ARM Overview

- Advanced RISC Machines (ARM)
  - Employs RISC-style architecture with some CISC-style features
  - 32-bit registers and addresses
  - Byte-addressable memory with 8-bit bytes, 16-bit halfwords and 32-bit words.
  - Enforces memory alignment for words and half words.
  - Supports both big-endian and little-endian memory storage order

ARM Overview

- RISC features
  - Fixed length 32-bit instructions
  - Load/store architecture
  - All arithmetic and logic instructions use only register operands

ARM Overview

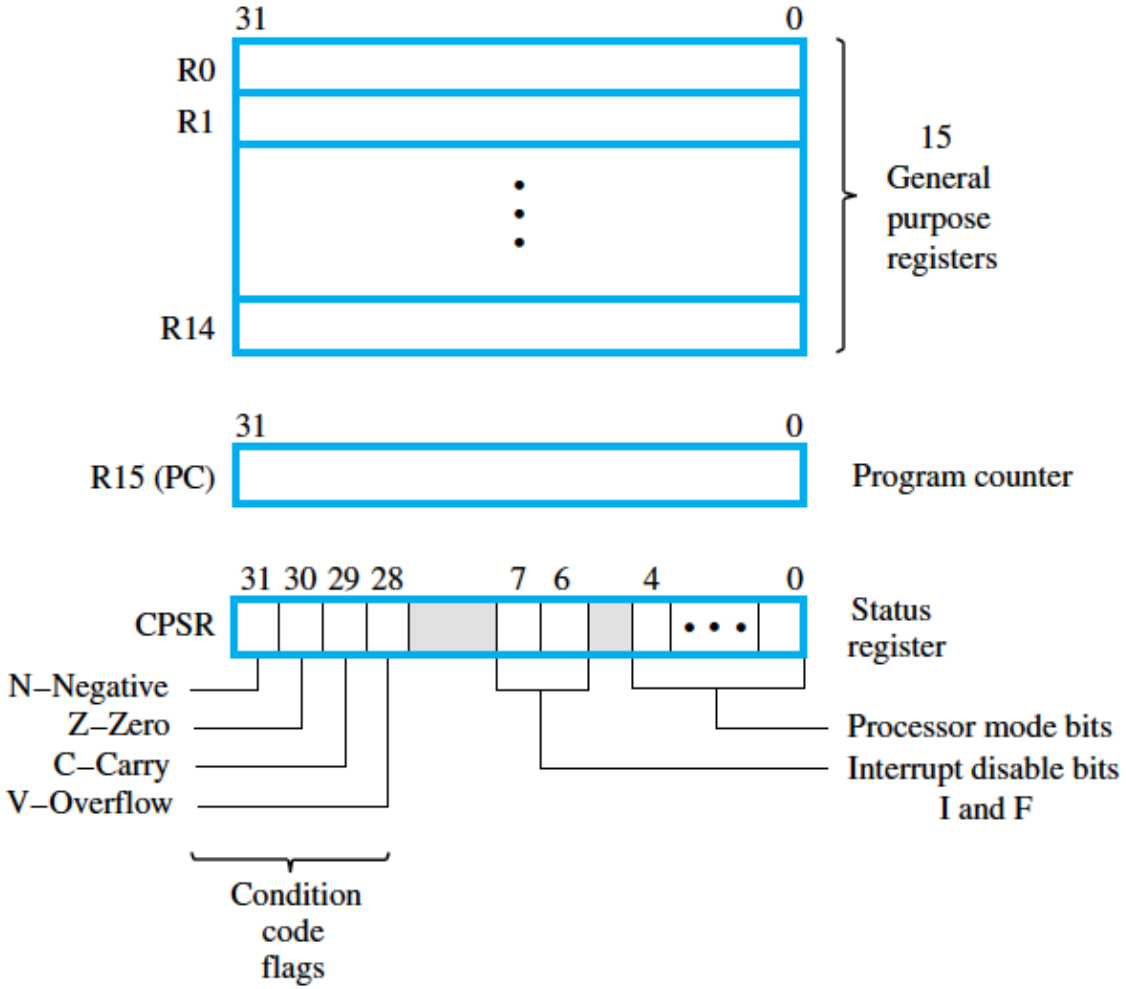
- CISC features
  - Supports autoincrement, autodecrement & PC-relative addressing modes
  - Condition codes (N,Z,V,C) are used for branching & conditional execution
  - Multiple registers can be loaded from or stored into a block of consecutive memory words

ARM Overview

- Unusual features
  - All instructions are conditionally executed
    - conditions specified by 4-bit field within each instruction
    - Can be set to always execute
    - Permits shorter routines than RISC machines using many branch instructions
  - No divide or explicit shift instructions
    - Division must be done in software
    - Immediate or register operands can be shifted by a prescribed amount before being used

ARM Overview

- 16 registers (R0 – R15)



ARM Overview

- R15 used as the program counter (PC)
- R14 is the subroutine linkage register
- R13 is the stack pointer
- "Banked registers" facilitate context switches
  - Additional copies of R0 – R14
  - Reduces the need to save and restore registers.

ARM Overview

- ARM addressing modes
  - Indexed addressing mode
  - Register mode
  - Immediate mode
  - Indirect mode
  - Absolute mode
  - Auto-increment & auto-decrement
  - PC relative mode

ARM Overview

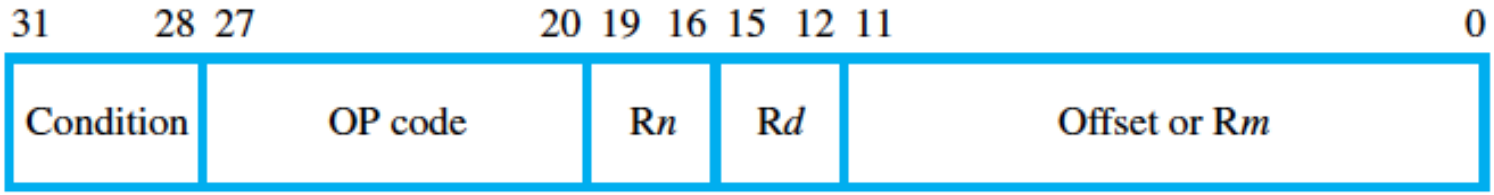
- Other ARM features
  - Employs memory mapped I/O
    - Direct programmed controlled
    - Interrupt driven
  - Supports coprocessors
    - Floating point
    - Signal and video coprocessors
    - Exchange data between CPU and coprocessor registers
    - Memory transfers to/from coprocessor registers

ARM Overview

- Supports THUMB ISA
  - Used in low-cost & low-power embedded systems
    - Mobile devices such as phones
    - 16-bit machine instructions (reduced code space)
      - Fetched and expanded to 32 bits at run time
  - T bit within CPSR indicates when in Thumb mode
    - Programs can contain a mix of Thumb and standard routines.
    - Many Thumb instructions use a two-operand format
    - Conditional execution applies to all standard instructions but only to branches in Thumb set

ARM Addressing Modes

- Basic Indexed Mode
  - Pre-indexed – effective address (EA) = contents of base register plus a signed offset
  - Load (LDR) & store (STR) instructions will be used to illustrate addressing modes



Format for Load and Store instructions.

High 4 bits in all instructions specify a condition that determines whether the instruction is executed.

ARM Addressing Modes

- Pre-indexed mode
  - A bit within the opcode indicates whether the offset is in the low 12 bits or whether the offset is in a register indicated by the low 4 bits of the instruction.
  - Offset is treated as an unsigned value (a bit within the opcode indicates sign for offset).
  - Examples:
    - LDR Rd,[Rn, #offset] ; Rd ← [[Rn] + offset]
    - LDR Rd,[Rn, Rm] ; Rd ← [[Rn]+[Rm]]
    - LDR Rd,[Rn] ; Rd ← [[Rn] + 0]

ARM Addressing Modes

- Relative Addressing Mode
  - The PC is used as the base register.
  - Programmer uses label and assembler determines offset
  - Offset = operand address - PC+8
  - Examples:
    - LDR R1,ITEM ; loads contents of memory location ITEM into R1.

ARM Addressing Modes

- Pre-indexed with writeback
  - Computed EA overwrites Rn
- Post-indexed
  - EA = [Rn]
  - Rn is then overwritten with EA + offset
- Offset in register may be scaled by power of 2
  - shifted right or left a specified amount (0 – 31)
  - direction & shift amount are encoded in Rm field

ARM Addressing Modes

- Example:
  - LDR R0,[R1, -R2,LSL #4]!
    - R0 ← [[R1] – 16*[R2]]
    - R1 is overwritten by EA (! specifies writeback)

ARM Addressing Modes

ARM indexed addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[R*n*, #offset]	EA = [R*n*] + offset
Pre-indexed with writeback	[R*n*, #offset]!	EA = [R*n*] + offset; R*n* ← [R*n*] + offset
Post-indexed	[R*n*], #offset	EA = [R*n*]; R*n* ← [R*n*] + offset
With offset magnitude in R*m*:		
Pre-indexed	[R*n*, ± R*m*, shift]	EA = [R*n*] ± [R*m*] shifted
Pre-indexed with writeback	[R*n*, ± R*m*, shift]!	EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted
Post-indexed	[R*n*], ± R*m*, shift	EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift; and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

ARM Addressing Modes

- Register mode
  - Used for arithmetic & logic instructions
    - 2 source registers and a result register
- Absolute mode
  - If base register contains 0, 12-bit offset = absolute address

ARM Addressing Modes

- Immediate mode
  - Provided via pseudo-instructions
  - Format: LDR Rd,=value
  - Equal sign indicates immediate value
    - Examples:
    - LDR R2,=127 ; replaced by MOV R2,#127
    - LDR R2,=&ABCD3456 ;replaced by LDR R2,MEMLOC

      where MEMLOC contains hex ABCD3456

      “&” prefix denotes a hex value

ARM Addressing Modes

- Load 32-bit addresses
  - Examples:
  - ADR Rd,LOCATION ; loads 32-bit address into Rd
    - Assembler computes offset from current PC value
    - If LOCATION is in forward direction

      ADD Rd,R15,#offset ;is substituted
    - If LOCATION is in backward direction

      SUB Rd,R15,#offset ; is substituted
  - In either case, offset is an unsigned 8-bit number
  - Rotating the 8-bit number can give larger offsets

ARM Instructions

- General features
  - All instructions are encoded as 32-bit words
  - Only load and store instructions can access memory
  - Arithmetic and logic instructions use register operands
- Load and Store instructions
  - LDR & STR read and write 32-bit memory words
  - LDRB & LDRH read 8-bit or 16-bit values into low part of register
    - High bits within register are zero filled
  - LDRSB & LDRSH read 8-bit or 16-bit values into low part of register
    - High bits within register are filled with copies of sign bit

ARM Instructions

- Other Store instructions
  - STRB stores the low byte of a register into memory
  - STRH stores the low 16 bits of a register into memory
    - Must use half-word aligned address (i.e., multiple of 2)

ARM Instructions

- Block Transfer Instructions
  - Transfer a block of consecutive memory words to or from a subset of general purpose registers
    - List of registers must appear in increasing order

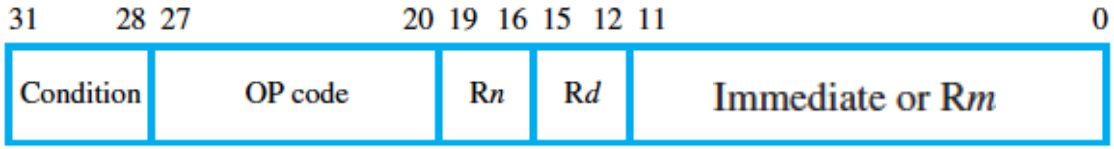
Example: base register R10 contains 1000 initially

```
LDMIA   R10!,{R0, R2, R5, R7}
```

Loads contents of locations 1000,1004,1008 and 1012 into registers R0, R2, R5 and R7. Suffix IA means “increment after” (i.e., post increment) So R10 is incremented by 4 after each word transfer and final value 1016 is written to R10 due to the “!” writeback indicator.

