Chapter 13
Reduced Instruction Set Computers

2020.6
Howon Kim

- 정보보호 및 지능형 IoT연구실 - http://infosec.pusan.ac.kr
- 부산대 지능형융합보안대학원 - http://aisec.pusan.ac.kr

Major Advances in Computers(1)

- The family concept
  - Different implementations with the same architecture → various performance & cost
  - e.g. IBM System/360 1964, DEC PDP-8
- Microprogrammed control unit
  - Idea by Wilkes 1951
  - An alternative to a hardwired control unit is a microprogrammed control unit, in which the logic of the control unit is specified by a microprogram.
  - A microprogram consists of a sequence of instructions in a microprogramming language. These are very simple instructions that specify micro-operations
- Cache memory
  - IBM S/360 model 85 1969

Major Advances in Computers(2)

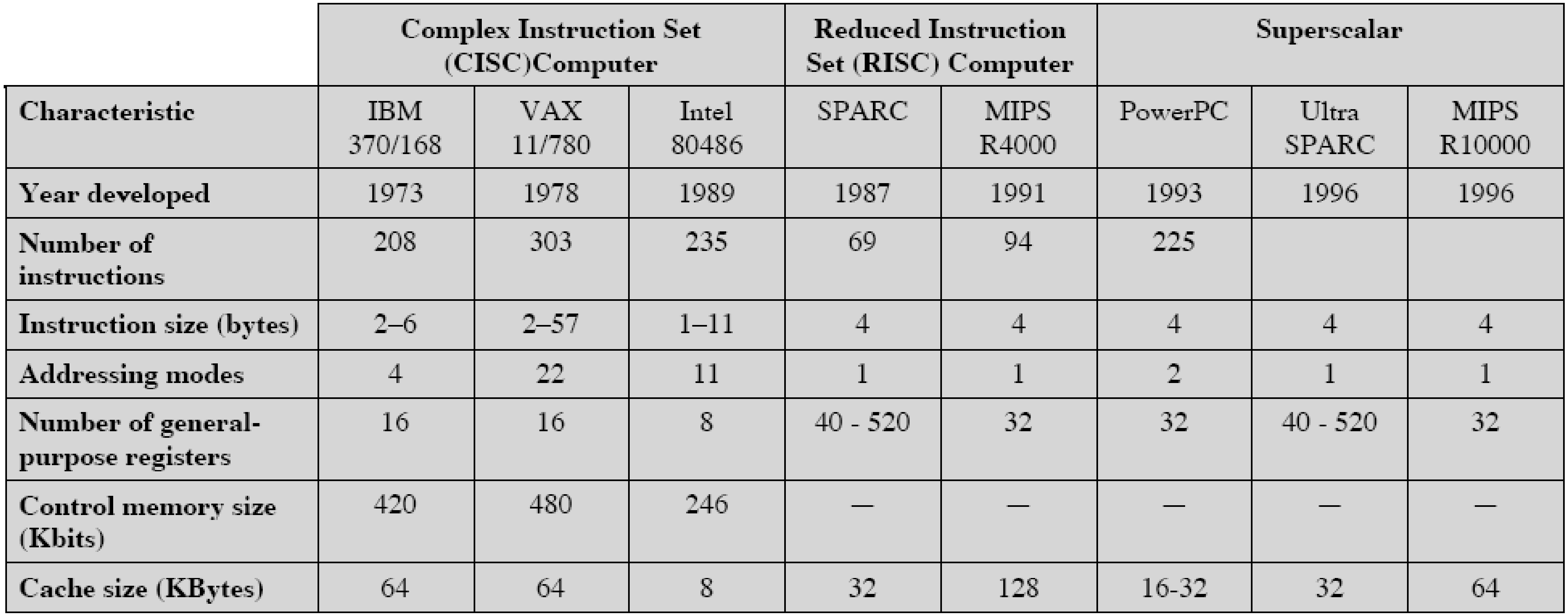
- Solid State RAM
- Microprocessors
  - Intel 4004 1971
- Pipelining
  - Introduces parallelism into fetch execute cycle
- Multiple processors

The Next Step - RISC

- Reduced Instruction Set Computer

- Key features
  - Large number of general purpose registers
  - or use of compiler technology to optimize register use
  - Limited and simple instruction set
  - Emphasis on optimizing the instruction pipeline

