, simple instructions, addressing modes
  - Usually one word per instruction
  - May take several instructions to accomplish what CISC can do in one
  - Complex address calculations may take several instructions
  - Usually has load-store, general register ISA

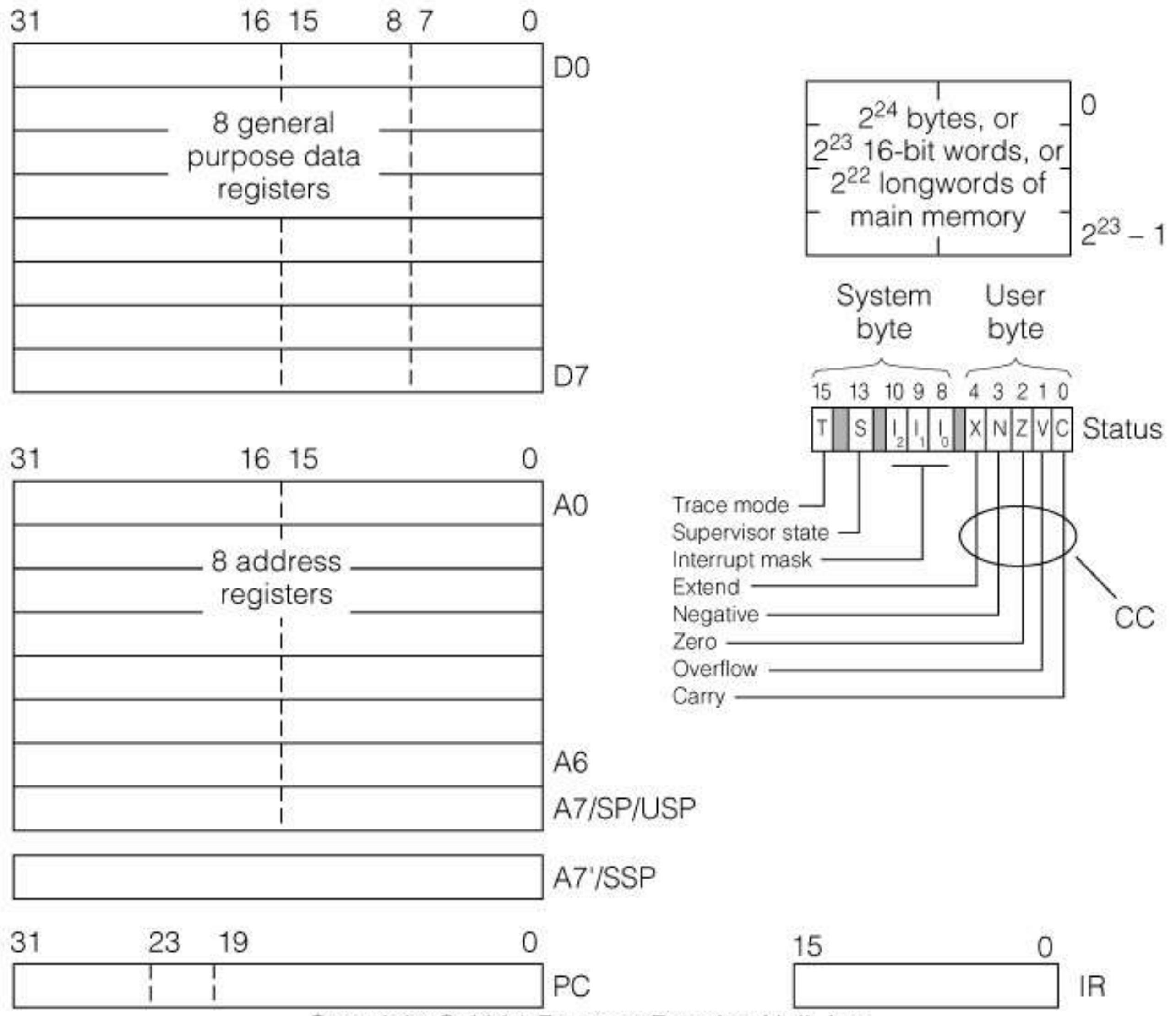
Developing and ISA (Table 3.1)

- Memories: structure of data storage in the computer
  - Processor-state registers
  - Main memory organization
- Formats and interpretation: meaning of register fields
  - Data types
  - Instruction format
  - Instruction address interpretation
- Instruction interpretation: things done for all instructions
  - Fetch-execute cycle
  - Exception handing
- Instruction execution: behavior of individual instructions
  - Grouping of instructions into classes
  - Actions performed by individual instructions

New Concepts from MC68000

- Variable length instructions
  - Large instruction set
- Operation on many different types
  - Must specify byte, word, longword
- Effective address (EA) calculation
  - 14 Addressing modes
- Subroutines
  - E.g. function calls
- Exceptions
  - Interruption of normal sequential instruction execution
- Memory-mapped I/O
  - Part of CPU memory reserved for I/O

MC68000 Programmer’s Model



Features of Processor State

- Distinction between 32-bit data registers and 32-bit address registers
- 16 bit instruction register
  - Variable length instructions handled 16 bits at a time
- Stack pointer registers
  - User stack pointer is one of the address registers
  - System stack pointer is a separate single register
    - Why a separate system stack?
- Condition code register: System & User bytes
  - Arithmetic status (N, Z, V, C, X) is in user status byte
  - System status has Supervisor & Trace mode flags and the Interrupt Mask