ARM Instructions

- Arithmetic instructions
  - Use registers or immediate operands
  - Assembly syntax is OP Rd,Rn,Rm
  - Machine code format:



Addition and Subtraction

The instruction

```
ADD   R0, R2, R4
```

performs the operation

$$R0 \leftarrow [R2] + [R4]$$

The instruction

```
SUB   R0, R6, R5
```

performs the operation

$$R0 \leftarrow [R6] - [R5]$$

ARM Instructions

The second source operand can be specified in the immediate mode. Thus,

```
ADD     R0, R3, #17
```

performs the operation

$$R0 \leftarrow [R3] + 17$$

The immediate operand is an 8-bit value contained in bits b_{7-0} of the encoded machine instruction. It is an unsigned number in the range 0 to 255. The assembly language allows negative values to be used as immediate operands. If the instruction

```
ADD     R0, R3, #−17
```

is used in a program, the assembler replaces it with the instruction

```
SUB     R0, R3, #17
```

ARM Instructions

- Shifting of second source register operand
  - Register second operand can be shifted before use
    - LSL (logical shift left)
    - LSR (logical shift right)
    - ASR (arithmetic shift right)
    - ROR (rotate right)
  - Specified after the register name:

    ADD R0,R2,R4,LSL #4
    - R4 is shifted left 4 bits (R4*16), then added to R2 & sum is placed into R0.

ARM Instructions

- Shifting of second source immediate operand
  - Contained in low byte of machine instruction (0 – 255)
  - Expanded into a limited number of 32-bit values
    - Programmer specifies value
    - Assembler zero-extends 8-bit value to 32 bits & rotates an even number of bits to generate the value
    - The shift amount and 8-bit value are encoded in the low-order 12 bits of the machine instruction

ARM Instructions

- Multiple-Word Operands
  - Carry flag, C, is used to perform multiple precision addition & subtraction
  - ADC (add with carry) & SBC (subtract with carry)

    Example - to add the 64-bit number in the register pair R2,R3 to the 64-bit number in the register pair R4,R5:

```
ADDS    R7,R3,R5    ;add low parts
ADC     R6,R4,R2    ;add high parts & include carry
```

    The S suffix causes the ADD to set the condition flags

ARM Instructions

- Multiplication
  - Two basic forms of multiply instruction
  - MUL R0,R1,R2 ; R0 ← [R1] × [R2]
    - Only the low 32 bits of 64-bit product are retained
  - MLA R0,R1,R2,R3 ; R0 ← ([R1] × [R2]) + [R3]
    - Called multiply and accumulate
    - Often used in signal-processing applications
  - Other versions of MUL and MLA are available
    - Generate 64-bit results
    - Handle signed and unsigned operands

ARM Instructions

- No provision for shifting or rotating operands in multiplication
- No divide instructions
  - Division must be done in software

ARM Instructions II

- Move Instructions
  - Copies an immediate value or contents of a register into a destination register
  - MOV Rd,Rm ; Rd ← [Rm]
  - MOV Rd, #value ; Rd ← value (8-bit immediate)
  - MVN R0,#4 ; moves one's complement
    ; representation of 4 into R0
    - To move 2's-complement representation of c into a register, MVN can be used with c-1 as immediate operand
    - E.g. MVN R0,#4 places 2's-complement representation of -5 into R0 (0xFFFFFFFB is -4 as 1's-comp, -5 as 2's-comp)
    - Assembler treats MOV R0,#-5 as pseudo-instruction and substitutes MVN R0,#4

ARM Instructions II

- Implementing shift and rotate instructions
  - There are no explicit ARM shift or rotate instructions
  - Shifting or rotating source register in Move instructions produces the same effect.
    - MOV Rd,Rm, LSL #4 shifts [Rm] left 4 bits and places result into Rd
    - MOV R3, R3, ROR #16 ; swap left & right halfwords in R3

ARM Instructions II

- Logic instructions
  - AND, OR, XOR (bit-wise AND, OR, XOR)
  - Bit-Clear
    - BIC R0, R0, R1 ; where there is a 1 bit in R1, the
      ; corresponding bit in R0 is cleared
    - Works by ANDing 1's-complement of [R1] with [R0]
- Bit Test instructions
  - TST R2,#1 ; sets Z flag = 0, if LSB of R2 is 1 (non-zero)
    - ANDs [R2] with immediate value
  - TEQ R2,#6 ; sets Z = 1, if [R2] = 6
    - XOR's the immediate value with the register

ARM Instructions II

- Compare instructions
  - CMP Rn,Rm ; sets condition flags based on [Rn] – [Rm]
    - Result is discarded
    - Second operand may be an immediate value
  - CMN Rn,Rm ; sets condition flags based on [Rn] + [Rm]
    - Compare negative
    - Second operand may be an immediate value
  - Second operand may be shifted before use

ARM Instructions II

- Branch instructions
  - Conditional branch instructions contain a 24-bit two's complement branch offset
    - offset is shifted left 2 bits & sign-extended to 32 bits
    - Offset + updated PC = branch target address
    - Instruction condition field (bits 31 – 28) indicate test to be performed (branch is taken if condition is met)
  - Conditions are defined below:

ARM Instructions II

Condition field encoding in ARM instructions.

Condition field $b_{31} \ldots b_{28}$	Condition suffix	Name	Condition code test
0 0 0 0	EQ	Equal (zero)	$Z = 1$
0 0 0 1	NE	Not equal (nonzero)	$Z = 0$
0 0 1 0	CS/HS	Carry set/Unsigned higher or same	$C = 1$
0 0 1 1	CC/LO	Carry clear/Unsigned lower	$C = 0$
0 1 0 0	MI	Minus (negative)	$N = 1$
0 1 0 1	PL	Plus (positive or zero)	$N = 0$
0 1 1 0	VS	Overflow	$V = 1$
0 1 1 1	VC	No overflow	$V = 0$
1 0 0 0	HI	Unsigned higher	$\overline{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
1 0 1 0	GE	Signed greater than or equal	$N \oplus V = 0$
1 0 1 1	LT	Signed less than	$N \oplus V = 1$
1 1 0 0	GT	Signed greater than	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	Always	
1 1 1 1		not used	

ARM Instructions II

- Subroutine Linkage instructions
  - Branch and Link instruction is used to call subroutines
    - BL SQRT ; call routine with name SQRT
    - Return address (address of instruction after BL) is loaded into R14 (the link register)
    - Link register must be saved on stack before a nested call
    - R13 is used as the processor stack pointer
  - STMFD is used to save (push) registers onto stack
    - Suffix FD means predecrement R13 toward lower addresses
  - LDMFD is used to restore (pop) contents of registers
  - Restoring PC (R15) returns to calling routine
    - MOV R15,R14 ; copy link register into PC to return

ARM Operating Modes

- Supports 7 Modes
  - System Mode
    - Privileged mode for exception handling
    - Uses same registers as user mode
    - Can only be entered from another exception
  - User Mode
    - For user applications
  - Five exceptions modes

ARM Operating Modes

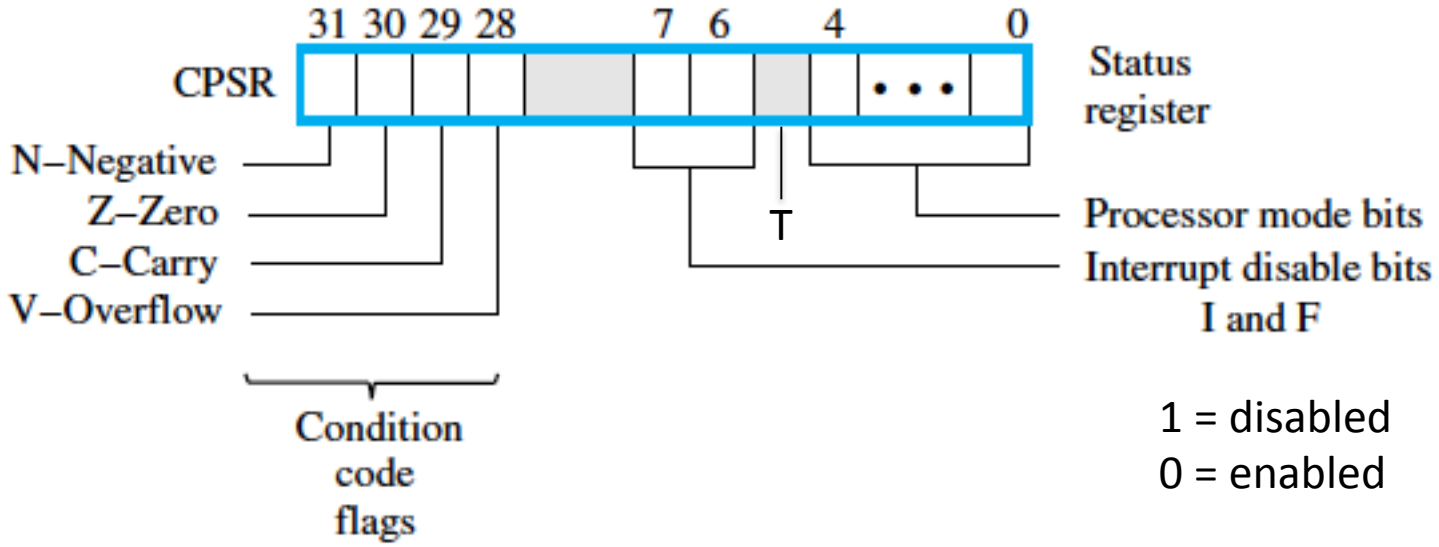
Exception Types

- Fast interrupt (FIQ) mode is entered when an external device raises a fast-interrupt request to obtain urgent service.
- Ordinary interrupt (IRQ) mode is entered when an external device raises a normal interrupt request.
- Supervisor (SVC) mode is entered on powerup or reset, or when a user program executes a Software Interrupt instruction (SWI) to call for an operating system routine to be executed.
- Memory access violation (Abort) mode is entered when an attempt by the current program to fetch an instruction or a data operand causes a memory access violation.
- Unimplemented instruction (Undefined) mode is entered when the current program attempts to execute an unimplemented instruction.

ARM Operating Modes

System Mode & exception modes are privileged modes.

Access to CPSR is allowed so I and F bits can be changed



User mode is unprivileged, so instructions that change CPSR are not available.

ARM Operating Modes

- Exception Handling
  - Vector table in low memory maps exception code

Exceptions and processor modes.

Exception	Processor mode entered	Vector address	Priority (Highest = 1)
Fast interrupt	FIQ	28	3
Ordinary interrupt	IRQ	24	4
Software interrupt	Supervisor (SVC)	8	–
Powerup/reset	Supervisor (SVC)	0	1
Data access violation	Abort	16	2
Instruction access violation	Abort	12	5
Unimplemented instruction	Undefined	4	6

ARM Operating Modes

- Banked Registers
  - Exceptions switch from user mode to one of 5 exception modes
  - Extra (banked) registers are substituted for some of the 16 normal registers used in System or User mode
  - Replaced registers are left unchanged (no need to save)
  - There is a set of banked registers for each exception mode

- *"Banked"* registers are used in exception handling

ARM Operating Modes

Accessible registers in different modes of the ARM processor.

User/System	FIQ	IRQ	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15	R15	R15	R15	R15	R15

Processor status register

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

ARM Operating Modes

- Actions taken when exception occurs

1. The contents of the Program Counter (R15) are loaded into the banked Link register (R14_mode) of the exception mode.
2. The contents of the Status register (CPSR) are loaded into the banked Saved Status register (SPSR_mode).
3. The mode bits of CPSR are changed to represent the appropriate exception mode, and the interrupt-disable bits I and F are set appropriately.
4. The Program Counter (R15) is loaded with the dedicated vector address for the exception, and the instruction at that address is fetched and executed to begin the exception-service routine.

ARM Operating Modes

- Return from exception
  - Operating System initializes R13_mode to point to top of stack area for each exception mode
  - Return from an exception handler is performed by copying the mode link register (R14_mode) into the program counter and copying the SPSR-mode register into the CPSR
    - Example: SUBS PC,R14_irq,#4
      - Sets PC = R14_irq – 4
      - The S suffix means copy SPSR_irq into CPSR
  - MOVS PC,R14_svc returns from software interrupt

ARM Operating Modes

Address correction during return from exception.

Exception	Saved address*	Desired return address	Return instruction
Undefined instruction	PC+4	PC+4	MOVS PC, R14_und
Software interrupt	PC+4	PC+4	MOVS PC, R14_svc
Instruction Abort	PC+4	PC	SUBS PC, R14_abt, #4
Data Abort	PC+8	PC	SUBS PC, R14_abt, #8
IRQ	PC+4	PC	SUBS PC, R14_irq, #4
FIQ	PC+4	PC	SUBS PC, R14_fiq, #4

*PC is the address of the instruction that caused the exception. For IRQ and FIQ, it is the address of the first instruction not executed because of the interrupt.