Comparison of processors



	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Driving force for CISC

- Software costs far exceed hardware costs
- Increasingly complex high level languages
- Semantic gap
- Leads to:
  - —Large instruction sets
  - —More addressing modes
  - —Hardware implementations of HLL statements

Intention of CISC

- Easy compiler writing
- Improve execution efficiency
  - —Complex operations in microcode
- Support more complex HLLs

Execution Characteristics

- Developments of RISCs were based on the study of instruction execution characteristics
  - Operations performed:

    determine functions to be performed and interaction with memory
  - Operands Used (types and frequencies):

    determine memory organization and addressing modes
  - Execution sequencing:

    determines the control and pipeline organization
- Studies have been done based on programs written in HLLs
- *Dynamic* studies are measured during the execution of the program

Operations

- Assignments
  - Movement of data
- Conditional statements (IF, LOOP)
  - Sequence control
- Procedure call-return is very time consuming
- Some HLL instruction lead to many machine code operations

Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

Shows simple movement is important

Shows sequence control mechanism is important

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

Weighted Relative Dynamic Frequency of HLL Operations [PATT82a]

Dynamic occurrence X # of instructions (machine codes)

Relative frequency of occurrence

It shows the actual time spent executing the various statement types

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%				
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

Procedure call/return is the most time-consuming operation !

Operands

- Mainly local scalar variables
- Optimization should be concentrated on accessing local variables

	Pascal	**C**	**Average**
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	**55%**
Array/Structure	26%	24%	25%

Procedure Calls

- Very time consuming
- Depends on number of parameters passed
- Depends on level of nesting
- Most programs do not do a lot of calls followed by lots of returns
- Most variables are local
- (c.f. locality of reference)
- Tanenbaum’s study
  - 98% of calls pass fewer than 6 arguments
  - 92% use fewer than 6 local scalar variables

Implications

- Making instruction set architecture close to HLL $\Rightarrow$ not most effective
- Best support is given by optimizing most used and most time consuming features

- Characteristics of RISC architecture
  - Large number of registers
    - Operand referencing
  - Careful design of pipelines
    - Branch prediction etc.
  - Simplified (reduced) instruction set

Approaches

- Hardware solution
  - Have more registers
    - Thus more variables will be in registers
    - e.g., Berkeley RISC, SUN SPARC
- Software solution
  - Require compiler to allocate registers
  - Allocate based on most used variables in a given time
  - Require sophisticated program analysis
  - e.g., Stanford MIPS

Use of Large Register File

- From the analysis
  - —Large number of assignment statements
  - —Most accesses to local scalars
  - $\Rightarrow$ Heavy reliance on register storage
  - $\Rightarrow$ Minimizing memory access

Registers for Local Variables

- Store local scalar variables in registers
  - Reduces memory access
- Every procedure (function) call changes locality
- Parameters must be passed
- Results must be returned
- Variables from calling programs must be restored
- Solution: register windows

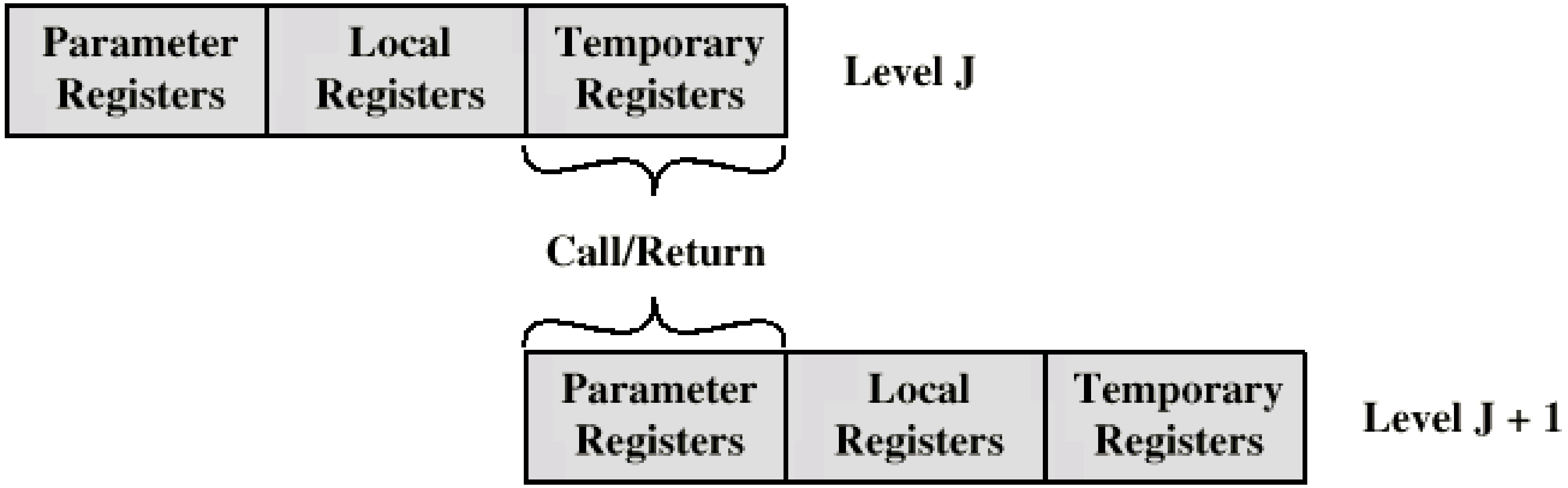
Register Windows (1)

