SPARC
and Register Windows

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

class notes 04/09/17

overview

- **SPARC architected registers**
- **implementation**
- **overlapping and circular physical register arrangement**
- **how they work**
- **why windows ?**
- **relationship to subroutine calls**
- **relationship to the program stack**
- **are windows still as good as originally assumed ?**

SPARC background

- **Sun Microsystems tried to follow the Berkeley RISC ideas**
  - 1987
- **Berkeley RISC work**
  - Berkeley 1 & 2
  - Patterson was consultant on SPARC
  - SPARC separated register windows from subroutine calls

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 (scientific code impact ?)

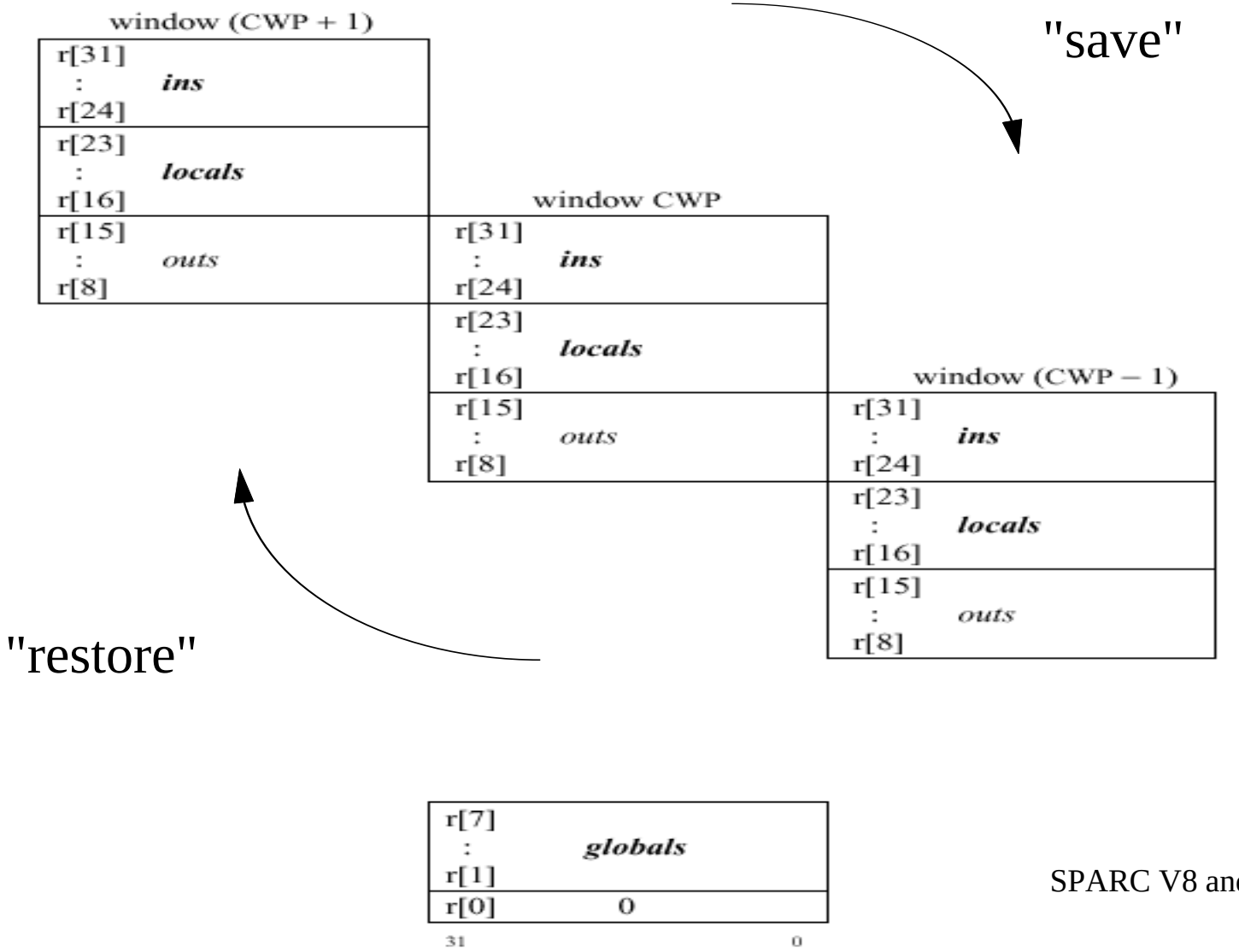
integer register use

- **global registers --** %g0 through %g7
  - —some are fixed by convention (%g0 is always '0' in HW)
- **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 (HW)
- **local registers --** %l0 through %l7
  - —freely usable by the subroutine
  - —%l1 and %l2 are loaded with PC and nPC on traps (HW)

how are windows implemented ?

