Miscellaneous Computer Architectural Features

presented by **David Morano**
class professor **Professor David Kaeli**

class notes 03/10/23

objectives

- **register windows (on SPARC)**
  - SPARC architected registers
  - implementation
  - overlapping and circular physical register arrangement
  - how they work
  - what are they good for
  - relationship to subroutine calls
  - relationship to the program stack
  - why they may not be such a good idea any longer
- **some miscelleneous items**
- **I/O architectural possibilities**
- **memory consistency model**

architected registers on SPARC

- **architected registers consist of :**
  - —32 general purpose integer registers (32-bit -- 64-bit on V9)
  - —32 floating point registers (32-bit) -- extra 8 64-bit on V9
  - —other (the 'Y' register for multiply step)
  - —integer registers are divided into 4 functional groups
    - 8 "global"
    - 8 "out"
    - 8 "local
    - 8 "in"
  - —special instructions and events rotate a "window" of the 'in', 'local', and "out" registers ('save' , 'restore', traps, 'rett')
  - —the 'global' registers do not rotate on window shifts
  - —floating point registers do not rotate

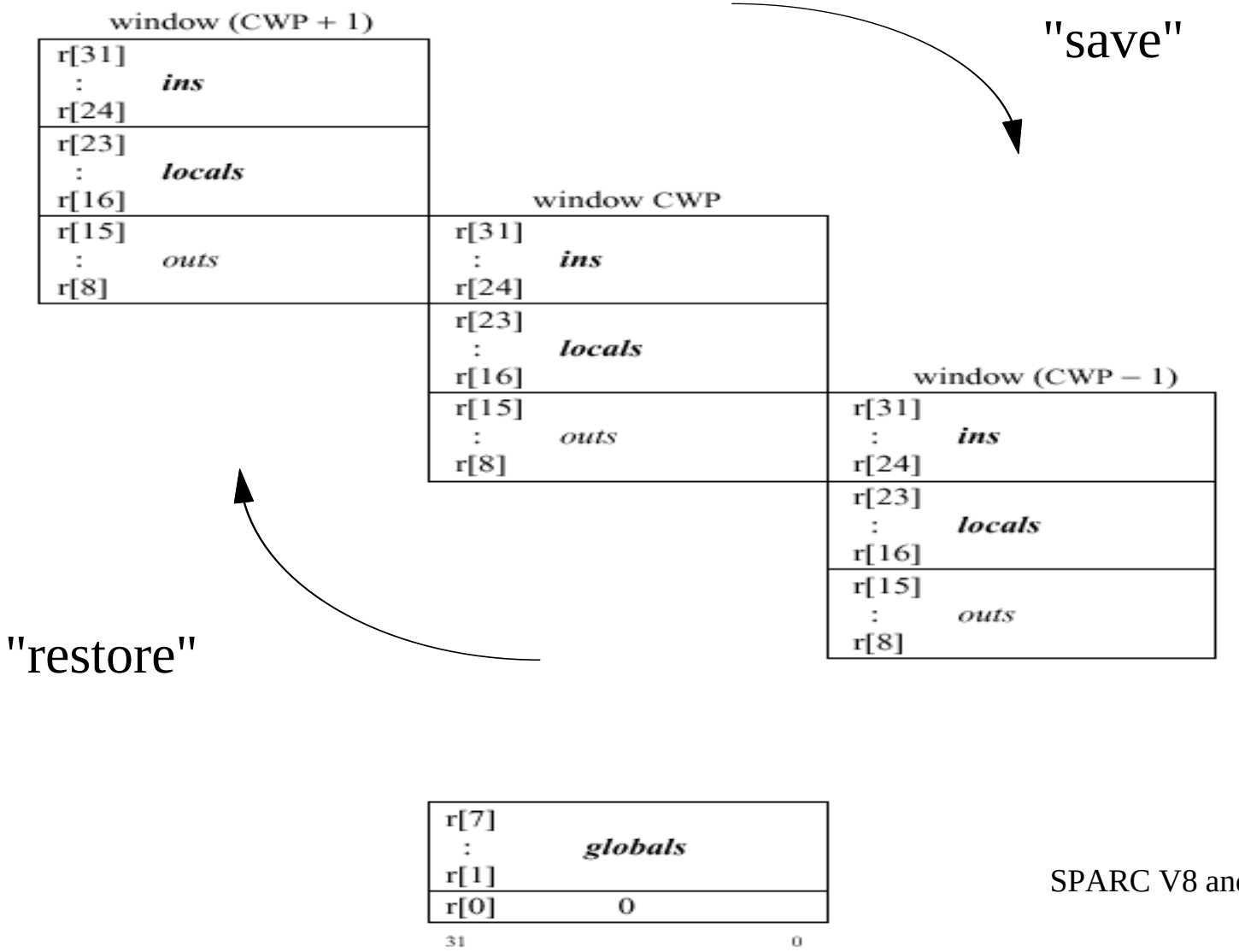
integer register use

- **global registers --** %g0 through %g7
  - —some are fixed by convention (%g0 is always '0')
- **in registers --** %i0 through %i7