ARM Instructions II

	Memory address label	Operation	Addressing or data information
Assembler directives		AREA	CODE
		ENTRY	
Statements that generate machine instructions		LDR	R1, N
		LDR	R2, POINTER
		MOV	R0, #0
	LOOP	LDR	R3, [R2], #4
		ADD	R0, R0, R3
		SUBS	R1, R1, #1
		BGT	LOOP
		STR	R0, SUM
Assembler directives		AREA	DATA
	SUM	DCD	0
	N	DCD	5
	POINTER	DCD	NUM1
	NUM1	DCD	3, –17, 27, –12, 322
		END	

Assembly-language source program

Intel IA-32 Architecture

- Evolved from the x86 architecture
  - Uses 32-bit memory addresses
  - 32-bit registers
    - Supports 8-bit, 16-bit, 32-bit and 64-bit operands
  - Byte-addressable memory
    - Allows unaligned memory accesses
    - Employs little-endian memory storage order
  - Presents a CISC architecture to programmer
    - Breaks down CISC instructions into micro-ops
    - Micro-ops execute internally like RISC instructions

Intel IA-32 Architecture

- Memory Organization
  - Early x86 machines defined “word” as 16 bits
  - IA-32 “double-word” is a 32-bit item
  - “Quad-word” is a 64-bit item
  - Uses 32-bit memory addresses
  - Earlier x86 machines used 20-bit addresses
    - Derived from 16-bit segment registers and an offset
    - For code, stack and four data segments
  - IA-32 architecture supports flat address space
    - Maps all segments to a common 4 GB space
    - Shared by code, stack and data areas

Intel IA-32 Architecture

- IA-32 instructions vary in length
  - Unlike fixed-length instructions on RISC processors
  - Instruction encoding is complex
- Only 8 general purpose register
  - Can be treated as 8-bit or 16-bit parts
  - Maintains compatibility with older x86 instructions
- Supports CISC instructions
  - These do the work of an entire routine on RISC system
    - Such as string search and manipulation
  - Arithmetic/logic instructions can use memory operands
    - Not a load/store architecture

Intel IA-32 Architecture

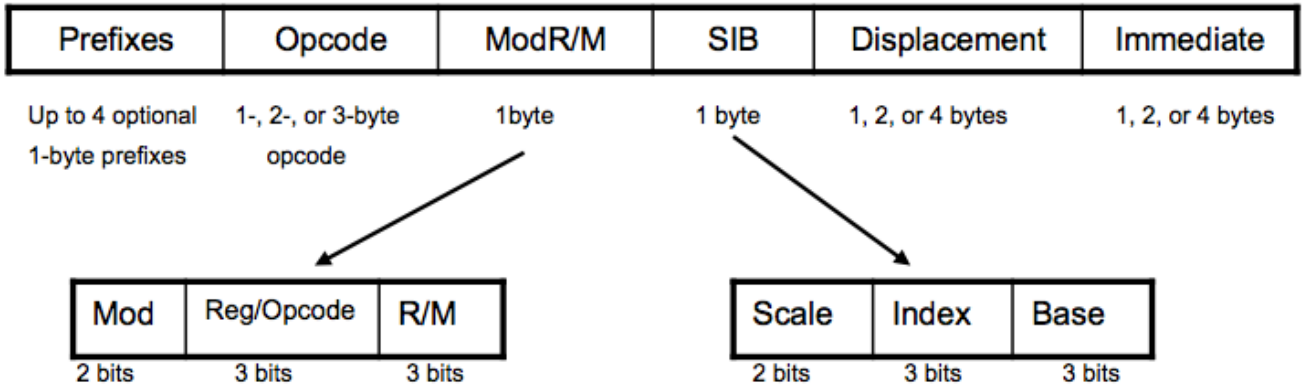
- Floating point unit uses separate set of 8 registers
  - Registers are organized as a stack ST(0) – ST(7)
  - Internal 80-bit extended precision float format is used
  - Converts IEEE 754 floats on the way to/from memory
- Supports SIMD vector type operations
  - MMX for integer operands
    - Multi-Media Extensions
  - SSE for floating point operands
    - Streaming SIMD Extensions

Intel IA-32 Architecture

- Performs I/O using special instructions
  - IN reads from an I/O device port
  - OUT sends data to an I/O device port
  - I/O ports are mapped to separate I/O space
  - Memory-mapped I/O can also be used
    - Using the normal memory access instructions
      - Like the only option for I/O on RISC systems

Intel IA-32 Architecture

- IA-32 Machine Instruction format is complex
  - Vary in size from 1 to 14 bytes
  - Complicates prefetching instructions



  - Opcode may be 1, 2 or 3 bytes
  - Other fields are optional and depend on the instruction type and the addressing mode used

IA-32 Registers

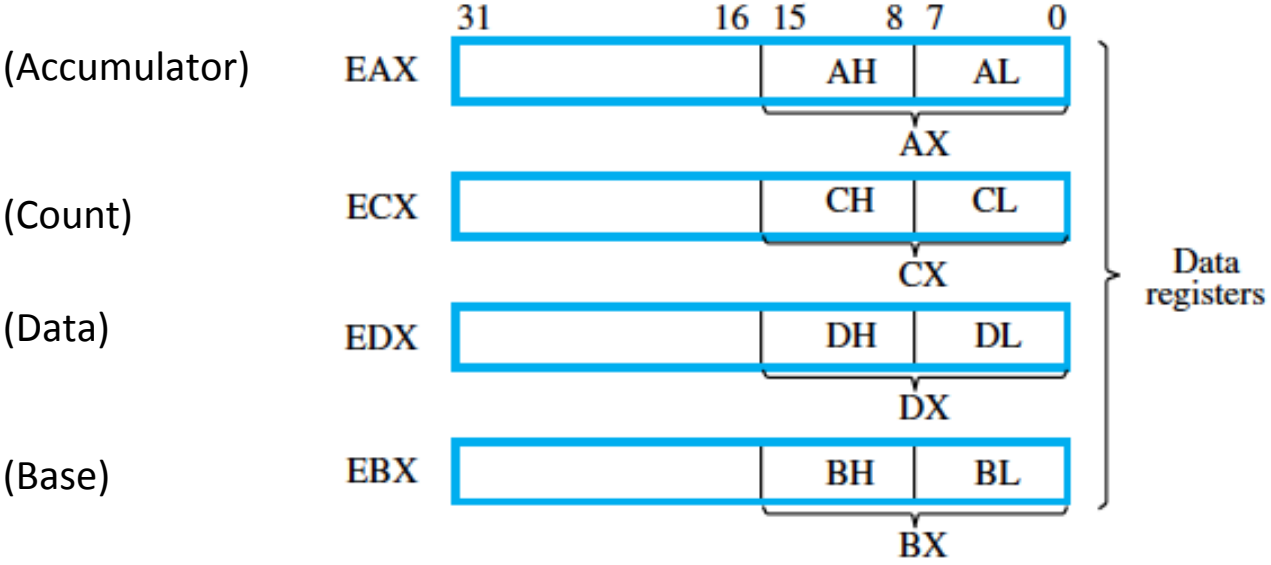
- Eight 32-bit General Purpose Registers
  - 4 Data Registers
  - 2 Pointer Registers
  - 2 Index Registers
- 32-bit Program counter
  - called Instruction pointer (EIP)
- 32-bit Status Register
    - Contains condition flags
- Register names, not numbers, are used

IA-32 Registers

- 8-bit Intel processors used 8-bit registers
  - Data registers were called A, B, C, and D.
- Later 16-bit processors used 16-bit registers
  - Data registers were called AX, BX, CX and DX.
- IA-32 Architecture uses 32-bit registers
  - Data registers are called EAX, EBX, ECX and EDX
    - E-prefix indicates 32-bit “extended” versions

IA-32 Registers

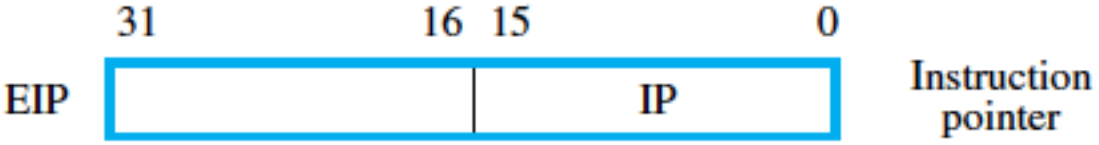
Data Registers



- EAX, ECX, EDX & EBX are 32-bit (extended)
- AX, CX, DX, and BX used for 16-bit items
- AH,AL, CH, CL, DH, DL, BH, BL are for 8-bit items
  - Maintains compatibility with earlier x86
  - Correctly runs 16-bit code on IA-32 processors

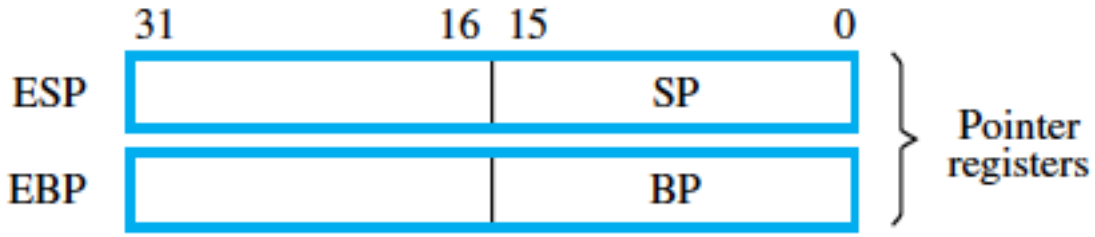
IA-32 Instruction Pointer

- Points to next instruction to execute
  - IP refers to the 16-bit instruction pointer
    - Limits code segment to 64 KB
  - EIP is the 32-bit extended instruction pointer
    - Code segment as large as 4 GB



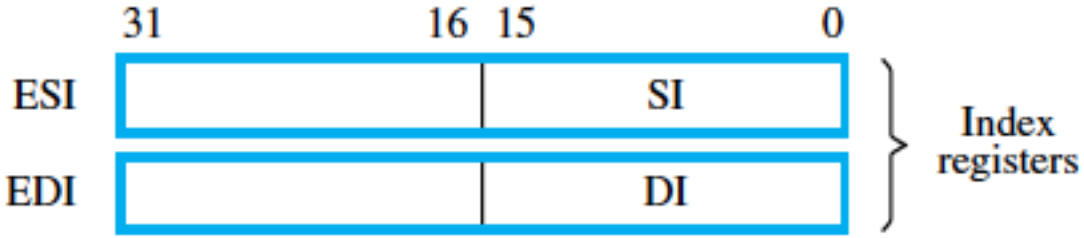
IA-32 Pointer Registers

- ESP & SP are 32-bit & 16-bit stack pointers
  - Points to stack in memory
- EBP & BP are 32-bit & 16-bit base registers
  - Used for call frames and data

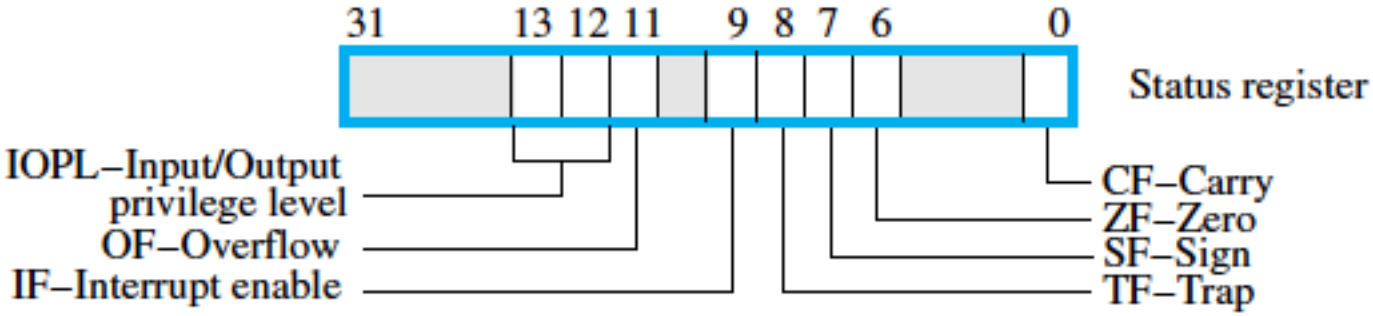


IA-32 Index Registers

- ESI & SI are 32-bit & 16-bit source index
- EDI & DI are 32-bit & 16-bit destination index
  - Used to reference memory operands
    - Array and string indices