Main Memory

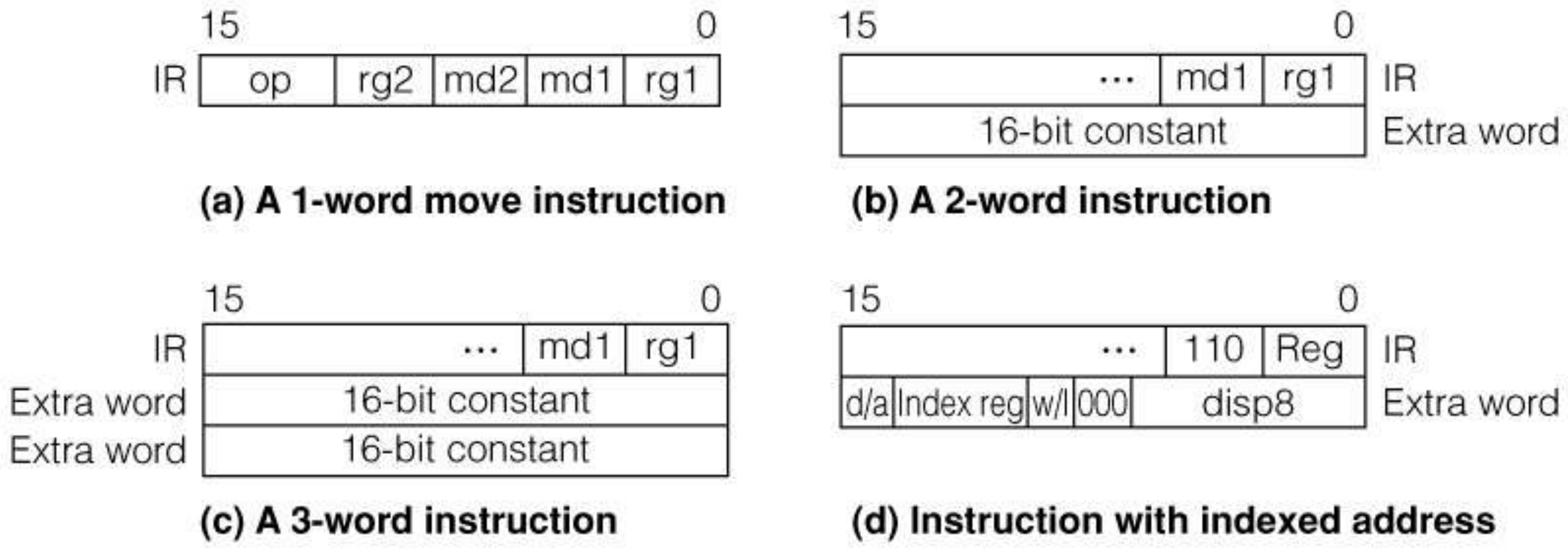
- Main memory:

`Mb[0..`2^{24}`-1]<7..0>:`	memory as bytes
`Mw[ad]<15..0> := Mb[ad]#Mb[ad+1]:`	memory as words
`Ml[ad]<31..0> := Mw[ad]#Mw[ad+2]:`	memory as longwords

- Word and longword forms are big-endian
  - Lowest numbered byte contains most significant bit of word
- Hard word alignment constraints
  - Not described in the RTN
  - Word addresses must in end in on binary 0
  - Longword addresses end in two binary 0
    - What are differences between soft alignment?

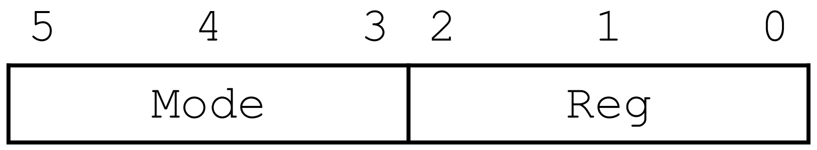
Instruction Formats

- Instructions accessed in 16-bit words
- Variable number of words in an instruction



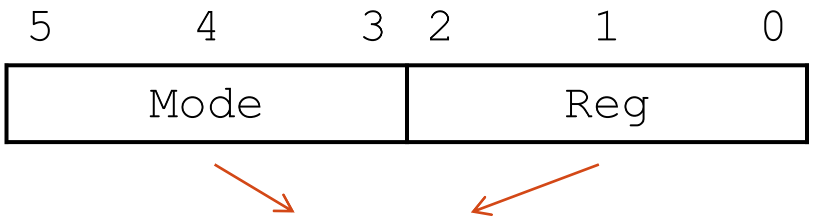
Addressing Modes

- General address of operand specified by 6-bit field
  - Access paths to memory and registers
  - See Table 3.2 for details
- 6-bit effective address



  - Mode field provides access paths to operands
- Not all operands/results can be specified by general address - some must be in registers
- Exception
  - Destination of MOVE instruction has mode and reg fields reversed