  - —used to access the first 6 (%i0 - %i5) input arguments to subroutine
  - —%i0 also gets loaded with the return value from the subroutine !
  - —%i6 serves as the "frame pointer"
  - —%i7 holds the address of the calling instruction (used for returning)
- **out registers --** %o0 through %o7
  - —%o0 through %o5 can pass up to 6 parameters to a subroutine
  - —%o6 is the current "stack pointer"
  - —%o7 gets loaded with the address of the calling instruction
- **local registers --** %l0 through %l7
  - —freely usable by the subroutine
  - —play a crucial role in interrupt and trap handlers !

how are windows implemented ?

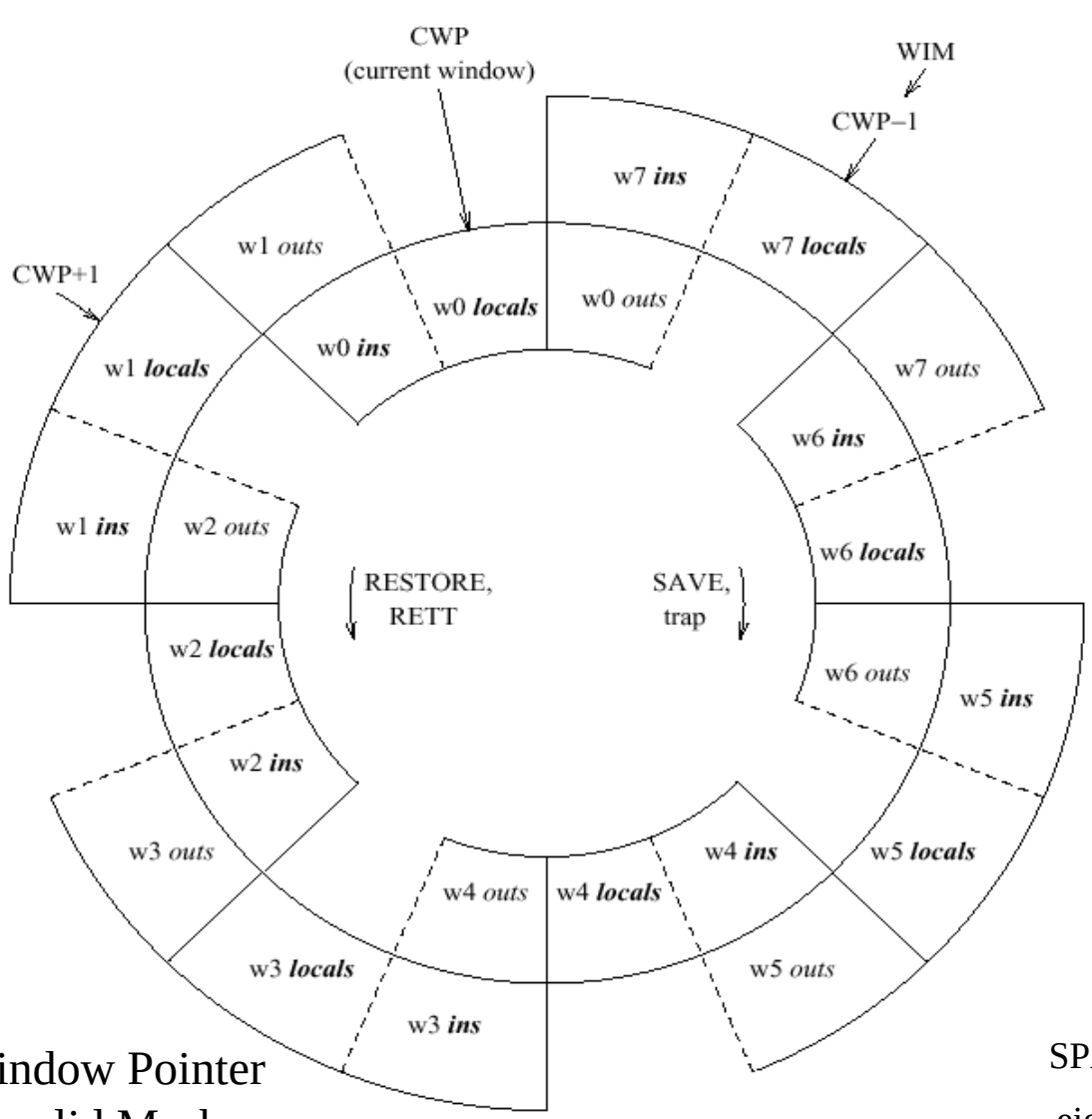
- **how many registers windows are there ?**
  - SPARC V8 and earlier allowed for 2 to 32 windows (scalable !)
  - for N windows, a machine can only use (N - 1) of them
- **there are more physical registers in the processor than "architected" registers**
  - SPARC V8 and earlier allowed for 40 to 520 physical registers to make up the windows
  - the smallest machine with 2 windows has the extra 8 registers as the "local" inside the trap handler
- **special kernel-mode registers (SPARC V8 and earlier)**
  - CWP (Current Window Pointer) -- part of the Processor Status Register (PSR) and points to the currently active window
  - WIM (Window Invalid Mask) -- used to designate which windows are valid and which are not -- trap occurs when "entering" an invalid window