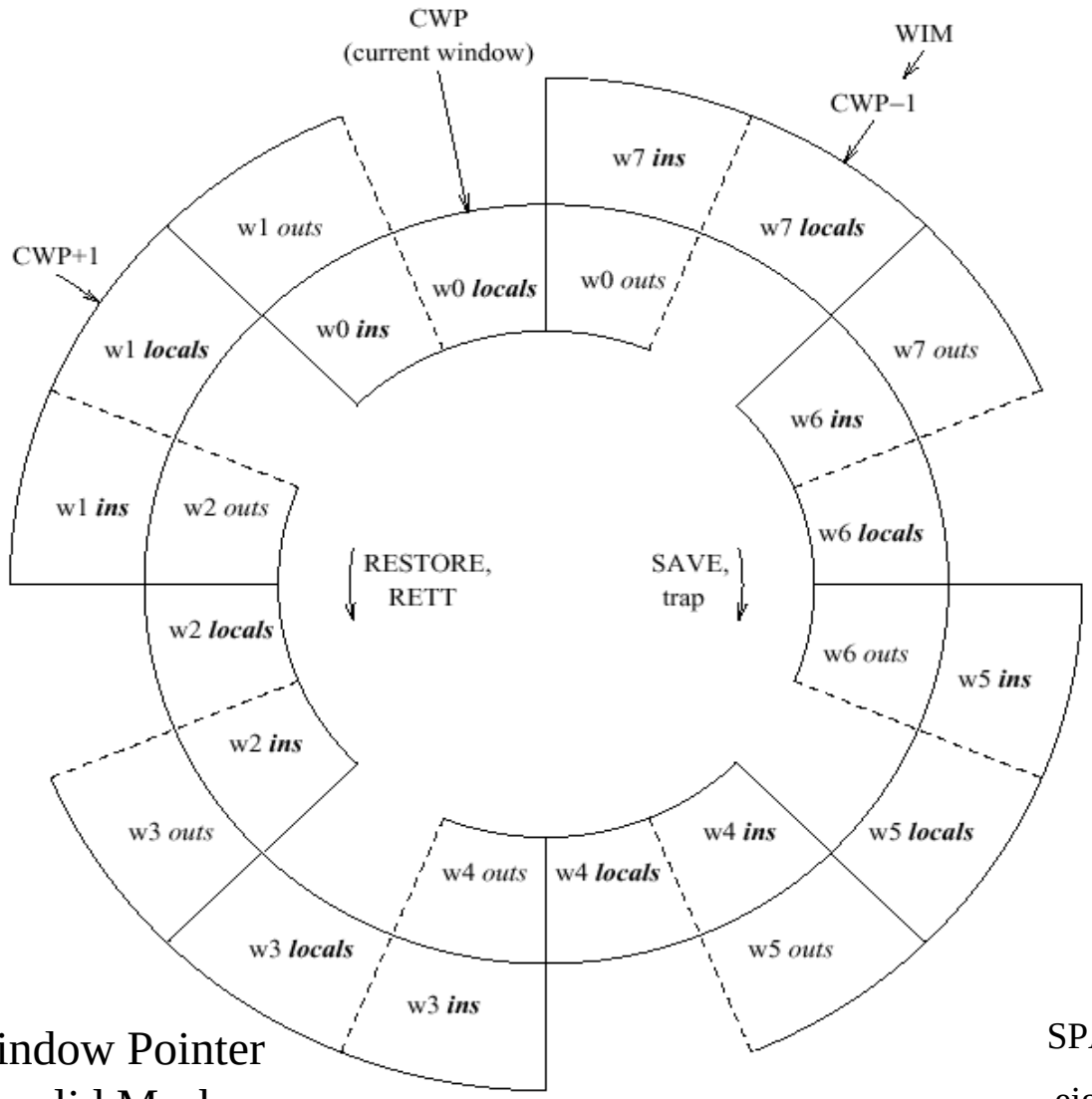
- **how many registers windows are there ?**
  - SPARC V8 and earlier allowed for 2 to 32 windows (scalable !)
  - for N windows, a machine can only use (N - 1) of them
- **there are more physical registers in the processor than "architected" registers**
  - SPARC V8 and earlier allowed for 40 to 520 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

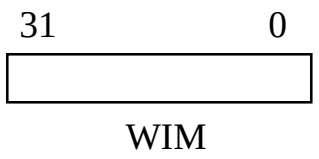
overlapping windows



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 (HW)
  - 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 !

why register windows ?