Table 3.2: MC68000 Addressing Modes

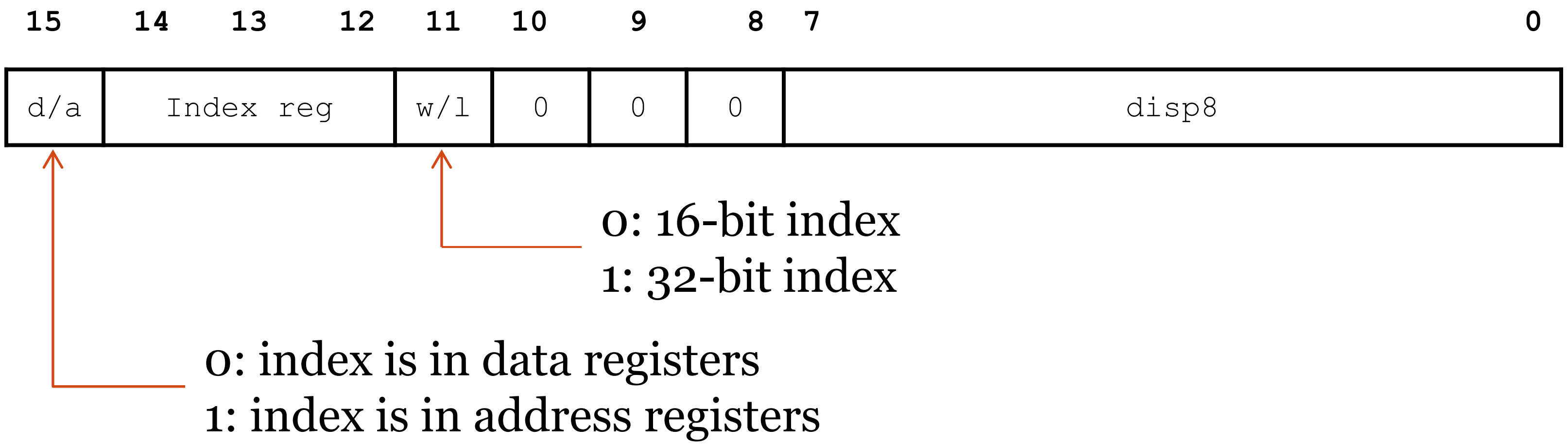


Name	Mode	Reg	Assembler Syntax	Extra Words	Description
Data register direct	0	0-7	`Dn`	0	`Dn`
Address register direct	1	0-7	`An`	0	`An`
Address register indirect	2	0-7	`An)`	0	`M[An]`
Autoincrement	3	0-7	`(An)+`	0	`M[An];An`$\leftarrow$`An+d`
Autodecrement	4	0-7	`-(An)`	0	`An`$\leftarrow$`An-d;M[An]`
Based	5	0-7	`disp16(An)`	1	`M[An+disp16]`
Based indexed short	6	0-7	`disp8(An,XnLo)`	1	`M[An+XnLo+disp8]`
Based indexec long	6	0-7	`disp8(An,Xn)`	1	`M[An+Xn+disp8]`
Absolute short	7	0	`Addr16`	1	`M[addr16]`
Absolute long	7	1	`Addr32`	2	`M[addr32]`
Relative	7	2	`disp16(PC)`	1	`M[PC+disp16]`
Relative indexed short	7	3	`disp8(PC,XnLo)`	1	`M[PC+XnLo+disp8]`
Relative indexed long	7	3	`disp8(PC,Xn)`	1	`M[PC+Xn+disp8]`
Immediate	7	4	`#data`	1-2	No location, data

RTN Formatting for EA Calculation

`XR[0..15]<31..0> := D[0..7]<31..0>#A[0..7]<31..0>:`	Index register can be D or A
`xr<3..0> := Mw[PC]<15..12>:`	Index number for index mode
`wl := Mw[PC]<11>:`	Short /long index flag
`dsp8<7..0> := Mw[PC]<7..0>:`	Displacement for index mode
`index := ((wl=0) → XR[xr]<15..0>:`	Short
`(wl=1) → XR[xr]<31..0>):`	Long index value

- 4-bit field specifies index register
- Either 16 or 32 bit s of index register may be used
- Low order 8-bits are used as offset



Addressing Mode Highlights

- Modes 0-6 use a register to calculate a memory address
  - Based modes (5-6) require an extra word (16-bits) to specify address
- Mode 7 does not use a register
  - Functionality is expanded by repurposing `reg` field
  - All variants require extra words to complete the instruction and specify the memory address

MC68000 Instruction Types

- Instruction fields not standardized
  - Maximize instructions in limited word size (bits)
  - Operates on different types (`B, W, L`)
- Data movement instructions
  - CC can be set during move
- ALU instructions
  - 1 EA, 1 Dn operand
  - Destination specified by 3-bit mode field
- Program control instructions
  - Use 16 condition codes
  - Has subroutine specific instructions

Starting a Program

- Assembler
  - Convert assembly language text to (binary) machine language
    - Addresses translated using a symbol table
    - Addresses adjusted to allow room for blocks of reserved memory (e.g. an array definition)
- Linker
  - Separately assembled modules combined and absolute addresses assigned
- Loader
  - Move binary words into memory
- Run time
  - PC set to started address of loaded module.
  - OS usually makes a jump or procedure call to the address

Pseudo Operations

- Operation performed by assembler at assembly time not by CPU at run time
- `EQU` - defines constant symbol
  - `PI:      EQU             3.14`
  - Substitution made at assemble time
- `DS.(B, W, L)` – defines block of storage
  - A label is associated with first word of block
  - `Line:    DS.B            132`
  - Program loader (part of OS) accomplishes this
- # indicates value of symbol rather than location addressed by symbol
  - `MOVE.L   #1000, D0` ; moves 1000 to D0
  - `MOVE.L   1000, D0` ; moves value addr. 1000 to D0
  - Assembler detects difference
- `ORG` – defines memory address where following code will be stored
  - `Start:   ORG    $4000` ; next instruction/data at addr. 0x4000
- Character constants in single quotes
  - `'X'`

Example: Clearing Block of Memory

```
MAIN      …
          MOVE.L      #ARRAY, A0      ;Base of array
          MOVE.W      #COUNT, D0      ;Number of words to clear
          JSR         CLEARW          ;Make the call
          …
CLEARW    BRA         LOOPE           ;Branch for init. Decr.
LOOPS     CLR.W       (A0)+           ;Autoincrement by 2    .
LOOPE     DBF         D0, LOOPS       ;Dec.D0,fall through if -1
          RTS                         ;Finished
```