- Register windows:

  Organization of registers to realize the goal
- From the analysis
  - Only few parameters
  - Limited range of depth of call
  - Use multiple small sets of registers
  - Calls switch to a different set of registers
  - Returns switch back to a previously used set of registers

Register Windows (2)

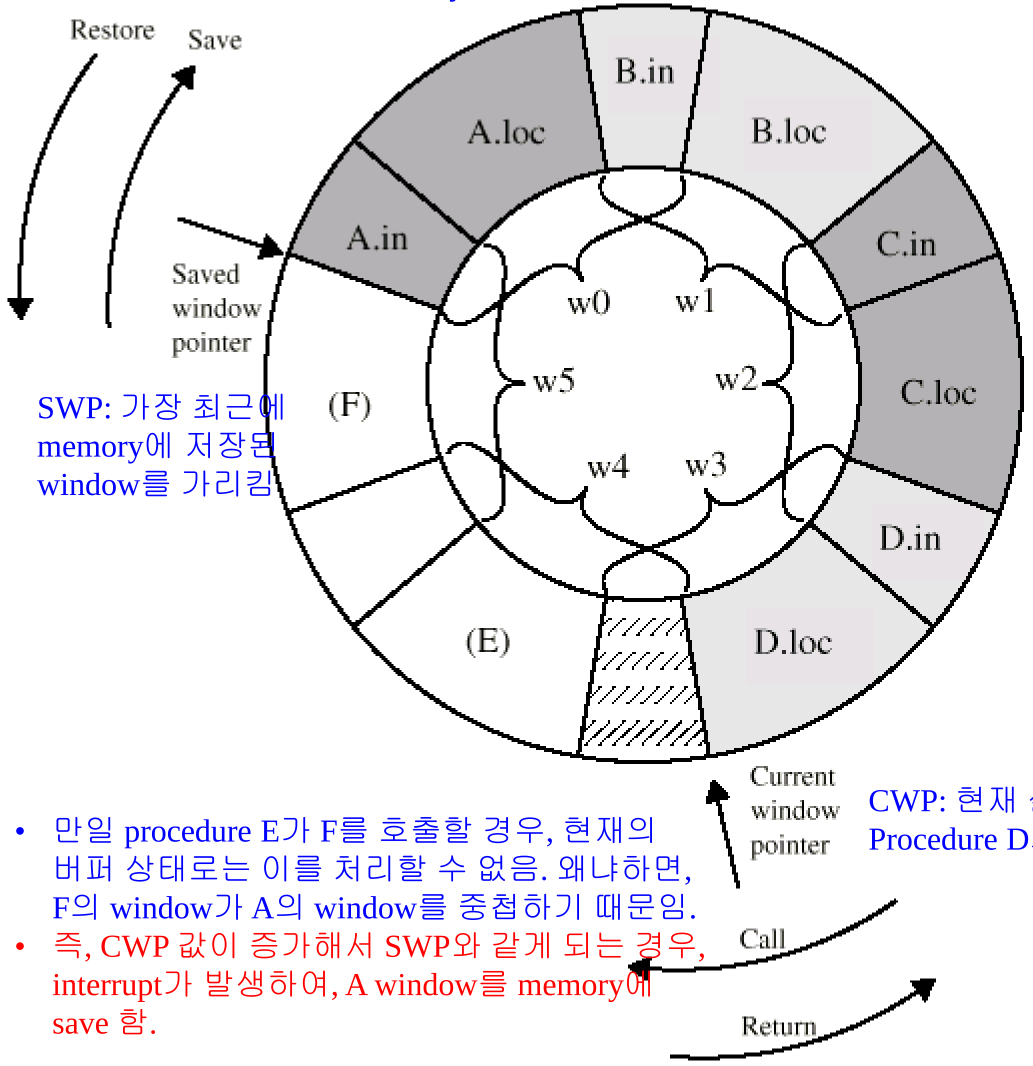
- Three areas within a register set
  - Parameter registers
  - Local registers
  - Temporary registers
  - Temporary registers from one set overlap parameter registers from the next
  - This allows parameter passing without moving data