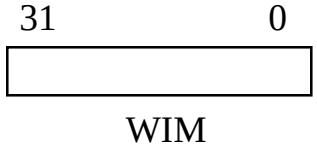
overlapping windows



SPARC V8 and earlier

circular window operation





CPW - Current Window Pointer
WIM - Window Invalid Mask

SPARC V8 and earlier

eight windows shown

windows in action

- **getting a new window**
  - execute instrction 'save' or 'trap'
  - take an exception (floating point, et cetera)
  - the CWP gets decremented (on SPARC V8) -- incremented on SPARC V9
- **what if there aren't any more valid windows to enter ?**
  - a trap is taken and the processor enters the invalid window anyway
  - since the window is "invalid" only the "local" registers are available
  - registers %l1 and %l2 will contain the PC and nPC from before
  - trap handler has to save the registers of the next window to the stack (actually a fairly complicated task in real life)
- **returning to an old window**
  - execute a "restore" or 'rett" (return from trap)
  - this shifts the window in the opposite direction and can also trap !

what are register windows good for ?

- **it's not clear anymore !!**
- **but the idea at the time was that it facilitated better subroutine call-return handling than conventional architectures due to not having to spill and restore registers to and from memory**
- **the numbers at the time (data for large C programs) :**
  - on non-RISC processors, about 50% or more instructions are load-stores
  - on non-windowed RISC processors about 20% - 40% of instructions are load-stores
  - on windowed RISC (like SPARC), only about 20% of all instructions are load-stores
  - observation shows that only about 4% of "save"s cause traps -- this gives about only 2/3 of a register saved per subroutine call
  - an average context switch requires only about half of all register windows to be saved