- Subroutine expects block base in `A0`, count in `D0`
- Linkage uses stack pointer
  - `A7` cannot be used for anything else

Exceptions

- Changes sequential instruction execution
  - Next instruction fetch not from PC location
  - Exception vector
    - Address supplying the next instruction
- Arise from instruction execution, hardware faults, external conditions
  - Interrupts – externally generated exceptions
  - ALU overflow, power failure, completion of I/O operation, out of range memory access, etc.
- Trace bit = 1 causes exception after every instruction
  - Used for debugging

Exception Handling Steps

1. Status change
   - Temporary copy of status register made
   - Supervisor mode bit S is set and trace bit T is reset
2. Exception vector address obtained
   - Small address made by shift 8-bit vector number left 2
   - Contents of longword at vector address is new address of next instruction
   - Exception handler or interrupt service routine starts at this address
3. Old PC and Status register are pushed onto supervisor stack, `A7′= SSP`
4. PC loaded from exception vector address
5. Return from handler is done by `RTE`
   - Works like `RTR` except Status register is restored rather than CCs

Exception Priority

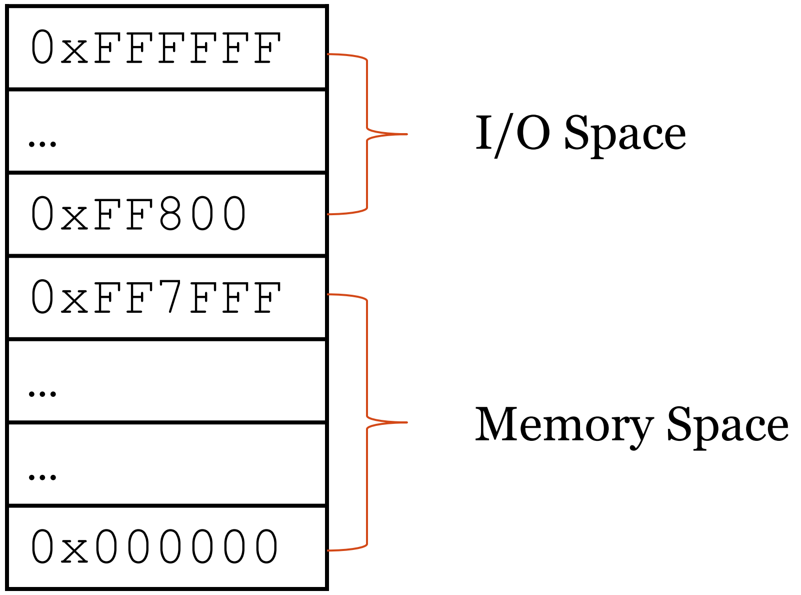
- Method to determine which exception vector to use when multiple exceptions occur at once
- 7 levels of priority in MC68000
  - Status Register contains current priority
- Exceptions with priority ≤ current priority are ignored

- Exceptions are sensed before fetching next instruction

Memory-Mapped I/O

- Part of CPU memory is devoted/reserved for I/O
  - No separate I/O space
  - Not popular for machines having limited address bits
- Single bus needed for memory and I/O
  - Less packaging pins
- Size of I/O and memory spaces independent
  - Many or few I/O devices may be installed
  - Much or little memory may be installed
- Spaces are separated by putting I/O at the top end of address space

24-bit address space with top 32K reserved for I/O



Notice top 32K can be addressed by a negative 16-bit value

Motorola MC68000 Highlights

- CISC – has many addressing modes and instruction formats
  - Pack as much functionality as possible into small word size
- 16-bit instruction load
  - Some instructions multiple words
- Interrupts and traps (a real machine)
- Memory mapped I/O