Circular Buffer diagram

Memory로부터 Buffer로 restore함

Buffer에서 memory로 save함



- 6개의 window를 가지는 circular buffer임
- 이 버퍼는 depth 4까지 채워져 있음 (A는 B를 호출, B는 C를 호출, C는 D를 호출, procedure D는 실행중)

SWP: 가장 최근에 memory에 저장된 window를 가리킴

- 만일 procedure D가 E를 호출할 경우, E에 사용되는 argument는 D의 temporary register 영역(w3과 w4의 중첩부분)에 저장됨 → 그 후 CWP 값은 one window 만큼 증가됨

CWP: 현재 실행중인 Procedure D의 window를 가리킴

- 만일 procedure E가 F를 호출할 경우, 현재의 버퍼 상태로는 이를 처리할 수 없음. 왜냐하면, F의 window가 A의 window를 중첩하기 때문임.
- 즉, CWP 값이 증가해서 SWP와 같게 되는 경우, interrupt가 발생하여, A window를 memory에 save 함.

Operation of Circular Buffer

- When a call is made, a current window pointer is moved to show the currently active register window
- If all windows are in use, an interrupt is generated and the oldest window (the one furthest back in the call nesting) is saved to memory
- A saved window pointer indicates where the next saved windows should restore to
  - —It identifies the window most recently saved in memory.