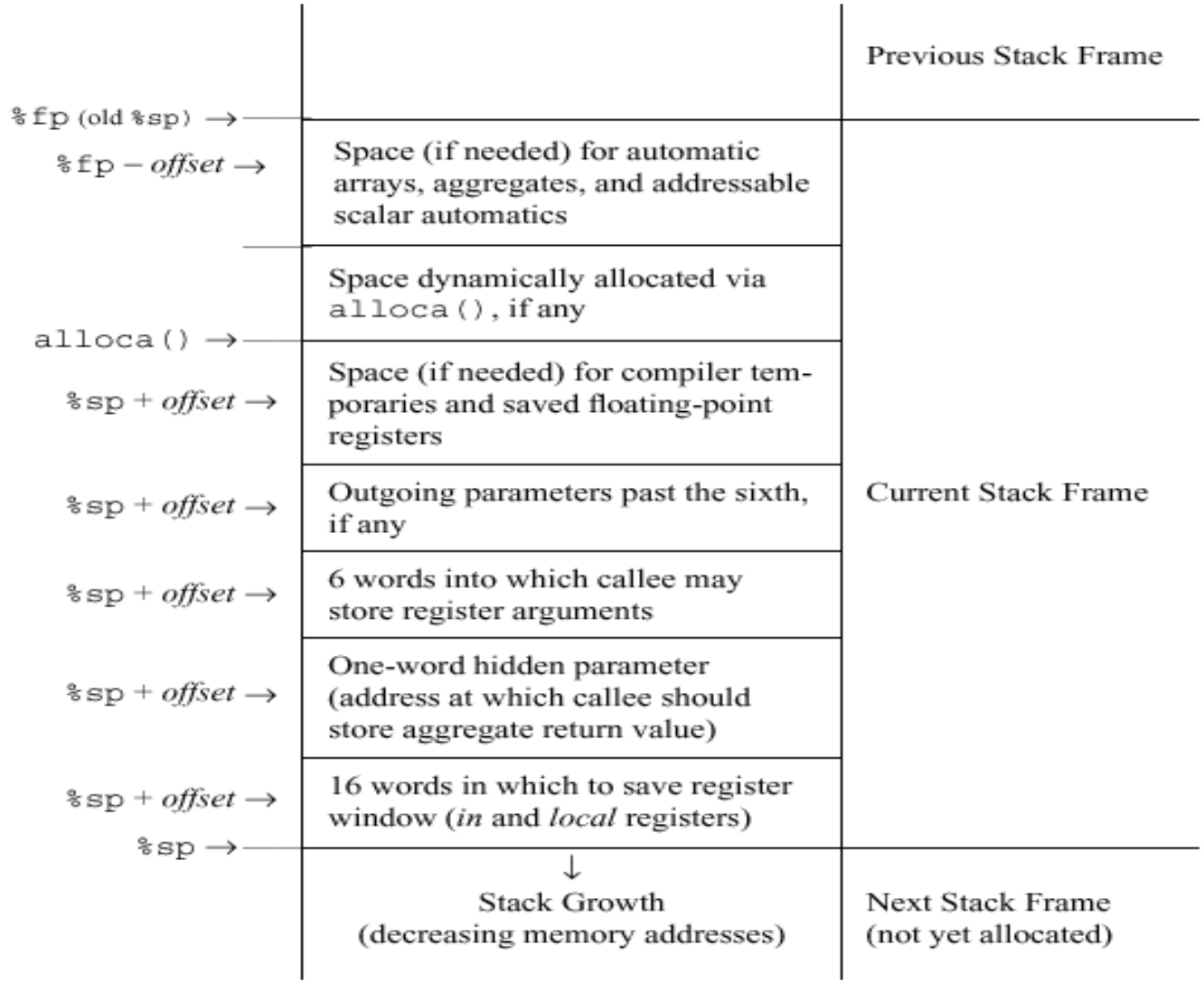
relationship to subroutine calls

- **there doesn't have to be one !**
- **a "save" instruction can be executed by a called subroutine -- the called subroutine does not have to execute a "save" (thus turning the window) if it doesn't want to !**
- **leaf subroutines provide an example where a "save" might not need to be executed**
- **if a subroutine does not do a "save" then it only has certain "out" registers (%o0 - %o5, %o7) and register %g1 to play with since all others need to be preserved**
- **if a subroutine does do a "save" -- it should generally do a "restore" also before returning to its caller !! :-)**
- **all subroutine stack frames have to have room for a possible window dump of 16 registers ("ins" and "locals")**
  - needed in case this window needs to be dumped
  - future window turns or traps can require a dump (register "spill")