The SPARC Microprocessor

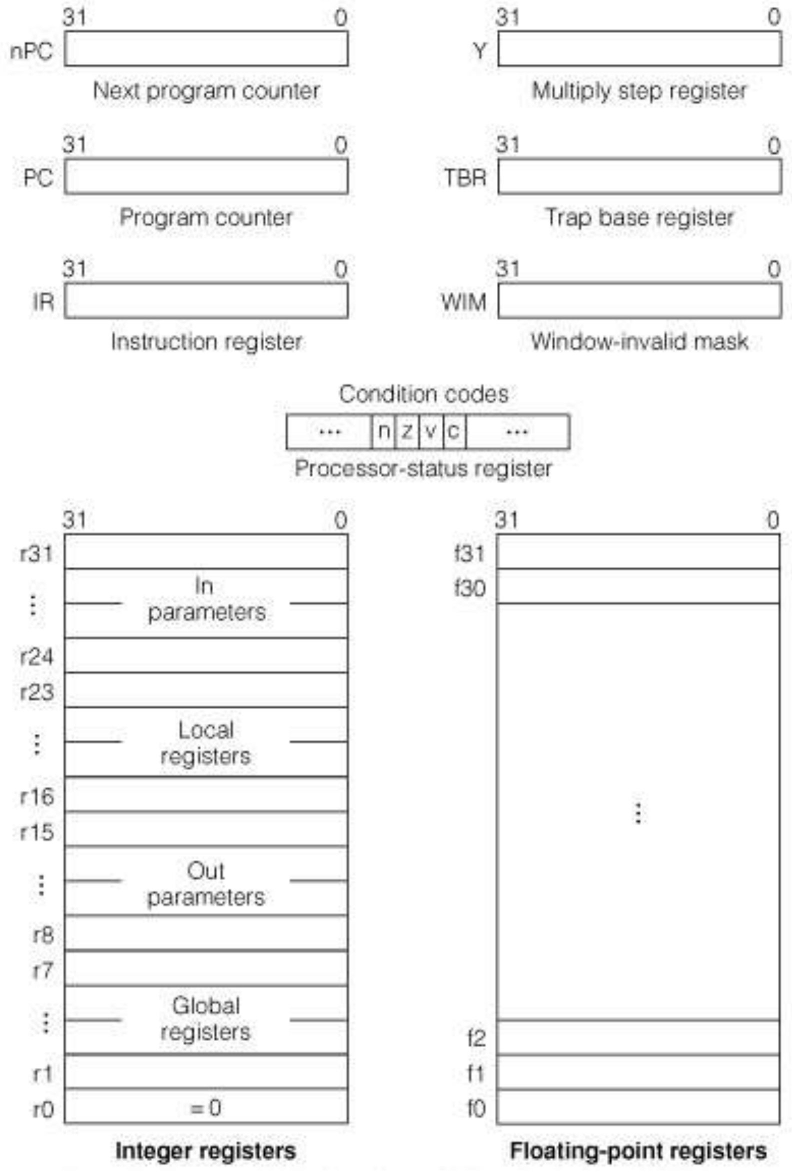
- Scalable Processor Architecture (SPARC)
  - RISC microprocessor architecture
  - Not a machine – specification for implementation
- General register, load/store architecture
- Only 2 addressing modes
  - Reg + Reg
  - Reg + 13-bit constant
- Only 69 basic instructions
  - 32-bit instruction length
  - Separate floating point handling
    - 3 processing units – integer unit, FP unit, coprocesser
- 4 stage pipeline in initial implementation
- Contains features not inherently RISC
  - Register windows
    - Separate, overlapping register sets for subroutines
  - Big-endian memory organization

Developing and ISA (Table 3.1)

- Memories: structure of data storage in the computer
    - Processor-state registers
    - Main memory organization
- Formats and interpretation: meaning of register fields
    - Data types
    - Instruction format
    - Instruction address interpretation
- Instruction interpretation: things done for all instructions
    - Fetch-execute cycle
    - Exception handing
- Instruction execution: behavior of individual instructions
    - Grouping of instructions into classes
    - Actions performed by individual instructions

SPARC Processor State

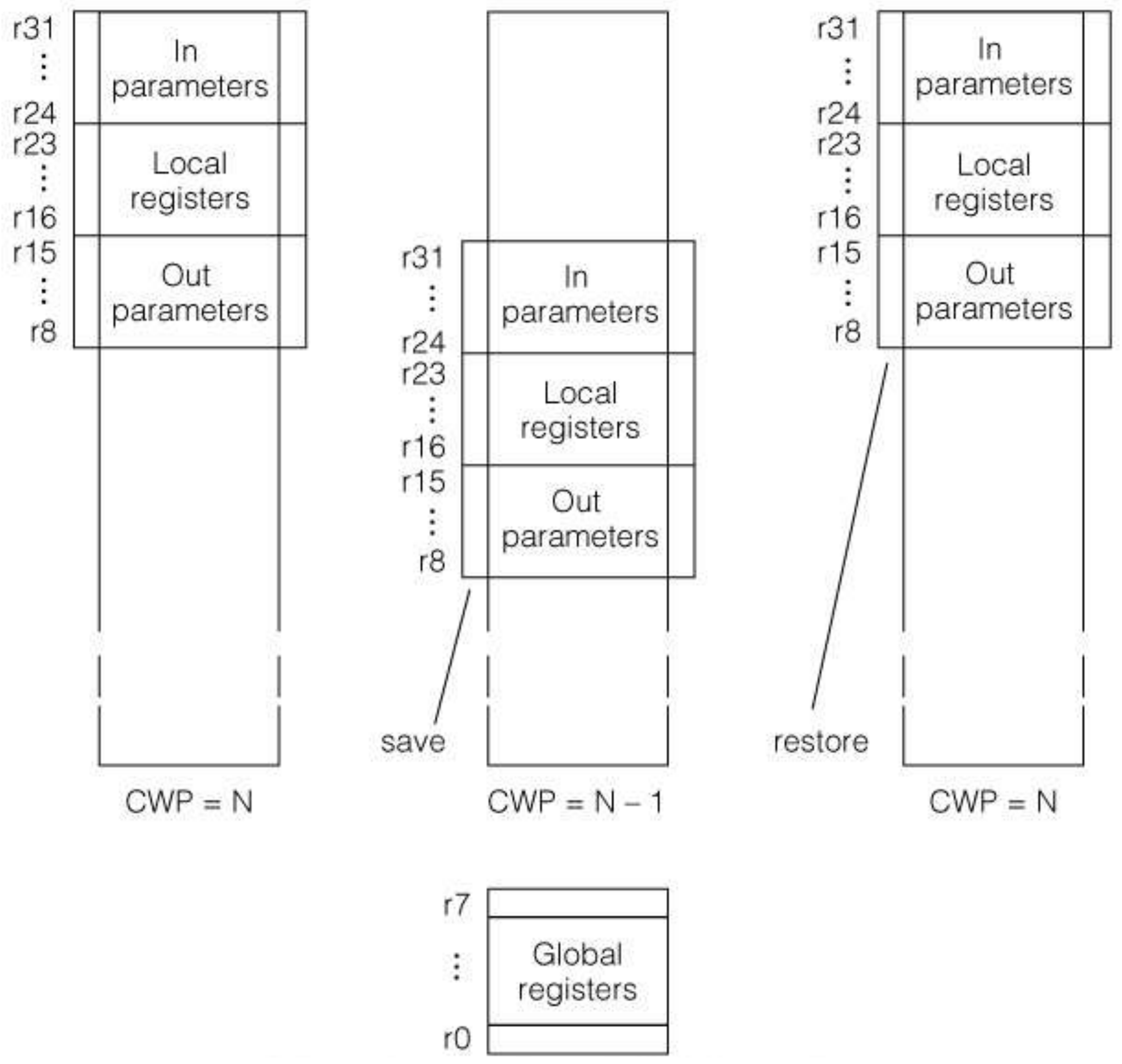
- 32-bit general registers
  - Integer and floating point separate
- Brach delays
  - Requires 2 program counters
- Processor-status register (PSR)
  - Condition codes
- Window-invalid mask (WIM)
  - Used for register windows
- Trap base register
  - Traps and interrupts