IA-32 Status Register

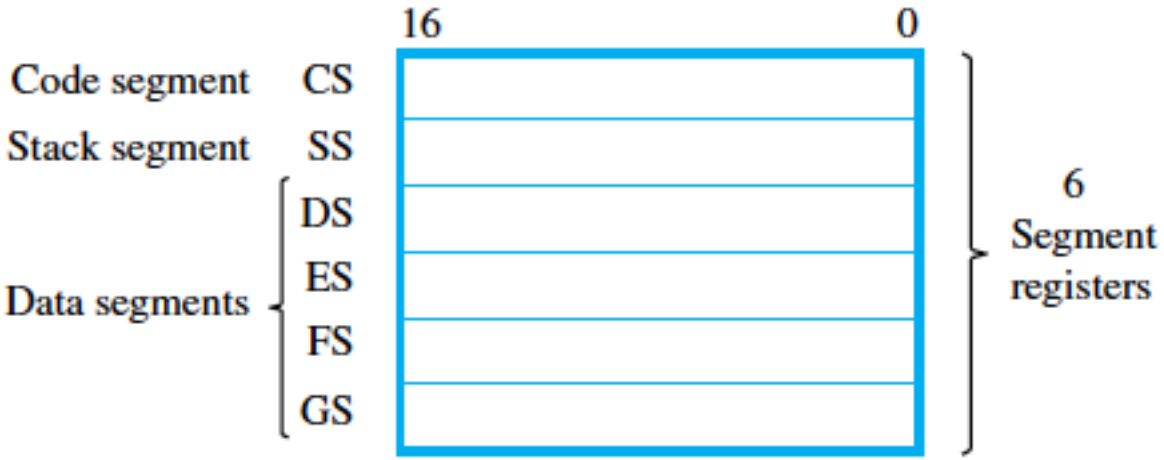


Also known as Flags register

- CF, ZF, SF, & OF are condition code flags
  - Reflect the result of arithmetic operations
- IOPL, IF, TF for I/O operations & interrupts
  - IF is interrupt enable/disable
  - TF is for single stepping through code
  - IOPL is set by the OS to control which operations are allowed

IA-32 Segment Registers

- Earlier x86 processors used memory segment
  - Segment starting address was held in a register
    - Segment registers are 16-bit
  - Segments were limited to 64 KB size

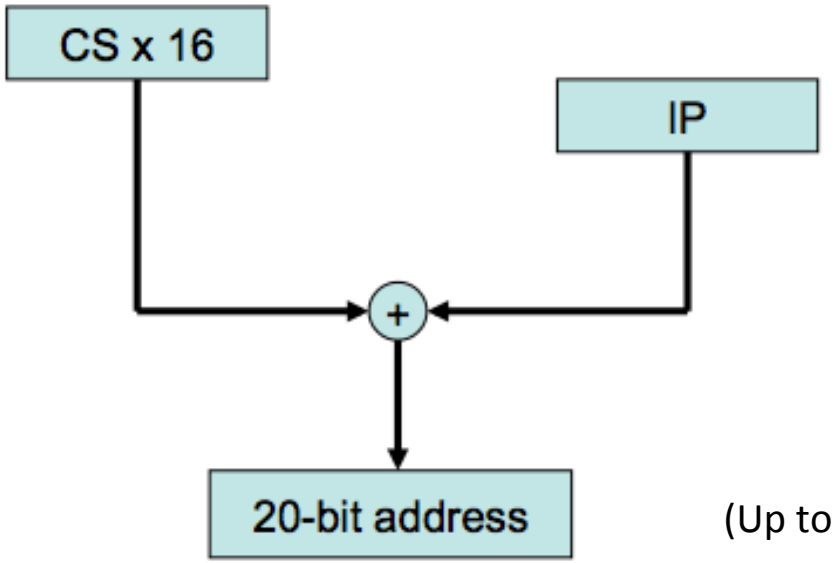


  - Memory address were specified as segment/offset
    - Segment_reg_contents:offset_value (xxxx:yyyy)

IA-32 Segment Registers

Address Generation Example:

The instruction Pointer register (IP) contains the offset within the current code segment from which the next instruction is fetched. On the 8086 this was a 16-bit register which is combined with the code segment register as shown below to produce the 20-bit memory address:



(Up to 1 MB address space)

Memory operand addresses are generated in a similar way, but one of the other segment registers is combined with a 16-bit offset.

IA-32 Segment Registers

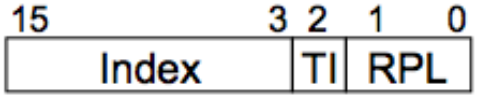
- Use of registers is based on Processor Mode
  - Real Mode
    - Segment registers are used the same as on x86 processors
      - Addresses are 20 bits
  - Protected Mode uses more complex segmentation
    - Segment registers contain index into descriptor table
      - 1 Local Descriptor table (LDT) for each task
      - Global Descriptor table (GDT) shared by all tasks
      - Interrupt Descriptor table (IDT) used by O.S.
      - Table entries contain 32-bit base address
      - Start address + 32-bit offset = 32-bit linear address
    - Overlapping segments provides a flat address space
    - Segments can be up to 4 GB in size

IA-32 Segment Registers

Protected Mode Segment Registers

Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.

In protected mode, every segment register has a "visible" part and an "invisible" The visible part is referred to as the segment selector. The invisible part is automatically loaded by the processor from a descriptor table.



Segment register Contents (protected mode)

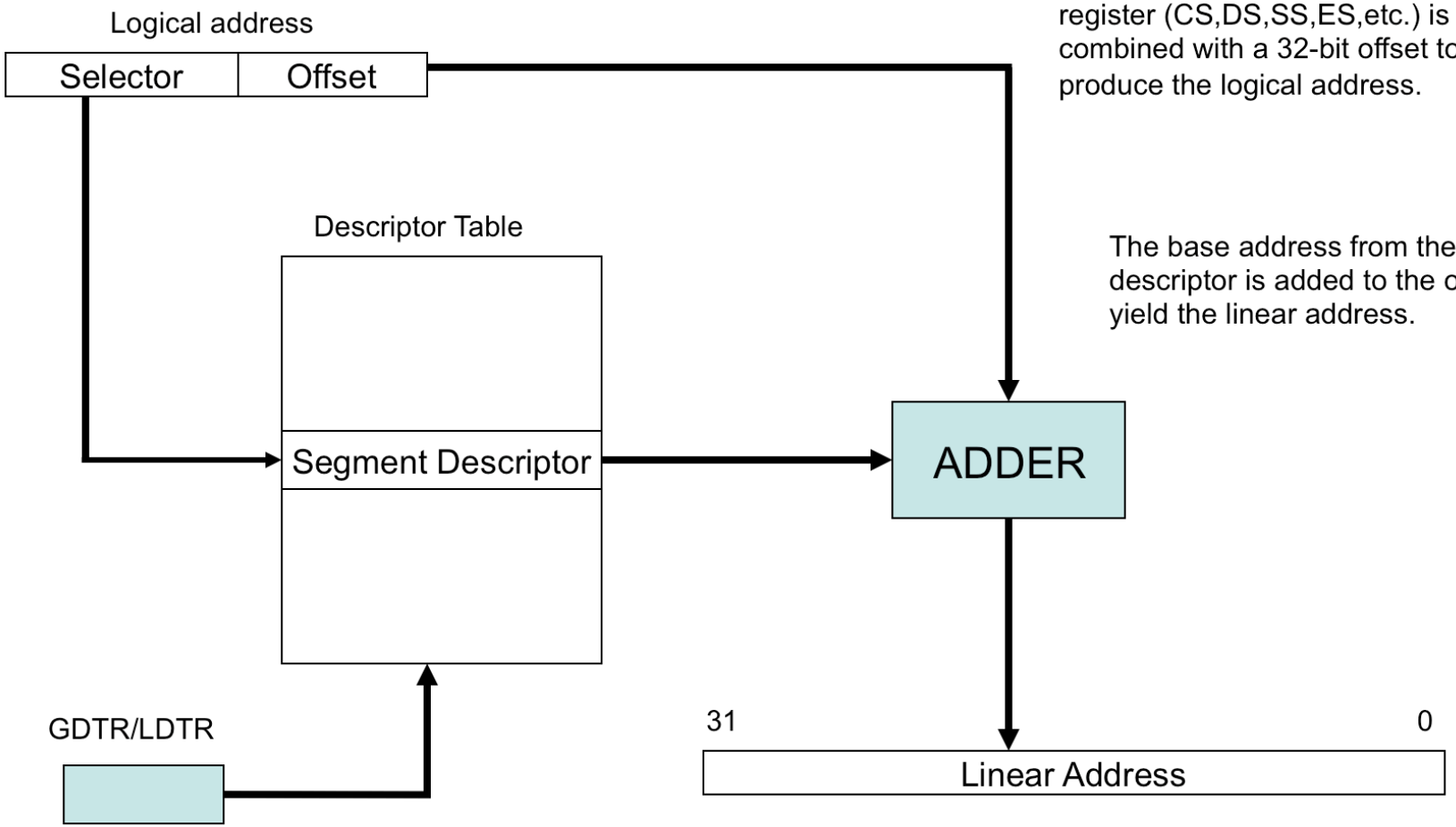
RPL – request privilege level

TI - table indicator 0=GDT, 1=LDT

Index – points to descriptor in table

IA-32 Segment Registers

Protected Mode Address Translation



A segment selector in the segment register (CS,DS,SS,ES,etc.) is combined with a 32-bit offset to produce the logical address.

The base address from the segment descriptor is added to the offset to yield the linear address.

Contains address of descriptor table

If paging is not used, then the linear address is the physical address.

IA-32 Addressing Modes

- IA-32 addressing modes are CISC like
  - Large number of flexible addressing modes
    - Register Mode – operands are in registers
    - Immediate Mode – instruction contains operand
    - Direct mode – instruction contains operand address
    - Register indirect – operand address in register
    - Base with displacement – [reg]+displacement = op adrs
    - Index with displacement – [reg]*scale + disp. = op adrs
    - Base with index – op adrs = [breg] + [ireg]*scale
    - Base with index & displacement
      - op adrs = [base_reg] + [index_reg]*scale + displacement

IA-32 Addressing Modes

- IA-32 addressing modes are CISC like
  - Large number of flexible addressing modes
    - Register Mode – operands are in registers
    - Immediate Mode – instruction contains operand
    - Direct mode – instruction contains operand address
    - Register indirect – operand address in register
    - Base with displacement – [reg]+displacement = op adrs
    - Index with displacement – [reg]*scale + disp. = op adrs
    - Base with index – op adrs = [breg] + [ireg]*scale
    - Base with index & displacement
      - op adrs = [base_reg] + [index_reg]*scale + displacement

IA-32 Addressing Modes

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = an 8- or 32-bit signed number
Location = a 32-bit address
Reg, Reg1, Reg2 = one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register.
Disp = an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S = a scale factor of 1, 2, 4, or 8

IA-32 Addressing Modes

- Instructions can have 0, 1 or 2 operands
  - Two-operand syntax: OP destination,source
  - Examples based on MOV instruction:
    - MOV EBP,EAX copies EAX reg into EBP reg
    - MOV EAX,25 copies 32-bit constant into EAX
    - MOV AX,320 copies 16-bit constant into AX
    - MOV AL,125 copies 8-bit constant into AL
    - MOV EAX,LOC1 copies 32 bits at LOC1 into EAX
    - MOV EBX, OFFSET LOC1
      - Puts address LOC1 into EBX
    - MOV EAX,[EBX]
      - EAX = 32-bit contents of location whose address is in EBX

IA-32 Addressing Modes

- Base with displacement examples
  - Assume that EBP contains 2000
  - MOV EAX,[EBP+60]
    - Copies contents of doubleword (32 bits) at 2060 into EAX
  - MOV AL,[EBP+60]
    - Copies contents of byte (8 bits) at 2060 into AL
  - MOV [EBP+67],AH
    - Copies contents of AH into byte at address 2067
  - MOV [EPB+100],28 size of constant is unclear
    - MOV BYTE PTR [EBP+67],28 for 8-bit
    - MOV WORD PTR [EBP+67],28 for 16-bit
    - MOV DWORD PTR [EBP+67],28 for 32-bit