relationship to stack

- **there generally needs to be close cooperation between how the register windows work and how the stack is managed**
- **the top of the stack always needs to have 16 words available for the possible dumping of the "in" and "local" registers of the associated stack frame**
- **although up to 6 subroutine arguments can be passed through registers -- "outs" become "ins" to the called subroutine -- space is always reserved on the stack for these parameters as well !!**
  - this is needed in those cases where the subroutine wants to reference the passed arguments as memory locations
  - the compiler can store the arguments there also even if not needed to be explicitly addressed
- **having to require the top of the stack to always be available for register dumps is a little weird ! :-) ; the compiler must make stack allocations accordingly**

typical SPARC stack frame (by convention)



example subroutine code

C language:

```
int sub(int a, int b)
{
      return (a + b) ;
}
```

remember SPARC has delay slots !

non-leaf assembly:

```
sub:
      save      %sp,-96,%sp
      ret
      restore %i0,%i1,%o0
```

saves and restores a register window

regular (window) type return

leaf assembly:

```
sub:
      retl
      add %o0,%o1,%o0
```

doesn't create a new window
uses special "leaf" type return

why windows are no longer so good (1)

- **it was assumed that there were few context switches as compared with subroutine call-returns at the time -- no longer as true**
  - —generally all registers in all windows need to be saved and restored during context switches -- this is far more stuff to be saved than on conventional processors (without register windows)
- **the cost of saving and restoring to memory was thought to dominate all other considerations -- no longer as true**
  - —this is no longer as true since L1 cache access has generally closely followed processor speeds
  - —subroutine save-restores to memory benefit with fast L1 cache while context switches generally do not benefit as much
- **register windows best benefit performance when call chains are shallow -- deep call chains can eliminate the most of the gains of having the register windows in the first place**

why windows are no longer so good (2)

- **when a subroutine does not need either 6 arguments or the 8 local registers provided by the register window, this amounts to a waste of processor resources that could have been more effectively used for some different purpose**
- **register windows are difficult to implement on fast dynamically scheduled processors and tend to hinder performance**
  - —complicates any architected to physical renaming register mapping
  - —due to remapping logic it adds propagation delay in the critical path for addressing registers
- **the OS software to manage register windows is remarkably complex despite appearances -- complaints from open-source OS developers ! :-)**
- **although in theory the SPARC register windows do not have to be used for call chaining, attempts to go to a flat register model while remaining binary compatible with existing software has not been very successful**

processor modes

- **the processor can support more than one instruction set at a time !**
  - — VAX 11-780 (VAX native and PDP-11)
  - — ARM-Thumb (ARM native and "Thumb")
- **the processor can be in one of several priviledged modes**
  - — examples: "kernel" (or "supervisor" , "master"), "executive", "user"
  - — interrupts, exceptions, or special instructions (traps) move to a more priviledged mode
  - — special return instructions return to the previous mode
- **the processor can have special architected stack pointers**
  - — "user" , "kernel" , "interrupt"
  - — provides a safe stack when entering the kernel, et cetera
- **these processor modes, along with other possible processor state, can make up additional "address spaces" in the machine**
  - — "kernel instrcution" , "kernel data" , "user instruction" , "user data" , other

alternate address space

- **MMU mapping makes up the current address space**
- **modes of the processor along with other state (like whether it is an instruction fetch or a data reference) can make up an "alternate address space"**
  - — some "alternate spaces" are predeterined by the processor vendor
  - — some can be created by the system designer
- **special instructions can be provided to allow code executing in one address space to access memory locations in another**
  - — "in", "out" , "movaas" (move alternate address space), others
- **of course the MMU can also be programmed (by setting up the proper page table entries) to map a page from an alternate address space into the current address space** (but this might be tiresome)
- **these facilities are used by the OS to access and manage :**
  - — data from a user address space like system call arguments or user data buffers while exeuting in the kernel space
  - — I/O memory or I/O control and data registers