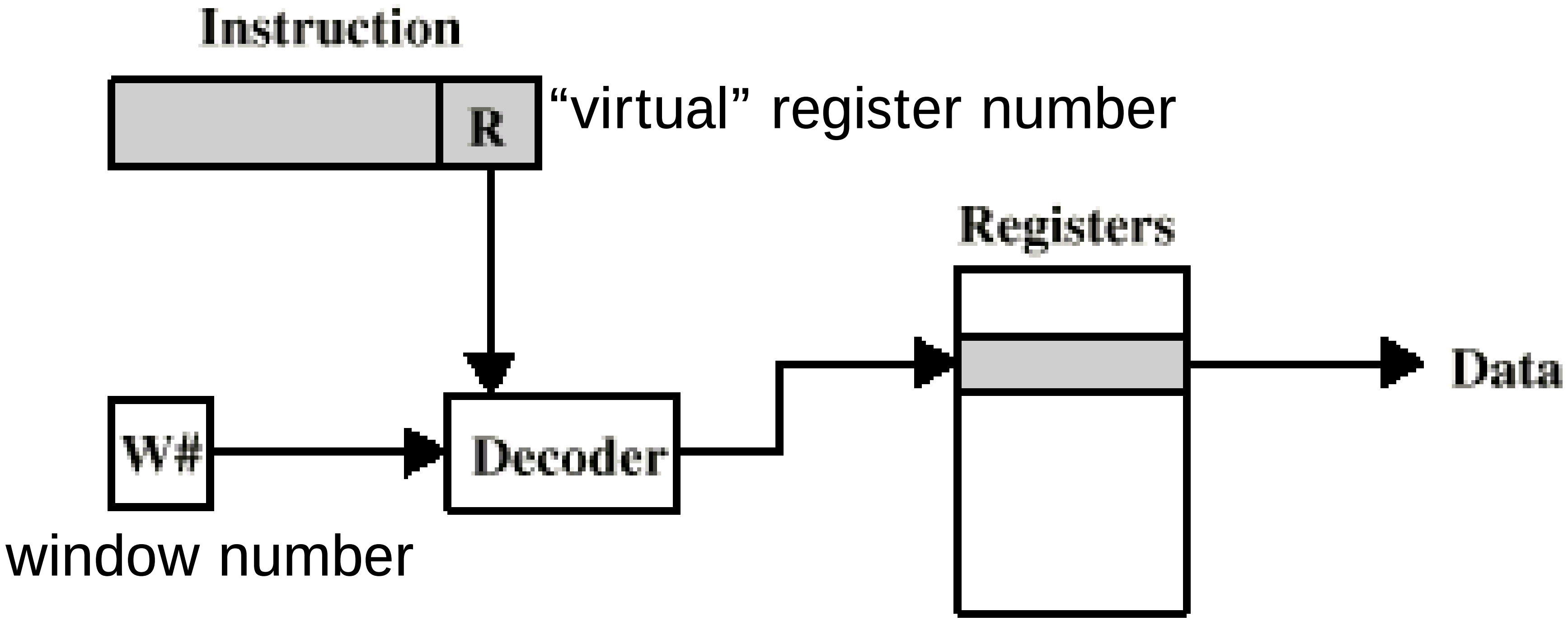
Global Variables - 2 Options

- Allocated by the compiler to memory
  - Straightforward
  - Inefficient for frequently accessed variables
- Have a set of registers for global variables
  - e.g., registers 0 - 7: global
    8 - 31: local to current window
  - Increased hardware burden
  - Compiler must decide which global variables should be designed to registers