IA-32 Addressing Modes

- Base with index & displacement example
  - Assume that [EBP] = 2000 & [ESI] = 0
  - MOV EAX,[EBP + ESI*4 + 100]
    - Copies contents of doubleword (32 bits) at 2100 into EAX
  - To copy contents of doublewords at 2100, 2104, 2108, etc. place in loop and increment ESI by 1 for each iteration
    - E.g., to step through array of 32-bit elements
    - Scale factor of 2 for 16-bit elements
    - Scale factor of 1 for 8-bit elements or characters

IA-32 Instructions

- Instructions can have 0, 1 or 2 operands
  - Zero operand examples:
    - PUSHAD pushes all 8 data regs onto stack
    - POPAD restores all 8 regs from stack (popping)
  - One operand examples:
    - INC EDI adds 1 to EDI register
    - DEC EBX subtracts 1 from EBX register
  - Two operand examples:
    - ADD EAX,EBX
    - MUL EBX,511

IA-32 Instructions

- Load-effective-address instruction
  - LEA EAX,LOC1
    - Puts address corresponding to LOC1 into EAX
    - Address is part of the instruction
  - MOV EAX,OFFSET LOC1 has same effect
    - Address of LOC1 is known by assembler
  - LEA EBX,[EBP + 8]
    - Puts address = (contents of EBP) + 8 into EBX at runtime
    - Assembler can't know what EBP will contain

IA-32 Instructions

- Arithmetic instructions
  - May use all register operands
  - one operand may be in memory
  - Operands can be 8-bit or 32-bit
  - Examples:
    - ADD EAX,EBX
    - ADC EAX,EBX
    - SUB EBX,[EAX+4]
    - SBB EAX,EBX
    - CMP [EBX + 10],AL

IA-32 Instructions

- Multiply instructions
  - IMUL EBX implicit multiplicand is EAX
    - Computes [EAX]*[EBX]
    - Puts 64-bit product into EDX,EAX (high,low)
  - IMUL EBX,[EBP]
    - Puts low 32 bits of product dest*src into dest
    - OF flag = 1 if high half of 64-bit product is nonzero
  - IMUL does signed multiply
  - MUL does unsigned multiply
  - Source can be immediate, register or in memory
  - Destination must be a register

IA-32 Instructions

- Division instructions
  - IDIV *src*
    - Divides EDX,EAX pair by src
    - 32-bit value in EAX must be sign extended to 64-bits
      - CDQ converts EAX into 64 bits in EDX,EAX
    - Sets EAX=quotient and sets EDX=remainder
    - Division by zero causes an exception
  - DIV does unsigned divide
  - Source can be immediate, register or in memory

IA-32 Instructions

- Conditional Jump instructions
  - All branches are called “jumps” with IA-32
  - Tests condition codes to decide

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF ∨ ZF = 0
JBE	Unsigned below or equal	CF ∨ ZF = 1
JGE	Signed greater than or equal	SF ⊕ OF = 0
JL	Signed less than	SF ⊕ OF = 1
JG	Signed greater than	ZF ∨ (SF ⊕ OF) = 0
JLE	Signed less than or equal	ZF ∨ (SF ⊕ OF) = 1

E.g.: JG EXIT

∨ denotes OR

⊕ denotes XOR

IA-32 Instructions

- Conditional Jump instructions
  - Assembler generates signed offset relative to next location
    - One-byte offset if in the range -128 to +127
    - Otherwise a 4-byte offset is generated
  - IP + Offset = target address
- Loop Instruction

```
        MOV     ECX,NUM_PASSES
START:
        .
        .
        .
        DEC     ECX
        JG      START
```



```
        MOV     ECX,NUM_PASSES
START:
        .
        .
        .
        LOOP    START
```

LOOP instruction implicitly decrements ECX and branches if > 0

IA-32 Instructions

- JMP is the unconditional jump
  - Uses 1-byte or 4-byte signed offset
  - May also use other addressing modes for target address
    - Example: JMP [JUMPTBLE + ESI * 4]
    - Uses ESI as Index into table of jump addresses
    - Implements high-level language Case statement

IA-32 Instructions II

- Logic Instructions
  - AND, OR, XOR
    - All use 2 operands and put result into destination
    - Example: AND EBX,EAX
  - NOT uses a single operand
    - Takes 1's complement (i.e. flips each bit)
    - Example: NOT EAX
  - NEG negates it's single operand (takes 2's complement)
- Test Instruction Format: TEST dst,src
  - Sets ZF=0 in any matching bits are set in src and dst
  - Operands are ANDed but neither operand is modified
  - destination may be a register or in memory
  - source may be a register, in memory or immediate

IA-32 Instructions II

- Bit Test Instructions
  - BT op1,n Sets CF = bit n within op1
  - BTC op1,n Sets CF = bit n within op1 & flip bit n within op1
  - BTR op1,n Sets CF = bit n within op1 & clear bit n within op1
  - BTS op1,n Sets CF = bit n within op1 & set bit n within op1
  - The operand op1 may be a register or in memory

  - Other examples of available CISC type instructions include:
  - BSF & BSR bit scan forward and bit scan reverse
  - SCANSB, SCANSW, SCANSD for scanning strings in search of a particular value (direction of scan is controlled by DF flag in status register).

IA-32 Instructions II

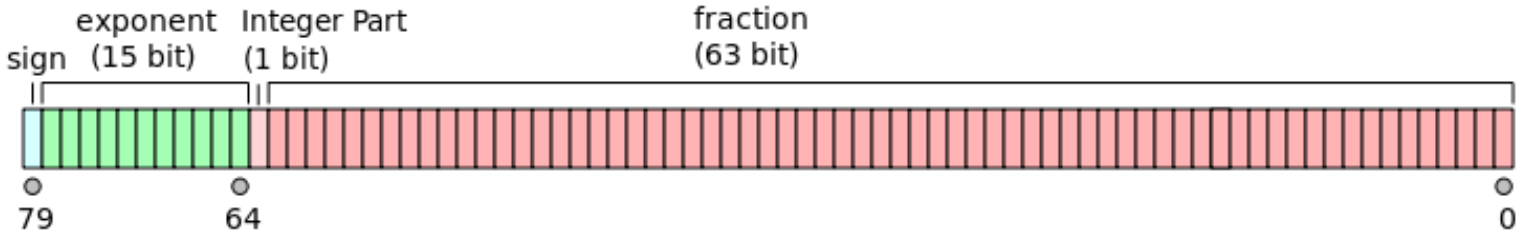
- Shift Instructions
  - SHL (shift left logical)
  - SHR (shift right logical)
  - SAL (shift left arithmetic; same as SHL)
  - SAR (shift right arithmetic)
- Rotate Instructions
  - ROL (Rotate left without the carry flag CF)
  - ROR (Rotate right without the carry flag CF)
  - RCL (Rotate left including the carry flag CF)
  - RCR (Rotate right including the carry flag CF)
- Shift & Rotate instruction format: OP dst,count
- dst is shifted (Any addressing mode can be used)
- Count must be an 8-bit immediate or in the 8-bit CL register (low byte of ECX)

IA-32 Instructions II

- Subroutine Linkage
  - The CALL instruction is used to call subroutines
  - Example: CALL ROUTINE
    - The address of the instruction following the CALL is the return address
    - The return address is pushed onto the stack before the transfer
    - ESP register points to the top of stack (TOS)
    - The stack grows downward toward lower addresses
    - PUSHAD can be used to save all 8 general purpose registers before call
    - The subroutine uses POPAD to restore the registers before returning
    - PUSH & POP may be used to handle individual registers or data items
    - RET returns control to caller by poping TOS into EIP
    - EBP register points to call frame on stack
    - LIFO stack facilitates nested subroutine calls

IA-32 Floating point Instructions

- Floating Point Unit (FPU) executes F.P. instructions
  - Has its own set of 8 registers
  - FPU & CPU execute their instructions concurrently
  - FPU uses 80-bit internal format for all operations
    - Converts IEEE-754 formatted operands to/from 80-bit format



Bit 79 is the sign bit (s)
Bits 64 – 78 give the 15-bit excess-16383 exponent (e)
Bit 63 = 1 for normalized values, = 0 for denormalized (i)
Bits 0 – 62 give the 63-bit fraction (f) [there is no hidden bit]
Number point is between bits 62 & 63
Value represented is $(-1)^s \times i.f \times 2^{e-16383}$

IA-32 Floating point Instructions

- Floating Point Stack
  - The 8 FPU registers are organized as a stack
    - ST(0) to ST(7) from top to bottom
    - Modulo-8 pointer defines top of stack
  - FLD copies its single memory operand onto stack ST(0)
  - FST writes contents of ST(0) into memory
    - FSTP does the same but also pops ST(0) from stack
  - FIST converts ST(0) to 32-bit integer & stores in memory
    - FISTP does same but also pops ST(0)

IA-32 Floating point Instructions

- Floating Point Load/Store examples
  - FLD DWORD PTR[EAX]
    - Converts 32-bit float in memory into 80-bit float in ST(0)
    - 80-bit value is pushed onto FPU stack
  - FST QWORD PTR [EDX + 8]
    - converts 80-bit float in ST(0) into 64-bit float in memory
  - FST writes contents of ST(0) into memory
    - FSTP does the same but also pops ST(0) from stack
  - FISTP [EDX + 8]
    - Pops ST(0) converts to 32-bit integer and stores in memory

IA-32 Floating point Instructions

- Floating Point Arithmetic Instructions
  - FADD QWORD PTR[EAX]
    - Converts 32-bit float in memory into 80-bit float & adds to ST(0)
  - FSUBP ST(1),ST(0)
    - puts ST(1)-ST(0) into ST(1) , pops ST(0), so result is now ST(0)
  - FSUBR ST(2),ST(0)
    - Puts ST(0) – ST(2) into ST(2) [reverse subtract]
    - FDIVR is similar
  - FISUB DWORD PTR [EDX + 8]
    - Converts 32-bit integer, sets ST(0) = ST(0) – float equivalent
    - FIADD, FIMUL & FIDIV are similar

IA-32 Floating point Instructions

- Floating Point Compare Instructions
  - FCOMI ST(0),ST(4)
    - Compares register with ST(0) and sets condition codes
    - ST(0) must be destination operand
  - FCOMIP ST(0),ST(2)
    - Similar to FUCOMI but also pops stack
  - Additional Float Instructions
    - FCHS changes sign of implicit operand ST(0)
    - FABS takes absolute value
    - FSQRT computes square root of ST(0)
    - FSIN & FCOS compute sine and cosine of ST(0)
    - FLDZ & FLD1 push 0.0 & 1.0 as 80-bit float
    - FLDPI pushes π as 80-bit float

Multimedia & Vector Instructions

- IA-32 has Vector or SIMD instructions
  - Images can be represented as matrices of pixels
    - Color & brightness are encoded in each element
    - Multiple pixels can be packed into a single elements
    - Simple arithmetic & logic operations are applied
    - SIMD instructions act on multiple pixels in parallel

Multimedia & Vector Instructions

- IA-32 has multimedia extensions (MMX)
  - Multiple data elements are packed into 64-bit quadwords
  - Operands can be in memory or in FPU registers
    - FPU registers correspond to MM0 – MM7
      - Lowermost 64 bits within each 80-bit register are used
    - MMX registers are not managed as a stack
  - Example MMX instructions
    - MOVQ MM0,[EAX] loads 64 bits from memory
    - MOVQ MM3,MM4 copies MM4 into MM3

Multimedia & Vector Instructions

- Example MMX arithmetic instructions
  - PADDB MM2,[EBX] add packed bytes
    - adds 8 memory bytes to corresponding bytes in MM2
    - The sums are computed in parallel
    - B suffix indicates 8 bytes
    - W indicates four 16-bit words
    - D indicates two 32-bit doublewords
    - Q indicates single 64-bit quadword
  - Other operations are also available:
    - PSUB
    - PMUL
    - PMADD
    - PAND
    - POR
    - PXOR

Multimedia & Vector Instructions

- Streaming SIMD extensions (SSE)
  - Handle packed 128-bit double quadwords
  - 8 additional 128-bit registers XMM0 to XMM7
  - MOVAPS & MOVUPS transfer between memory and registers
    - PS suffix indicates packed single-precision float values
    - A or U indicates aligned or unaligned (16-bit word aligned)
- Examples:
  - MOVUPS XMM3,[EAX]
    - Copies 4 32-bit floats from memory into XMM3
  - MOVUPS XMM4,XMM5 copies XMM5 into XMM4
  - ADDPS XMM0,XMM1
    - Adds 4 corresponding pairs of 32-bit floats
    - SUBPS, MULPS & DIVPS are also available

IA-32 Instructions II

- Memory mapped Input/Output
  - The code below is an example illustrating the use of mapped I/O
  - Polls keyboard to echo characters until CR (carriage return) is hit

```
        LEA   EBX, LOC          Initialize register EBX to point to the
                                  address of the first location in main memory
                                  where the characters are to be stored.
READ:   BT    KBD_STATUS, 1     Wait for a character to be entered
        JNC   READ                in the keyboard buffer KBD_DATA.
        MOV   AL, KBD_DATA      Transfer character into AL (this clears KIN to 0).
        MOV   [EBX], AL         Store the character in memory
        INC   EBX                 and increment pointer.
ECHO:   BT    DISP_STATUS, 2    Wait for the display to become ready.
        JNC   ECHO
        MOV   DISP_DATA, AL     Move the character just read to the display
                                  buffer register (this clears DOUT to 0).
        CMP   AL, CR            If it is not CR, then
        JNE   READ                branch back and read another character.
```

IA-32 Instructions II

- Port Mapped I/O (isolated I/O) is common with IA-32
  - Requires special Input/Output instructions
  - Examples:

```
IN  AL,DX   ; reads one byte from the devices whose port number is in DX
IN  AX,DX   ; reads two bytes of data from the port
IN  EAX,DX  ; reads 4 bytes

OUT  61h,AX  ; sends two bytes to output port number 61h
OUT  DX,AL   ; sends one byte to the port whose number is in DX
```

Port mapped scheme uses I/O address space separate from the program address space.