- **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 (only used windows need be saved)

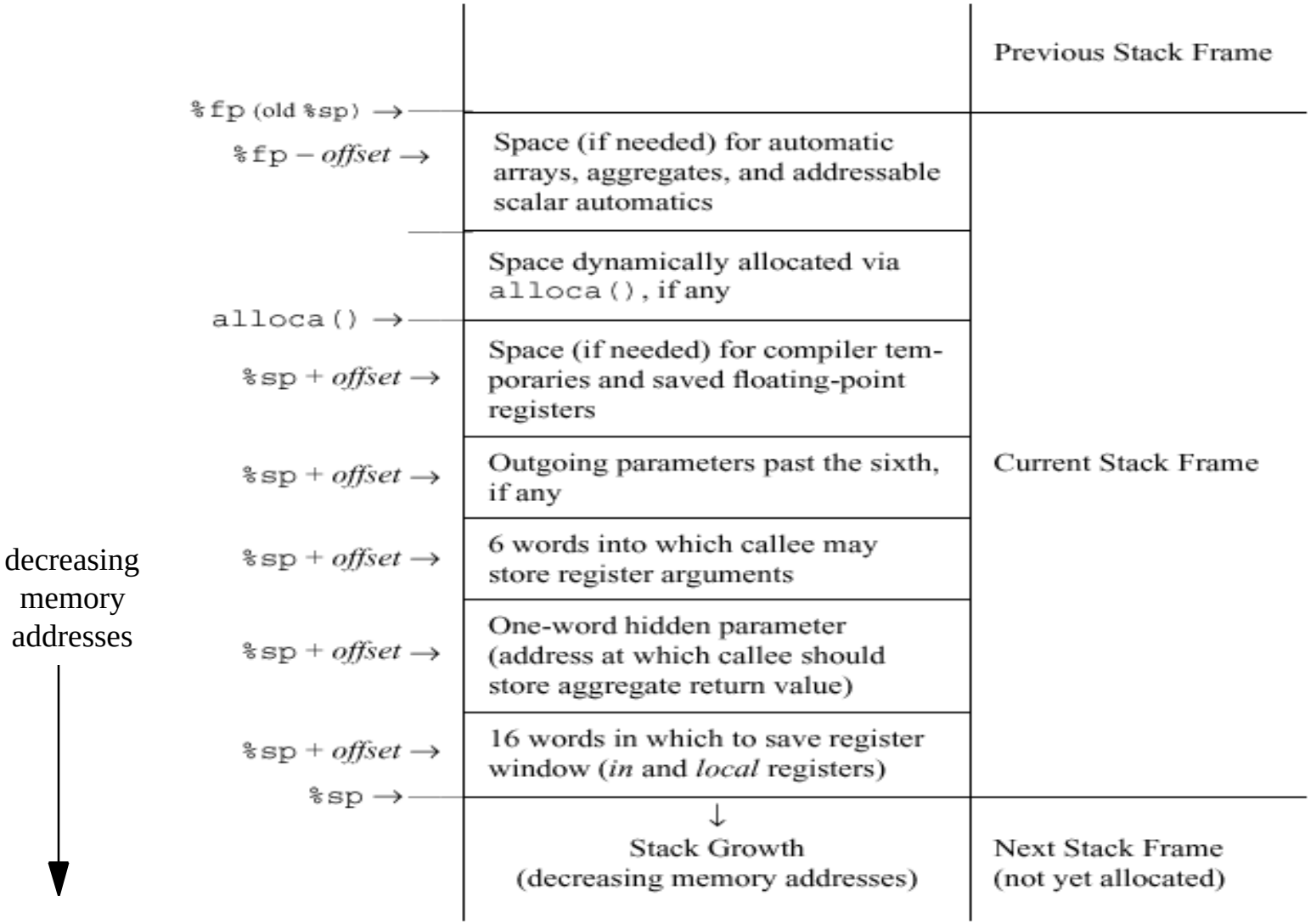
relationship to subroutine calls

- **there doesn't have to be one !**
- **a "save" instruction can be executed by a called subroutine -- but it 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

window tradeoffs

- **basic problem is that the hardware went to a lot of trouble to provide a ton of registers (ro reduce memory references ?) and the software only uses a few of them at a time**
  - — the question of the utility of available registers
  - — avoiding register spilling to memory
- **different code behavior makes better or worse use of windows**
  - — code without a lot of subroutine calls suffers greatly in terms of the potential utility of hardware registers
  - — scientific code generally suffers due to lack of register temporaries (floating point registers are not windowed on SPARC)
  - — relatively frequent subroutine calls make better use of windowed registers
  - — but deep call chains defeat register utility by requiring register spilling to memory
  - — some feel that C++ performs better due to more frequent subroutine calls -- but deeper call chains offset this possible register utility

are windows still 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**

are windows still 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**

summary

- 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

- 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 -- all of them !)