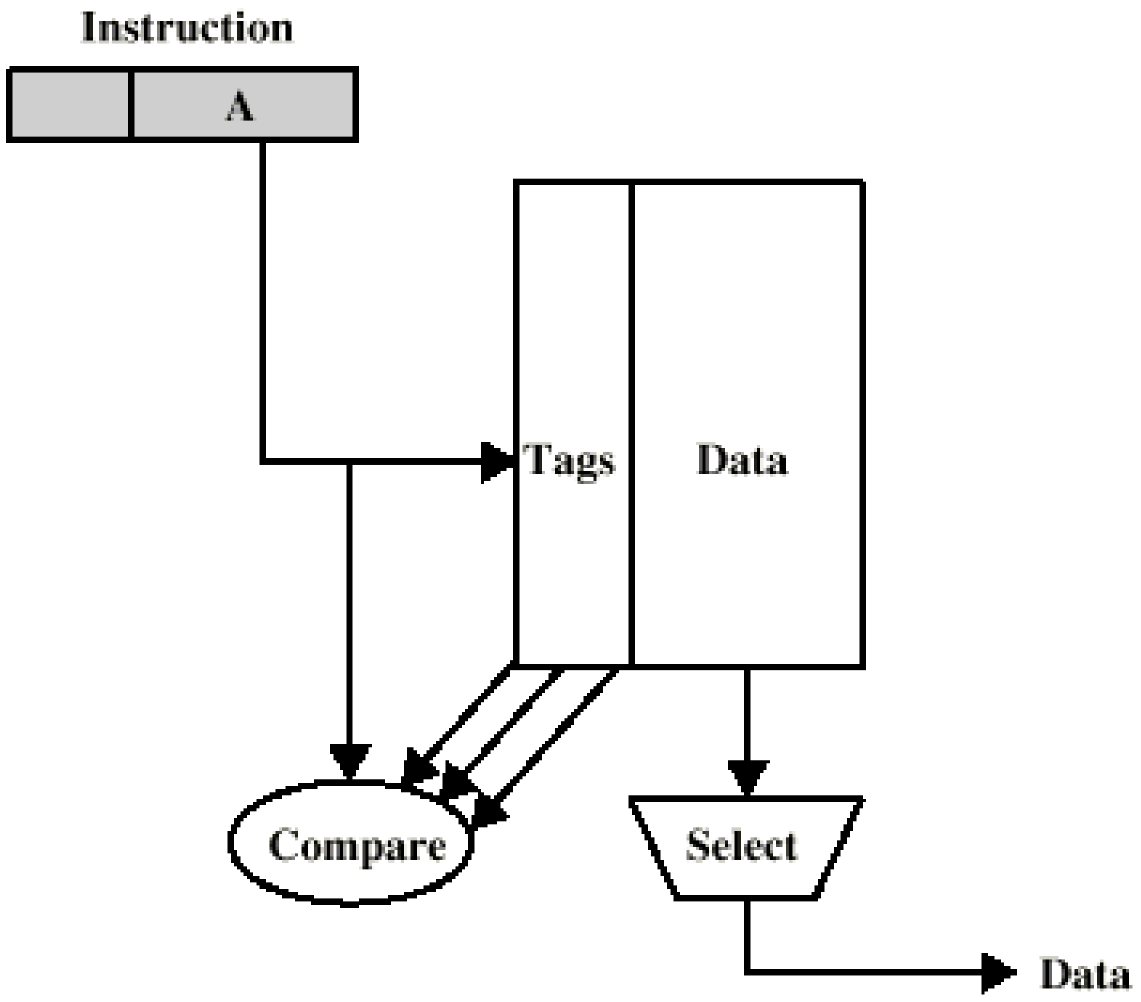
Registers v Cache

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing

Referencing a Scalar - Window Based Register File



Referencing a Scalar - Cache



Compiler Based Register Optimization

- Assume small number of registers (16-32)
- Optimizing use is up to compiler
- HLL programs have no explicit references to registers
  - usually - think about C - register int
- Assign symbolic or virtual register to each candidate variable
- Map (unlimited) symbolic registers to real registers
- Symbolic registers that do not overlap can share real registers
- If you run out of real registers some variables use memory

Graph Coloring (1)

- Given a graph of nodes and edges
- Assign a color to each node
- Adjacent nodes have different colors
- Use minimum number of colors
- Nodes are symbolic registers
- Two registers that are *live* in the same program fragment are joined by an edge
- Try to color the graph with *n* colors, where *n* is the number of real registers

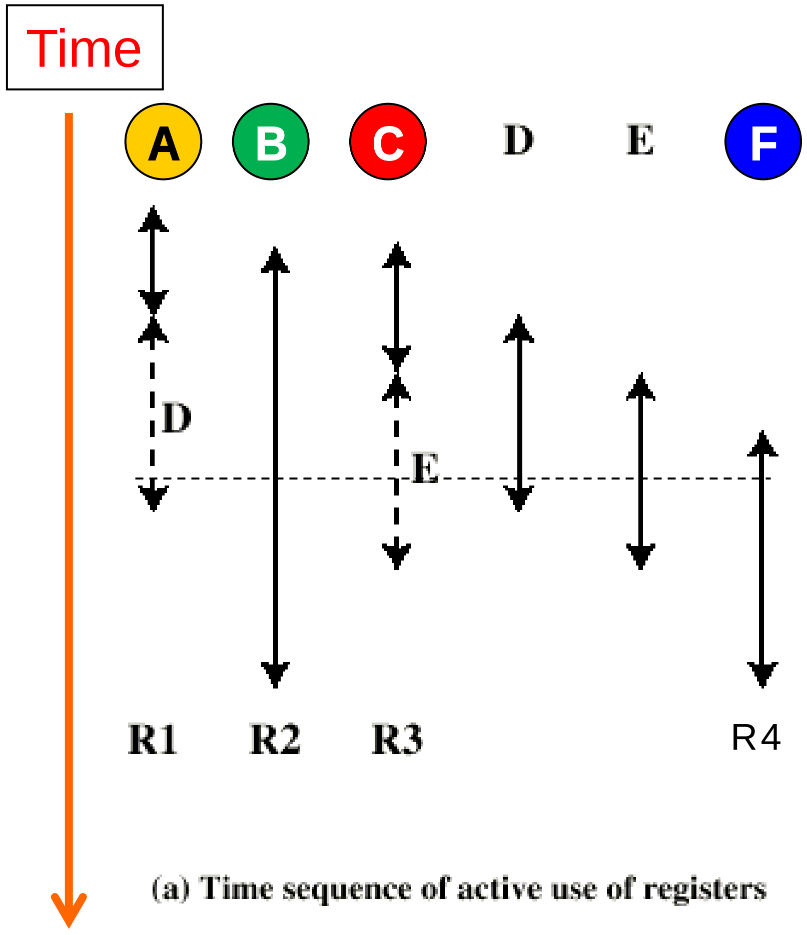
Graph Coloring (2)