IA-32 Instructions II

- IA-32 Interrupts and Exceptions
  - Two interrupt lines
    - NMI non-maskable is always accepted by processor
    - INTR user interrupt is maskable
      - Enabled/disabled using IF flag within status register
      - Accepted if priority > privilege level of currently running program
  - Events other than external interrupts cause exceptions
    - Vector number is assigned to each interrupt or exception
    - This index identifies an IDT (interrupt descriptor table) entry
    - Table entry contains starting address of corresponding handler
    - I/O devices are connected through an APIC
      - Advanced Programmable Interrupt Controller
      - Controller implements priority scheme and sends vector number to CPU

IA-32 Instructions II

- Actions taken for Interrupts or Exceptions

1. Push status register, code segment register and EIP onto processor stack
2. For exceptions due to abnormal condition, push cause of exception
3. Disable further interrupts of the same type
4. Use vector number to load appropriate handler address from IDT into EIP

When finished, the handler uses the IRET instruction to resume the program.

IRET pops EIP, code segment register and status register from stack
Handler must undo any changes it made to stack pointer before executing IRET

Sparc V8 Overview

- **SPARC** stands for **S**calable **P**rocessor **Arc**hitecture.
- developed by Sun Microsystems in the 1980s.
- based on the RISC II designed at UC Berkeley in early 1980s
- "Scalable" from embedded systems to servers
- RISC design that shares many features with the MIPS system
- There are some major differences between Sparc V8 & MIPS
- Sparc V8 is the only version examined here

Sparc V8 Overview

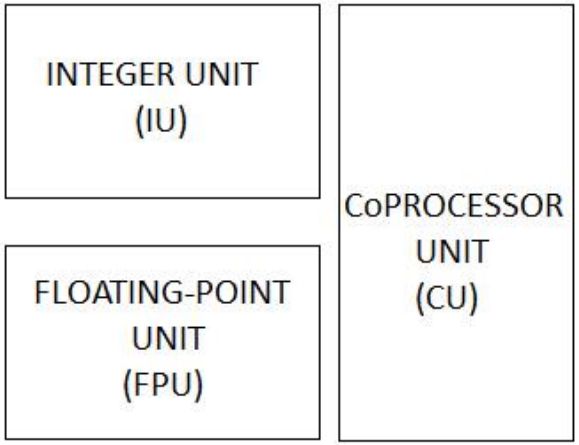
- Features Shared with MIPS:
  - 32-bit registers and addresses
    - Register 0 is hardwired to 0
  - Byte-addressable memory
    - 8-bit bytes, 16-bit halfwords and 32-bit words
  - Load/store architecture
  - Enforces memory alignment
  - Employs big-endian memory storage order
  - Delayed branches
  - Passes arguments via registers
    - Using stack when needed

Sparc V8 Overview

- Features that differ from MIPS:
  - May have hundreds of registers
  - Registers are grouped into “windows”
    - 32 registers are visible at one time
    - Windows slide and overlap to pass arguments
  - Branch delay slot can be “annulled”
  - Has next PC (NPC) as well as PC (program counter)
    - PC points to current instruction
    - NPC points to instruction to be executed next
  - Instructions can set condition codes or not

Sparc V8 Overview

- Sparc V8 Organization



  - Units have separate register sets
  - Units operate in parallel
  - FPU has 32 floating point registers (not windowed)

Sparc V8 registers

Names of registers visible to a user program at any one time:

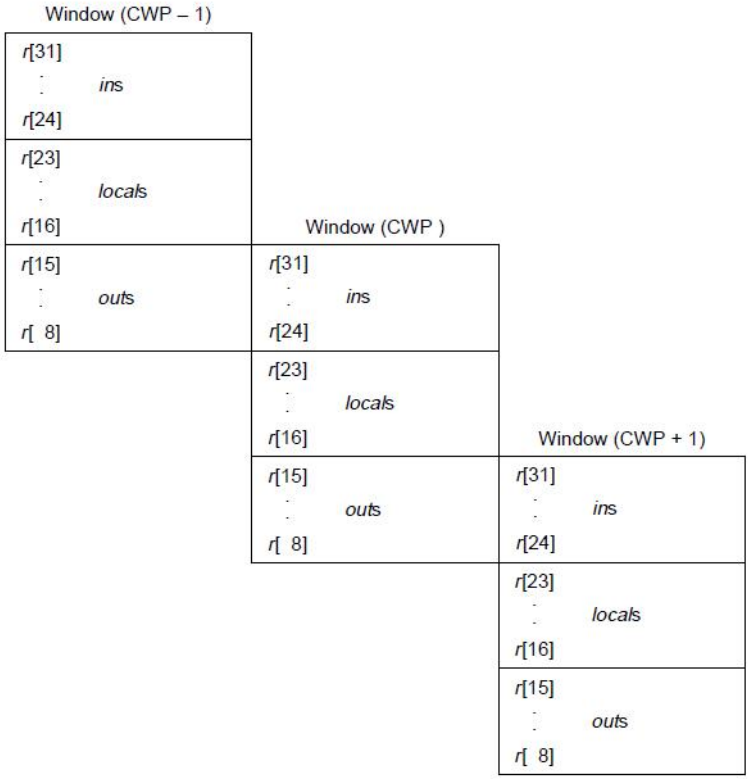
Register	Alias	Usage
%g0	%r0	Hardwired to zero
%g1	%r1	The next seven are global registers
%g2	%r2	
%g3	%r3	
%g4	%r4	
%g5	%r5	
%g6	%r6	
%g7	%r7	
%o0	%r8	First of six registers for local data and subroutine arguments
%o1	%r9	
%o2	%r10	
%o3	%r11	
%o4	%r12	
%o5	%r13	
%sp	%r14, %o6	Stack pointer
%o7	%r15	Linkage register containing subroutine return address
%l0	%r16	First of eight registers for local variables
%l1	%r17	
%l2	%r18	
%l3	%r19	
%l4	%r20	
%l5	%r21	
%l6	%r22	
%l7	%r23	

Sparc V8 registers

Caller	Callee	Usage
%o0	%i0	first argument
%o1	%i1	second argument
%o2	%i2	third argument
%o3	%i3	fourth argument
%o4	%i4	fifth argument
%o5	%i5	sixth argument
%o6	%i6	frame pointer
%o7	%i7	return address

Register Window

- Current window pointer *(CWP)*



There can be up to 32 register windows

Each window contains 24 registers

Windows can slide to pass arguments

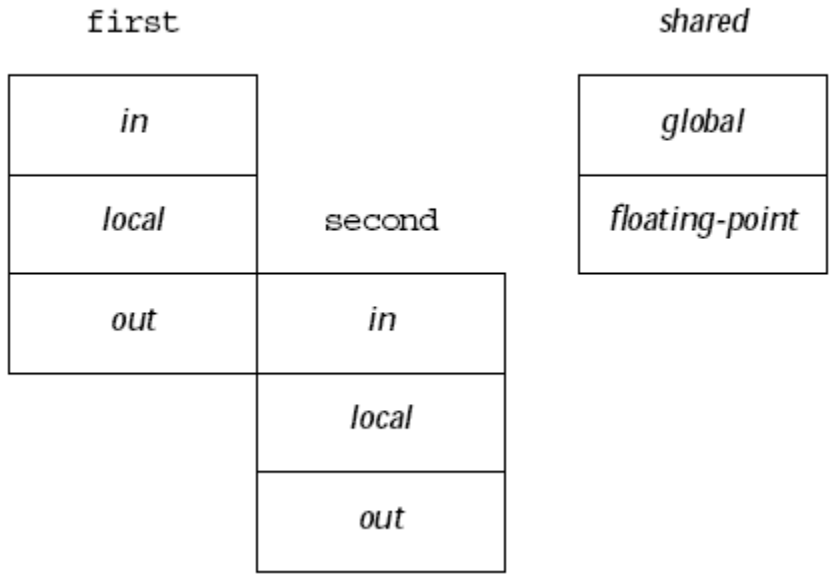
Windows overlap

CWP identifies current window

Register Window

- Example: function first calls second:

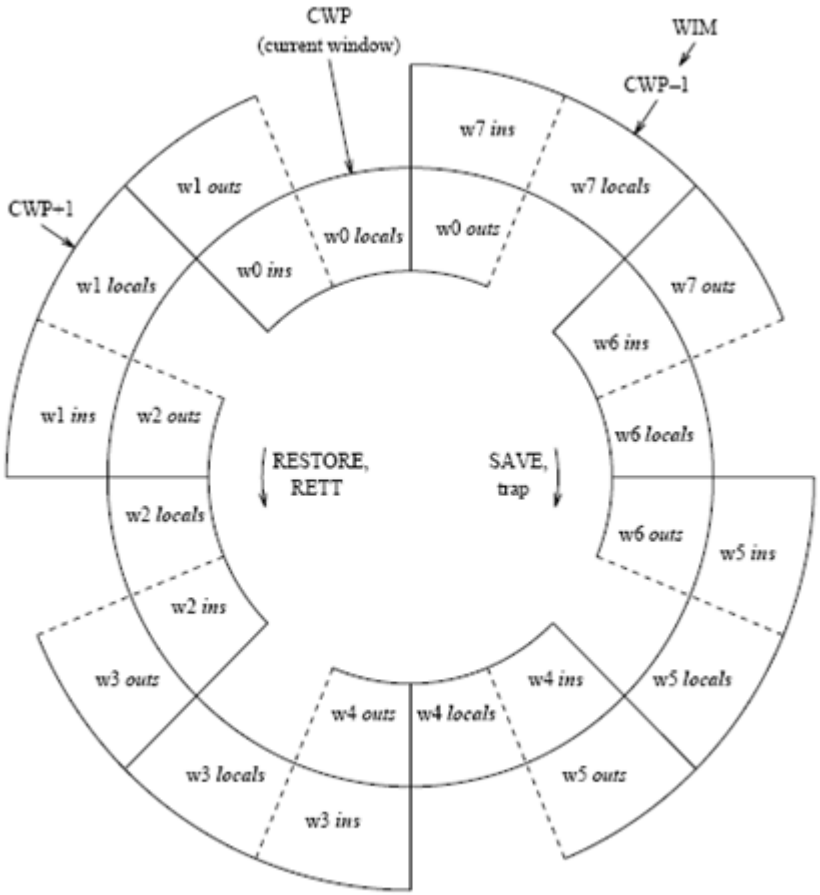
```
first()
{
     ...
     second();
     ...
}
```




```
second:
        save    %sp, -80, %sp
```