interrupts, traps, and exceptions

- **trap -- generally an instruction that causes a "mode" change in the processor**
  - — often used to implement system calls or special instructions
  - — occurs "synchronously" with program execution
- **interrupt -- generally an external event that causes a pseudo-random control flow change in the processor**
  - — generally causes a switch to a priviledged mode of the processor
  - — can also be software initiated for OS convenience reasons
  - — occurs "asynchronously" with program execution !
- **exception -- unexpected (?) change of control flow by executing an instruction that went "wrong" somehow**
  - — divide by zero
  - — page fault
  - — floating point overflow or underflow if enabled
  - — register window overflow or underflow
  - — tons of others !
  - — similar handling as above -- occurs "synchronously"

handling I/O from the processor

- **several ways to implement I/O and how it appears architecturally to an executing program**
- **programmed I/O**
  - — using special I/O or address-space instructions
  - — using memory mapping
- **direct memory access (DMA)**
  - — using an I/O intermediary
  - — directly from the peripheral
- **I/O completion or readiness is usually signalled by an "interrupt" to the processor**
  - — vectored interrupt -- I/O device identifies an interrupt service routine
  - — polling interrupt -- I/O device interrupts but does not identify a particular interrupt service routine to execute (the interrupting device must be found by a poll of all possible devices that can cause that interrupt)
  - — not interrupting on completion at all is bad form and should normally be avoided !

programmed I/O

- **special I/O instructions**
  - — "in" and "out" -- Intel 8080, x86, IBM 370, others
  - — these are part of a class of instructions more generally known as special "address-space" or "alternate-space" instructions
  - — these instructions access what looks like a form of memory but in a separate address space than that of the executing program
  - — these instructions can be "priviledged"
  - — can provide an automatic memory barrier for proper I/O access ordering
  - — can generally access the peripheral from any existing address space
- **memory mapped I/O**
  - — I/O registers simply appear to be memory locations like other memory
  - — references to these I/O registers do NOT need to use special instructions !
  - — can be memory mapped and priviledged (via MMU) to anywhere desired
  - — can be made accessible to user (or kernel) programs (like for graphics frame buffers -- very convenient !)
- **many processors now implement both types simultaneously for OS convenience reasons**

direct memory access (DMA) I/O

- **often used for large data or high bandwidth data sources or sinks**
  - — disks
  - — tapes
  - — high speed serial comm (like Ethernet)
- **also good for periodic data that comes in small increments**
  - — many serial comm I/O
  - — terminal output (even relatively low speed)
- **relieves the processor of otherwise burdensome and monotenous data moving tasks**
  - — processor can do other work while I/O data movement is in progress
- **the processor can program the device directly to carry out future DMA transfers or can program an intermediary device to do the transfers on behalf of the I/O device**
  - — programming a DMA device uses programmed I/O itself