Register Windows

- High percentage of memory traffic for saving and restoring registers during procedure calls
  - More registers = less memory traffic
  - Reduce overhead of calls
- Only a small subset of registers is visible to the programmer at a given time (within procedure)
  - Dedicated but overlapping registers groups
    - Global
    - Input parameters
    - Output parameters
    - Local registers
  - Overlap designed to prevent swapping of registers
    - Output parameters in one window become input parameters in the next

Register Windows Mechanism



Register Window Format

- 32 general addresses accessible at a time (integer and address) from set of 120
  - Global registers = `g0..g7` are not part of a window and always available
    - `g0 = 0`, writes ignored, read returns zero
  - 24 in a movable window from within 120
- During subroutine call
  - `r24..r31` before call become `r8..r15` after
  - `r[8..15]` are for incoming parameters
  - `r[24-31]` for outgoing parameters
  - `r15` used for return address, available in `r31` after a `save`
- Current Window Pointer (CWP) locates available registers within larger register space
- Overflow of register space causes a trap

Window Specifics

- CWP points to register currently called `r8`
  - `save` moves CWP to former r24
  - `restore` reverses process
- Parameters placed in `r24..r31` by caller are available in `r8..r15` by callee
- Spill := attempt to `save` when all windows have been used
  - `save` traps to routine to store registers to memory
  - Window wraps around like a circular buffer
    - On overflow, first window is reused

Main Memory

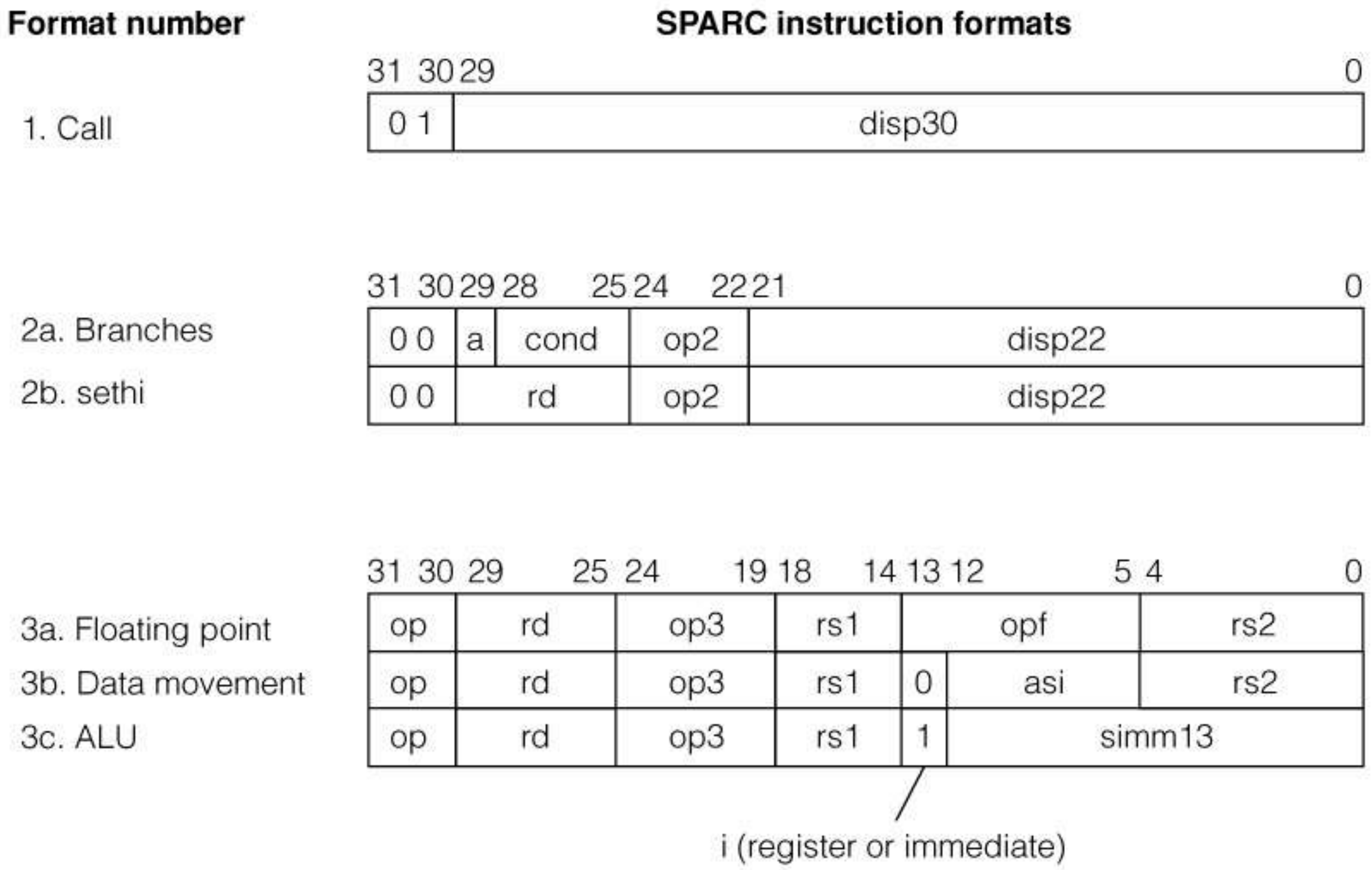
- Main memory:
  - `Mb[0..2`^{32}`-1]<7..0>:` memory as bytes
  - `Mh[ad]<15..0> := Mb[ad]#Mb[ad+1]:` memory as halfwords
  - `Mw[ad]<31..0> := Mh[ad]#Mh[ad+2]:` memory as words
- Word and halfword forms are big-endian
  - Lowest numbered byte contains most significant bit of word
- Hard word alignment constraints
  - Not described in the RTN
  - Word addresses must in end in binary 00
- Memory mapping unit (MMU) defined
  - Allows multiple address spaces (Chapter 7)
  - Will not discuss