restore instruction is used to unwind stack and slide window back

Register Window



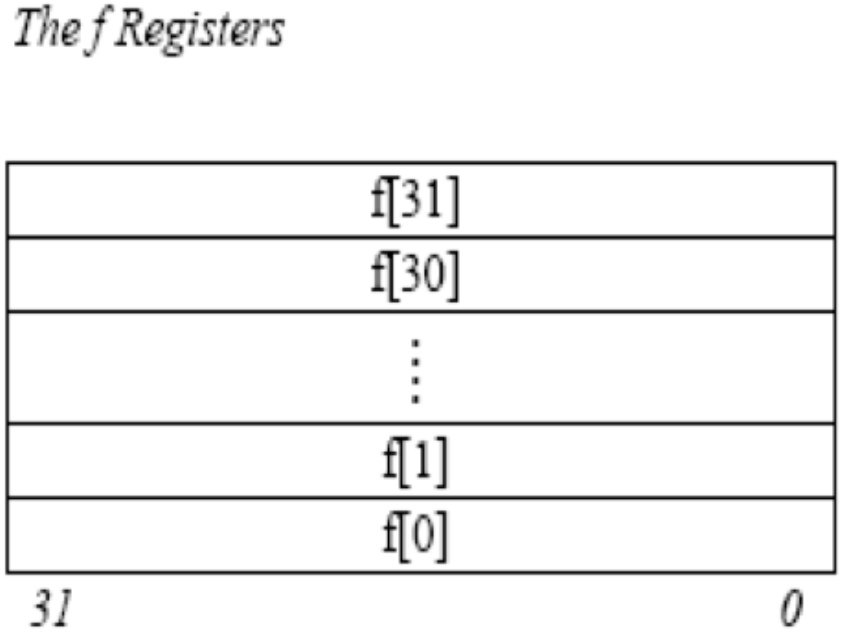
Sparc V8 registers

Special-purpose registers

PC	program counter	Holds the address of the currently executing instruction
nPC	next PC	Holds address used to fetch next instruction to be executed
PSR	processor state register	Holds state of CPU, condition codes, mode, trap enable, bit, etc.
WIM	window-invalid mask	Indicates when window underflow or overflow should occur
TBR	trap base register	Holds address of trap table and indicates trap type
Y	Y register	High part of product for mul ; high part of dividend for div
FSR	floating point state	Holds floating point mode and status, controls rounding and floating point traps

Floating Point registers

Floating point unit has a single set of 32 floating point registers



Each float register is 32 bits wide.

Sparc V8 Addressing Modes

Supported Modes:

- Source operands are on left, result on right in assembly instructions

- Register mode (all operands in registers)
  - add %g2,%o4,%o1 [%g2] + [%o4] → %o1

- Immediate mode
  - sub %o2, 23, %g4 [%o2] – 23 → %g4

- Base register with displacement
  - ld [%g2 + 8], %o3 [%g2 + 8] → %o3

- Register indirect with index
  - stb %o4, [%g4 + %o2] [%o4] → [%g4 + %o2]
  - stb %o4, [%g4 + %g0] [%o4] → [%g4] (%g0 always 0)

Sparc V8 Addressing Modes

Loading a 32-bit constant or address into a register

To load an address into %g2:

```
sethi   %hi(X), %g2          high 22 bits of address
or      %g2, %lo(X), %g2     merge in low 10 bits
```

To load the constant 0x4A3C4098 into %o2:

01001010001111100010000 0010011000

```
sethi     0x128F10, %o4      high 22 bits
or        %o4, 0x98,  %o4    low 10 bits
```

Synthetic instructions:

```
set   X, %g2
set   0x4A3C4098, %o4
```

%hi() and %lo() are implemented by the assembler

Sparc V8 Instructions

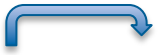
Instructions fall into following categories :

- Load/store
- Arithmetic/logical/shift
- Control transfer
- Read/write control register
- Floating-point/Coprocessor operate

Sparc V8 Instructions

Load Integer Instructions

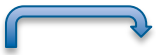
opcode	*op3*	*operation*
LDSB	001001	Load Signed Byte
LDSH	001010	Load Signed Halfword
LDUB	000001	Load Unsigned Byte
LDUH	000010	Load Unsigned Halfword
LD	000000	Load Word
LDD	000011	Load Doubleword



ld [%sp – 4], %o2

Store Integer Instructions

opcode	*op3*	*operation*
STB	000101	Store Byte
STH	000110	Store Halfword
ST	000100	Store Word
STD	000111	Store Doubleword



sh %o4, [%fp + 8]

Sparc V8 Instructions

SETHI Instruction

opcode	*op*	*op2*	*operation*
SETHI	00	100	Set High-Order 22 bits

```
sethi   0x4A3, %g2
sethi   %hi(Z), %o3
```

SETHI zeroes the least significant 10 bits of "r[*rd*]", and replaces its high-order 22 bits with the value from its *imm22* field.

A SETHI instruction with *rd* = 0 and *imm22* = 0 is defined to be a NOP

Shift Instructions

opcode	*op3*	*operation*
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

```
sll     %g2, 4, %g2
sra     %o4,%g4,%g2
```

Sparc V8 Instructions

Logical Instructions

opcode	*op3*	*operation*
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

```
and     %g2, 4, %g2
xorcc   %o4,%g4,%g2
```

13-bit immediate operands are sign extended to 32 bits

Sparc V8 Instructions

Add and subtract instructions

opcode	*op3*	*operation*
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

```
addcc    %g2, 4, %g2
addxcc   %o4,%g4,%g2
```

opcode	*op3*	*operation*
SUB	000100	Subtract
SUBcc	010100	Subtract and modify icc
SUBX	001100	Subtract with Carry
SUBXcc	011100	Subtract with Carry and modify icc

```
sub    %g2, 4, %g2
subx   %o4,%g4,%g2
```

Sparc V8 Instructions

Multiply Instructions

opcode	*op3*	*operation*
UMUL	001010	Unsigned Integer Multiply
SMUL	001011	Signed Integer Multiply
UMULcc	011010	Unsigned Integer Multiply and modify icc
SMULcc	011011	Signed Integer Multiply and modify icc

```
smul    %g2, 4, %g2
umul    %o4,%g4,%g2
```

Result=Reg * sign-extended 13-bit immediate
Or = reg1 * reg2
Low part of product goes to rd
High part of product goes to Y register

Divide Instructions

opcode	*op3*	*operation*
UDIV	001110	Unsigned Integer Divide
SDIV	001111	Signed Integer Divide
UDIVcc	011110	Unsigned Integer Divide and modify icc
SDIVcc	011111	Signed Integer Divide and modify icc

```
udiv        %g2, 4, %g2
sdiv        %o4,%g4,%g2
```

Y:reg1 ÷ sign-extended 13-bit immediate
Or Y:reg1 ÷ reg2
32-bit quotient goes into rd
Remainder is discarded

wry instruction writes Y register
rdy instruction reads Y register

Sparc V8 Instructions

FPU instructions employ separate float registers

Floating point arithmetic	(fadds, fsubs, fmuls, fmuld, fdivs, fdivd, fsqt)
Conversion between float and integer	(fitos, fitod, fstoi, fdtoi)
Floating point comparison	(fcmps, fcmpd)
Load & store floating point operands in memory	(ldf, lddf, stf, stdf)
Copy between float registers	(fmovs, fnegs, fabss)

Sparc V8 Instructions

PC and NPC are used in executing and fetching instructions

1. The instruction pointed to by the PC executes while the next is fetched using NPC

2. Contents of NPC get copied into PC and NPC is updated
   NPC is incremented by 4
   For taken branches, NPC is overwritten with branch target address

Branches use PC-relative addressing

Target address = PC + 4*sign-extended(disp22)
NPC = target address
Control is transferred by copying NPC into PC

Sparc V8 Instructions

Branch Instructions

opcode	*cond*	*operation*	*icc test*
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not (Z or (N xor V))
BLE	0010	Branch on Less or Equal	Z or (N xor V)
BGE	1011	Branch on Greater or Equal	not (N xor V)
BL	0011	Branch on Less	N xor V
BGU	1100	Branch on Greater Unsigned	not (C or Z)
BLEU	0100	Branch on Less or Equal Unsigned	(C or Z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
BCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPOS	1110	Branch on Positive	not N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	not V
BVS	0111	Branch on Overflow Set	V

Conditional branching is based on condition codes

Delayed branching is used

instruction in delay slot always executes unless it is annulled

Sparc V8 Instructions

Annulment (controlled by bit29, the annul bit, in machine instruction)

Branches have a single delay slot (contains the delay instruction)

For conditional branches that are taken, the delay instruction is always executed (independent of the annul bit).

For conditional branches that are NOT taken, a=1 annuls delay instruction
(i.e., the instruction in delay slot is not executed)

E.g.: bne,a %g2,done

Unconditional branches are always taken, a=1 annuls the delay instruction
(if the a bit = 0, the delay instruction is executed for unconditional branches)

E.g.: ba,a exit

Sparc V8 Instructions

Two instructions support calling subroutines: call and jmpl

call func1 overwrites pc with the address corresponding to func1
address of call instruction is written into %o7 (link register)
target address = pc + 30-bit signed-displacement*4

return address = %o7 + 8 to get past the instruction = the delay slot

call func1 has same effect as jmpl %o2, %o7 if %o2 contains address of func1
using register as function pointer allows for dynamic addresses (e.g., jump table)

jmpl %o7+8, %g0 same as ret (return) synthetic instruction (%g0 is read-only)

Caution: if function executes a save instruction, a new register window appears
(%o7 becomes %i7)

Sparc V8 Instructions

Window Management

Subroutines and functions can use the save instruction to slide the register window
- new registers become visible (no need to save registers on stack)
- may also allocate space on stack (stack frame)

Example: save %sp, -512, %sp gets new register window and allocates 512 bytes

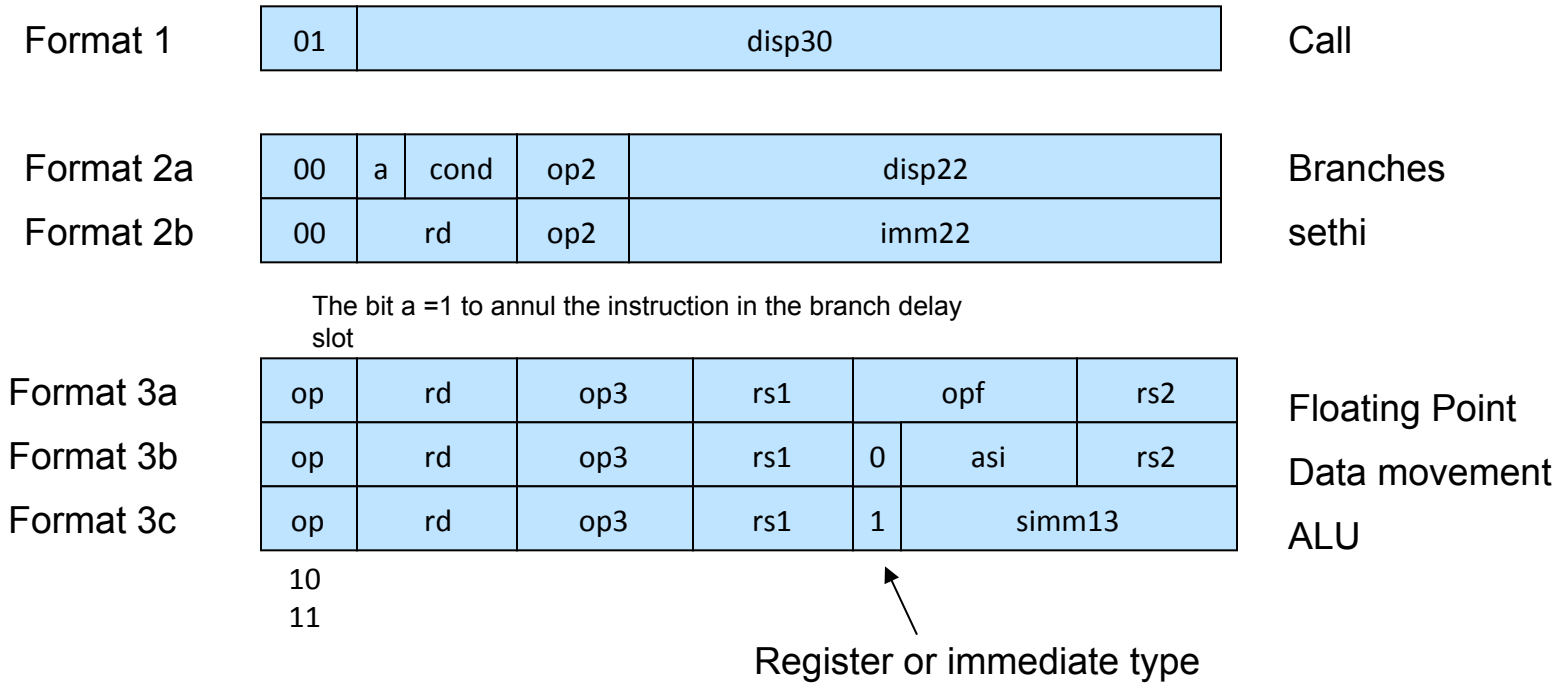
return address = %i7 + 8 to get past the instruction = the delay slot

The restore instruction slides window back to previous registers (restores %sp)

Example: ret same as jmpl %i7+8, %g0
restore executes in delay slot of the ret instruction

Sparc Machine Instructions

- There are 3 machine instruction formats



The leftmost 2 bits indicate to which of the 3 groups the instruction belongs.