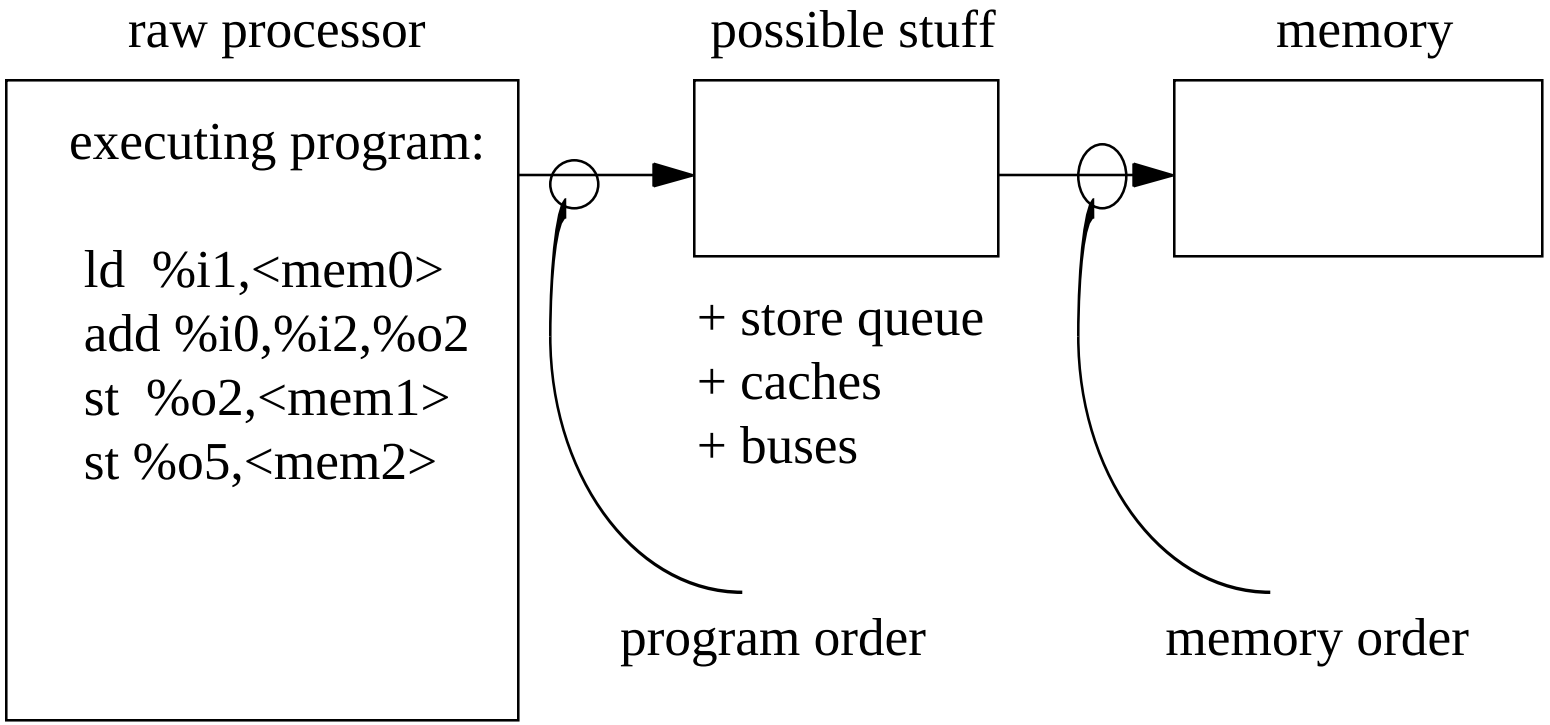
memory consistency (ordering) models (1)

- **attribute of the ISA of a processor**
- **some processors can support more than one model**
- **is a contract with the programmer about how memory references from an executing program will appear to memory or to other interested entities (other processors, I/O devices)**
- **many types are out there :**
  - — "strong" or "serial" -- all memory reads and writes appear in the same order to other entities (example SGI machines)
    - some vendors include instruction reads, others may not
  - — "weak" , "partial" , "relaxed" (not necessarily the same semantics)
    - instruction reads are not serially consistent with data accesses
    - reads can slip past writes -- very common (example "Total Store Order" TSO on SPARC)
    - writes can slip past each other (example "Partial Store Order" PSO on SPARC)
  - — there are a plethera of combinations and possibilities
  - — other names and other semantics

memory consistency (ordering) models (2)

- **what causes non-serial memory ordering ?**
  - — processor store buffers (having FIFO and non-FIFO store semantics)
  - — store buffers allow for better memory operational performance
  - — NOT enforcing FIFO ordering can provide even higher performance
- **all memory references always "appear" strongly consistent (serially consistent) with the program causing the references (required for proper program execution !)**
- **why is it important to know the memory ordering model ?**
  - — implementing locking semaphores with other processors (or I/O processors)
  - — properly reading and writing to I/O control and data registers
  - — note that multi-processor synchronization algorithms like Dekker's algorithm CANNOT be used on processors that do not provide for strong (serial) memory ordering -- this is because Dekker's algorithm (and others like it) REQUIRE strong memory ordering to function properly
  - — for general purpose computers, as far as I know only SGI still provides a strong serially consistent memory model

memory consistency (ordering) models (2a)



memory consistency (ordering) models (3)