Datatypes

- Integer instructions access many different types
  - Bytes
  - 16-bit halfwords
  - 32-bit words
  - 64-bit double words
- 32, 64, 128-bit floating point word sizes (more bits needed for FP Chapter 6)

Instruction Formats

- Three basic formats with some small variations
  - Support 55 basic integer and 15 FP instructions



Addressing Modes

- Only 2 modes for load/store
  - Sum of two registers
  - Sum of register and sign extended 13-bit constant
- Allows for a variety of addressing modes can be synthesized
  - Indexed
    - Base in one register, index in another
  - Register indirect
    - `g0 + rn            ; r0 = 0`
  - Displacement
    - `rn + const.        ; n≠0`
  - Absolute
    - `g0 + const.`
    - Can only reach the bottom or top 4K bytes of memory

Addressing Modes RTN

- Immediate operand indicator
  - `i:= IR<13>`
- Address for load, store, and jump
- `adr<31..0> := (i=0 → r[rs1] + r[rs2]:`
- `                  i=1 → r[rs1] + simm13<12..0> {sign ext.}):`
  - 4K offset for immediate
- Call relative address
- `calladr<31..0> := PC<31..0> + disp30<29..0>#002:`
  - Reaches full 32-bit address space
- Branch Address
- `bradr<31..0> := PC<31..0> + disp22<21..0>#002{sign ext.}:`
  - PC $\pm$8M addresses

RTN for Instruction Interpretation

-
  ```
  Instruction_interpretation := (
        IR ← Mw[PC] ; instruction_execution;
        update_PC_and_nPC; instruction_interpretation):
  ```

- Notice execution occurs before PC updates
    - 2 PC values to update because of delayed branch
- Interrupts not mentioned in this simple RTN statement

Data Movement instructions

- Typically specified by both an `op` field and secondary `op3` field
- Supports byte, halfword, word, doubleword moves
  - Loaded into lsb of register
  - Top bits cleared or sign-extended based on unsigned or signed load
  - Even register is needed for destination of `ldd`
- Register moves possible with `or` with `g0`
- Loading a register with 32-bit constant takes 2 instructions
  - `SETHI  #upper22, R17`
  - `OR     R17, #lower10, R17`

ALU Instructions

- Instr. Format 3, `op=10` (binary)
- CCs set based on S flag bit
  - Bit 5 of `op3` field
- Arithmetic
  - Multiply and divide either as FP or in software
  - Multi-step instructions placed in FPU
- Logical
  - And, or, xor, and shifts defined
  - Not synthesized using `orn` with `g0`

Branch and Control Instructions

- Implement several branch and procedure calls
- Conditional branches
  - 4-bit condition field
  - Branch target address is a word offset relative to PC
    - `bradr<31..0> := PC<31..0> + disp22<21..0>#002{sign ext.}:`
- Delayed branching
  - Instruction after branch executed prior to completion of branch instruction
  - Annul bit `a` allows programmer control of instruction in branch-delay slot
    - When branch is not taken:
      - `a=1` delay instruction annulled
      - `a=0` delay instruction executed

SPARC Assembly

- Format
- `Label: instruction        !comment`
- Destination is always right most operand
- Register (denoted with %) aliasing
  - `%r8 - %r15 = %o0 - %o7`
  - `%r16 - %r23 = %l0 - %l7`
  - `%r24 - %r31 = %i0 - %i7`
- Memory addresses enclosed in square brackets `[]`
  - Branch, jump, call addresses do not use [] but typically a label