Sparc Machine Instructions

The CALL instruction contains a 30-bit PC-relative displacement to the target location.

npc = (instruction<29:0> << 2) + pc generates address of function to be called

Using the pc-relative displacement instead of the direct address makes the CALL instruction position-independent.

Sparc Machine Instructions

Format 2a

Branches

Branch instructions contains a 22-bit PC-relative displacement to the target location.

Limits branch range to $\pm 2^{21}$ instructions from program counter

Op2 = 010 indicates a conditional branch

4-bit cond field indicates condition that causes branch to be taken

Annul bit (a) prevents execution of instruction in delay slot when branch is not taken

Sparc Machine Instructions

The sethi instruction has op2 = 100 & contains a 22-bit immediate value

Value is loaded into upper 22 bits of rd register and low 10 bits are cleared to 0

Can be combined with “or” instruction to fill rd with a 32-bit constant

facilitated by %hi(x) and %lo(x) assembler operators

```
sethi   %hi(x),  %o1          ; puts upper 22 bits of address x into %o1 reg.
or      %o1, %lo(x), %o1      ; merges in low 10 bits of address x
```

Sparc Machine Instructions

Arithmetic/logic instructions use one of two formats.

Format 3b	10	rd	op3	rs1	0		rs2
Format 3c	10	rd	op3	rs1	1	simm13	

ALU

Rd is destination result register

Op3 indicates the operation

1st source register is rs1

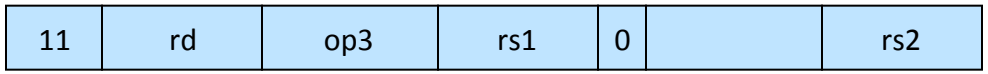
If bit13 = 0 the rs2 is the 2nd source register

If bit13 = 1 the second source operand is a 13-bit signed immediate value

Sparc Machine Instructions

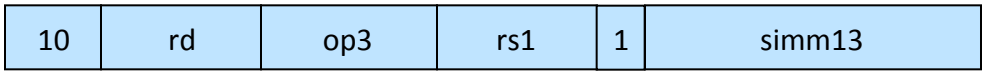
Load & store instructions access memory

Format 3b



Data movement

Format 3c



Op3 indicates type of data movement instruction (6 bits)

Loads read from memory into the rd register

Stores write contents of rd register into memory

Memory address accessed = [rs1] + [rs2] if bit13=0

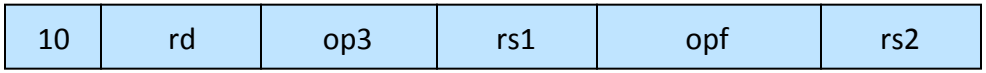
= [rs1] + simm13 if bit13 = 1 (-4096 ≤ offset ≤ 4095)

if rs2 = %g0, this is effectively register indirect addressing for the memory operand

Sparc Machine Instructions

Floating Point machine instruction format

Format 3a



Op3 = 110100 indicates Floating Point instruction

Opf indicates floating point operation

FP result register is rd, source registers are rs1 and rs2

Sparc V8 Traps

Sparc V8 Traps

Traps are the mechanism for responding to events during program execution. Such events include:

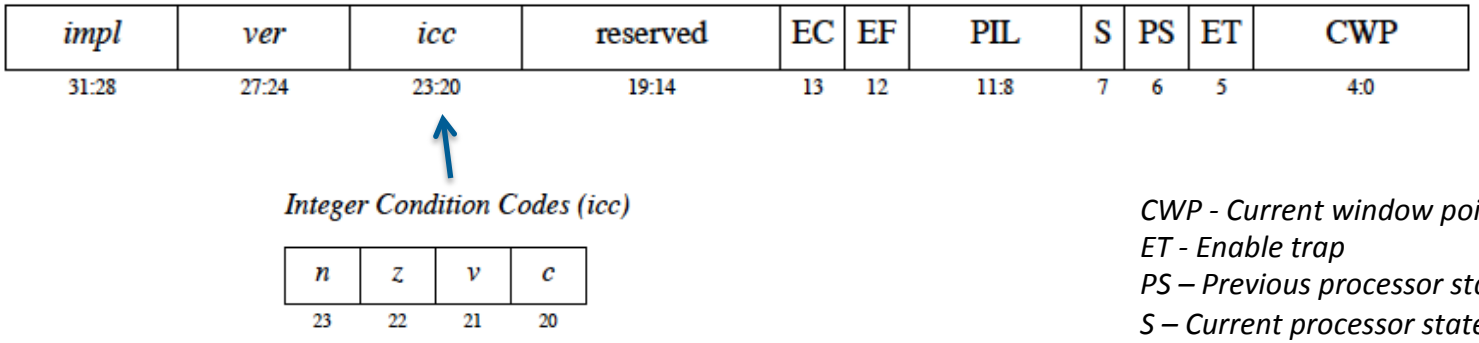
- Attempts to execute privileged or unimplemented instructions
- Exceptions such as divide by zero or overflow
- Attempted access to restricted or reserved memory areas
- Operating system service calls
- External interrupts

To understand trap handling, the processor state register must first be understood.

Sparc V8 Traps

Processor State Register

The processor state register is accessed using privileged read/write instructions (rdpsr & wrpsr).



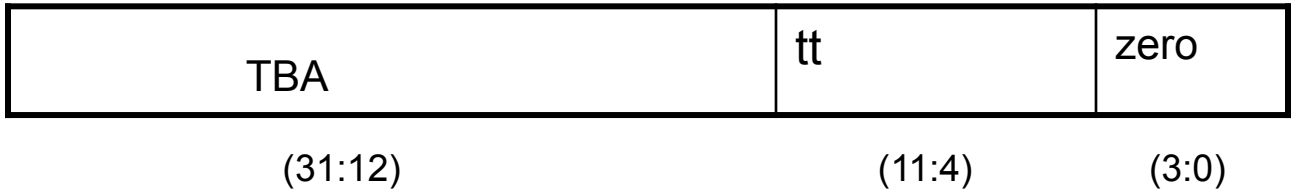
CWP - Current window pointer
ET - Enable trap
PS – Previous processor state
S – Current processor state
PIL – Processor interrupt Level
EF - Enable FPU
EC – Enable Coprocessor

Sparc V8 Traps

Sparc V8 Traps

A trap is a vectored transfer of control to supervisor code through a 256-entry trap table.

The trap base address (TBA) is held in the trap base register (TBR).



TBA = the most significant 20 bits of the trap base address

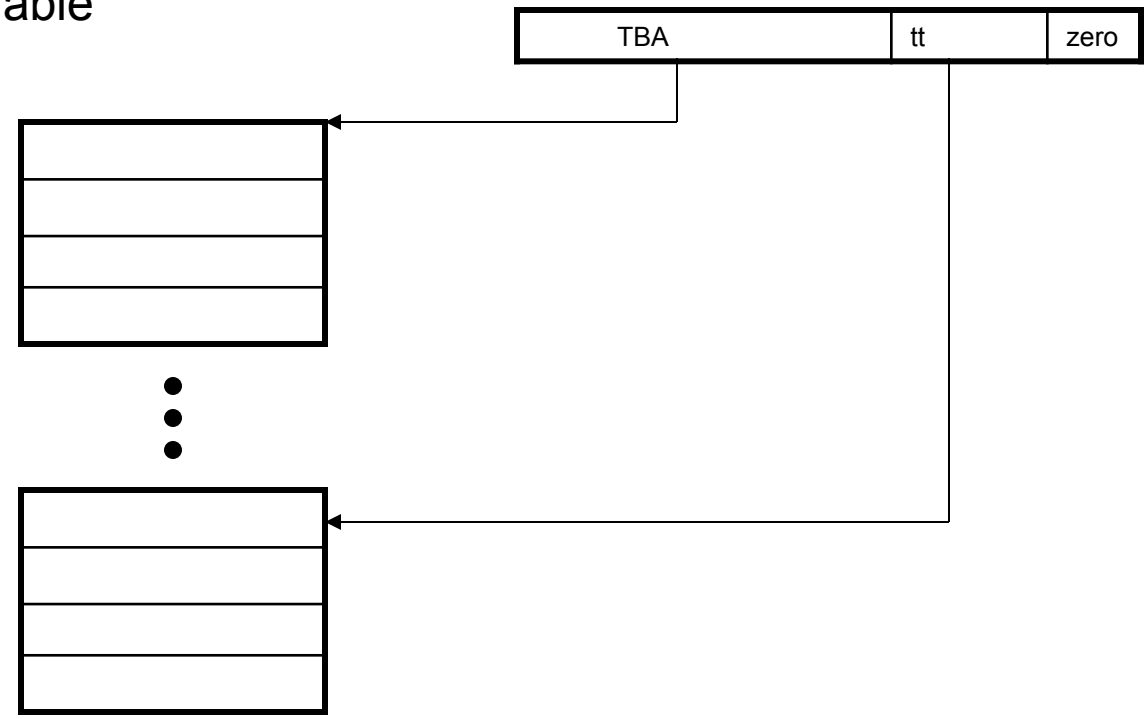
(tt) = an 8-bit number set by hardware when a trap occurs to identify the trap type

(zero) = Bits 3 through 0 are zeroes since each trap table entry contains 16 bytes (the first 4 instructions of the handler)

The TBA field within the TBR can be written by the privileged instruction WRTBR (write TBR).

Sparc V8 Traps

Trap Vector Table



Each entry is 4 words in size and contains the first four instructions of the corresponding trap handler.

Recall that the MIPS employs a single exception vector address through which all exceptions are funneled and examines the cause register to determine the action to take.

Sparc V8 Traps

Traps are like unsolicited or unexpected procedure calls

New register window is obtained when a trap occurs

pc, npc and psr are saved

copied into local registers within the new window

For traps other external reset, hardware generates a trap ID

trap ID is written into the tt field within the TBR

ET bit is cleared within PSR (preventing further traps)

Control is transferred into the trap table whose address is in the TBR

Sparc V8 Traps

Trap Table Entries

The tt field (in TBR) can specify 256 distinct types of trap

– half for hardware traps & half for software traps

ticc instruction generates software trap (for system service, etc.)

operand is software_trap#. Example: ta 12 trap always

opcode	*cond*	*operation*	*icc test*
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	not (Z or (N xor V))
TLE	0010	Trap on Less or Equal	Z or (N xor V)
TGE	1011	Trap on Greater or Equal	not (N xor V)
TL	0011	Trap on Less	N xor V
TGU	1100	Trap on Greater Unsigned	not (C or Z)
TLEU	0100	Trap on Less or Equal Unsigned	(C or Z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not C
TCS	0101	Trap on Carry Set (Less Than, Unsigned)	C
TPOS	1110	Trap on Positive	not N
TNEG	0110	Trap on Negative	N
TVC	1111	Trap on Overflow Clear	not V
TVS	0111	Trap on Overflow Set	V

Sparc V8 Traps

Trap Table Entries

Since the low 4 bits of the TBR are zero, table entries are on 16-byte boundaries

Each 4-word (16-byte) entry contains the first 4 of the trap handler's instructions

These 4 instructions can be used to call the corresponding handler

A reset will be performed if a trap occurs while the ET bit is 0.

A reset trap causes a transfer of control to address 0. (boot code)

Sparc V8 Traps

Returning from Traps (rett instruction)

The code sequence below returns to the instruction that caused the trap:

```
jmpl   %l1, %g0      Load PC with saved PC from local register 1
rett   %l2           Load NPC with the saved NPC
```

Alternative return to instruction following the one causing the trap:

```
jmpl    %l2, %g0     Load PC with saved NPC
rett    %l2 + 4      Load NPC with saved NPC+4
```

Memory mapped I/O is used and is based on interrupts or on polling