- **special instructions can be used to temporarily force a deterministic memory ordering**
  - — I/O instructions ("in" , "out" , et cetera)
  - — "memory store barrier" -- force serial ordering of stores
  - — "memory barrier" -- force serial ordering of data loads and stores
  - — "flush" -- force serial ordering of data and instruction accesses
  - — others, different vendors may use a variety of names
- **page table entries (using the MMU) can be marked to force certain memory ordering for accesses to that page**
- **hardware can provide a signal to the processor to force a certain type of deterministic memory ordering**
  - — whole address ranges (like all or part of the I/O address range) can be so decoded to provide such a signal
  - — individual I/O devices may be able to provide a signal back to the processor for memory ordering (not common)
- **the processor can have a "mode" bit that determines the semantic**
  - — SPARC V9 (TSO or PSO) on a per-process basis !

memory consistency (ordering) models (4)

- **interactions with other processors using locking are often supplied by the vendor for user use so that the user doesn't mess up ! (which is fairly common)**
  - — it is a pain and non-portable for the user to totally understand the semantics of each machine architecture
  - — the vendor can write locking primatives once for all to use safely ! :-)
  - — high-level languages essentially provide no memory ordering guarantees
- **in like maner, reading and writing to I/O registers is often also supplied as primatives by the vendor**
  - — primatives are generally supplied as subroutines that are not available for inlining or other optimizations by a compiler
  - — they are written in assembly language and use the special memory barrier instructions to enforce required memory ordering behavior

some key points

— register windows were conceived as an idea to reduce the number of memory reads and writes that programs need to do -- this was thought valuable since memory reads and writes are a time-expensive operation

— register windows were thought to be a benefit since the ratio of subroutine call-returns to context switches was originally fairly high (more of a batch oriented processing assumption)

— SPARC implemented a form of register windows -- maybe the only commercial processor ever to do so !

— register windows on the SPARC are not required as part of subroutine handling but are very useful for that -- few other uses ever became popular

— generally only good for shallow subroutine calling chains -- but SPARC was made "scalable" so that newer processors can accomodate more register windows while maintaining full binary compatibility

— registers are "spilled" and "restored" to-from the windows by an OS exception handler (called "overflow" and "underflow")

— the program stack is specially laid out to work with the idea of register windows

review questions (1)

- **register windows**
  - — what were some of the motivations for designing a processor with register windows ?
  - — what is the relationship between register window usage and subroutine call-returns ?
  - — what happens when all of the windows are used and the program wants to make another subroutine call ?
  - — what happens when the program returns from subroutines up the call chain all the way back to the first valid register window present -- and then wants to return up the call chain again ?!
  - — how is a register window marked as being "invalid" in the SPARC V8 (and earlier) processors ?
  - — how might dynamic scheduling of instructions and out-of-order execution increase design complexity for processors with register windows ?
  - — how many used register windows for a given process need to be saved during a process context switch ? (trick question -- generally all of them !)

review questions (2)

- **performing I/O operations**
  - — what are two basic ways that I/O can be performed ?
    - – programmed I/O
    - – direct memory access (DMA) I/O
  - — what are different ways that programmed I/O might be performed ?
    - – using special I/O or alternate address space insutrcitons
    - – just mapping the I/O and using regular read-write instructions
  - — what is a memory consistency model and what problems might one cause for doing I/O operations ?
  - — how does a "strong" (also "serial") consistency memory ordering model work ?
    - – all loads, stores, and maybe instruction loads are strictly serially ordered
  - — why are non-serially consistent memory ordering models even used ?
  - — what is Total Store Order (TSO) consistency on the SPARC ?
    - – all stores, atomic load-stores, and flushes are strictly ordered (but not loads)
  - — what is Partial Store Order (PSO) on the SPARC ?
    - – stores can slip past each other (along with loads)