SPARC Example Code

- Program to add 2 integers

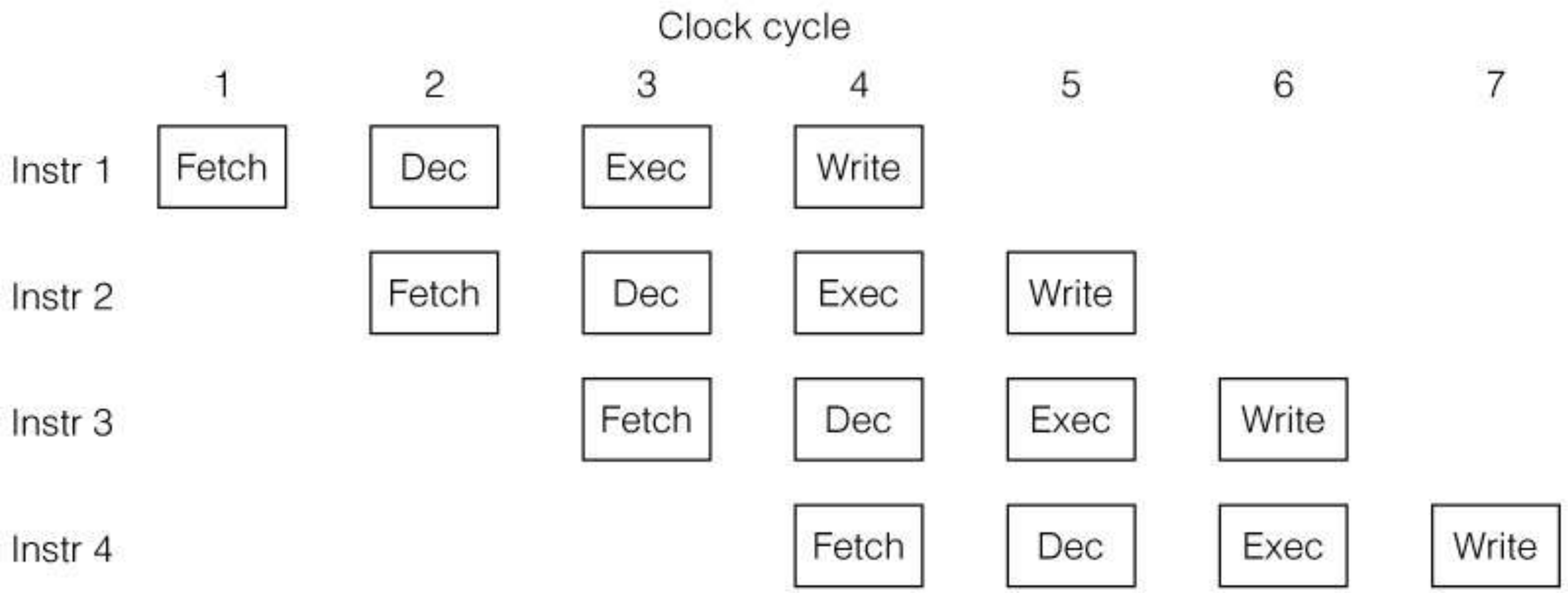
```
            .begin
            .org
progl:      ldw         [x], %r1           ! Load word from M[x] into register %r1
            ldw         [y], %r2           ! Load word from M[x] into register %r2
            addcc       %r1, %r2, %r3      ! %r3 ←%r1 + %r2  ; set CCs
            st          %r3, [z]           ! store sum in M[z]
            jmpl        %r15, +8, %r0      ! Return to caller
            nop                            ! branch delay slot
x:          15                             ! Reserve storage for x, y, and z
y:          9
z:          0
            .end
```

- `r15` contains return address of `progl`
  - Placed there by OS in this case

SPARC Pipelining

- Many aspects of the SPARC design are in support of a pipelined implementation
    - Simple addressing modes, simple instructions, delayed branches, load/store architecture
- Simplest form of pipelining is fetch/execute overlap—fetching next inst. while executing current inst.
- Pipelining breaks inst. processing into steps
    - A step of one instruction overlaps different steps for others
- A new inst. is started (issued) before previously issued instructions are complete
- Instructions guaranteed to complete in order

SPARC MB86900 Pipeline



Clock cycle

	1	2	3	4	5	6	7
Instr 1	Fetch	Dec	Exec	Write			
Instr 2		Fetch	Dec	Exec	Write		
Instr 3			Fetch	Dec	Exec	Write	
Instr 4				Fetch	Dec	Exec	Write

- 4 stage pipeline
- Results written to registers in write stage

Pipeline Hazards

- Branch or jump change the PC as late as Exec. or Write, but next inst. has already been fetched
  - One solution is ‘Delayed Branch’
  - One (maybe 2) instruction following branch is always executed, regardless of whether branch is taken
  - SPARC has a delayed branch with one ‘delay slot”, but also allows the delay slot instruction to be annulled (have no effect on the machine state) if the branch is not taken
- Registers to be written by one instruction may be needed by another already in the pipeline, before the update has happened (Data Hazard)

SPARC Highlights

- RISC machine has fewer simple instructions
  - Multistep arithmetic operations happen in special units
  - Regular instruction formats and few addressing modes simplify instruction decode
- Load/store machine with ALU only on registers
- Use of branch delays for pipelining
  - No load delay
- Use of register windows
  - Extend register space for fewer memory operations

CISC vs. RISC Recap

- CISCs supply powerful instructions tailored to commonly used operations, stack operations, subroutine linkage, etc.
- RISCs require more instructions to do the same job
- CISC instructions take varying lengths of time
- RISC instructions can all be executed in the same, few cycle, pipeline
- RISCs should be able to finish (nearly) one instruction per clock cycle

Chapter 3 Summary

- Machine price/performance are the driving forces.
  - Performance can be measured in many ways: MIPS, execution time, Whetstone, Dhrystone, SPEC benchmarks.
- CISC machines have fewer instructions that do more.
  - Instruction word length may vary widely
  - Addressing modes encourage memory traffic
  - CISC instructions are hard to map onto modern architectures
- RISC machines usually have
  - One word per instruction
  - Load/store memory access
  - Simple instructions and addressing modes
  - Result in allowing higher clock cycles, prefetching, etc