- Nodes that can not be colored are placed in memory
- Formally, register interference graph G = (V, E), where

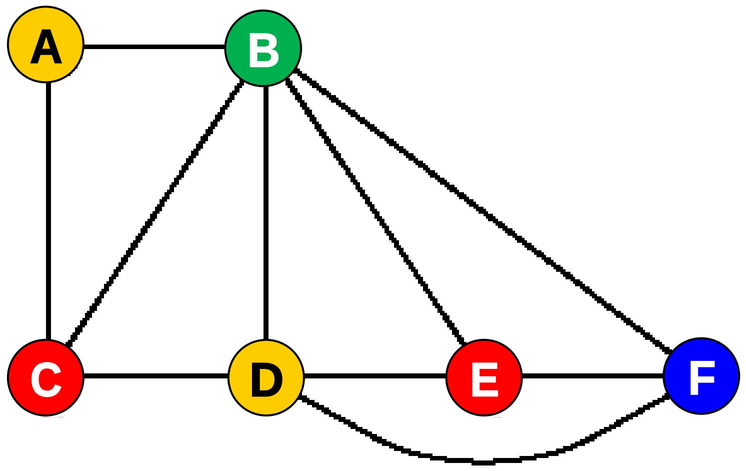
  V = {symbolic registers}

  E = $\{\overline{v_i v_j}: v_i, v_j \in V$ and v_i, v_j active at the same time$\}$
- Studies show
  - 64 registers are enough with simple register optimization
  - 32 registers are enough with sophisticated register optimization

Graph Coloring Approach



(a) Time sequence of active use of registers



4 colors → Use 4 registers !

(b) Register interference graph

Reduced Instruction Set Architecture (1)

- Why CISC?
  - Compiler simplification?
    - Disputed…
    - Complex machine instructions harder to exploit
    - Optimization more difficult
  - Smaller programs?
    - Program takes up less memory but…
    - Memory is now cheap
    - May not occupy less bits, just look shorter in symbolic form
      - + More instructions require longer op-codes
      - + Register references require fewer bits

Reduced Instruction Set Architecture (2)

- Why CISC (cont'd)
  - Faster programs?
    - More complex control unit
    - ➔ Larger microprogram control store
    - ➔ Simple instructions take longer to execute
    - BUT, bias towards use of simpler instructions

- It is far from clear that CISC is the appropriate solution

Reduced Instruction Set Architecture (3)

- “Potential” benefits of RISC
  - Performance
    - More effective compiler optimization
    - Faster control unit
    - More effective instruction pipelining
    - Faster response to interrupts
      (Recall: when is an interrupt processed?)
  - VLSI implementation
    - Smaller area dedicated to control unit
    - Easier design and implementation
      $\Rightarrow$ Shorter design and implementation time

RISC vs. CISC

- Not clear cut
- Many designs borrow from both philosophies
  - E.g. PowerPC no longer pure RISC
  - E.g. Pentium II and later incorporate RISC characteristics

RISC Characteristics

- One instruction per cycle
- Register to register operations
- Few, simple addressing modes
- Few, simple instruction formats
- Hardwired design (no microcode)
- Fixed instruction format
- More compile time/effort

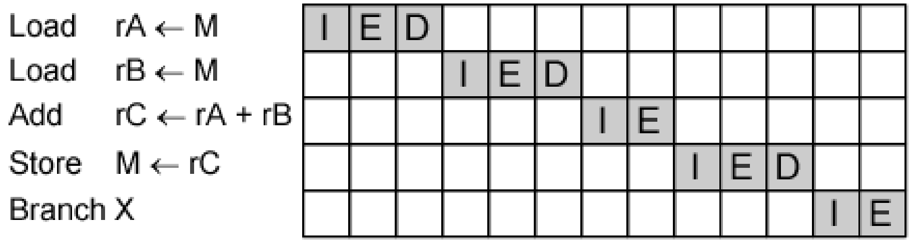
RISC Pipelining

- Most instructions are register to register
- Two phases of execution
  - I: Instruction fetch
  - E: Execute
    - ALU operation with register input and output
- For load and store
  - I: Instruction fetch
  - E: Execute
    - Calculate memory address
  - D: Memory
    - Register to memory or memory to register operation

Effects of Pipelining

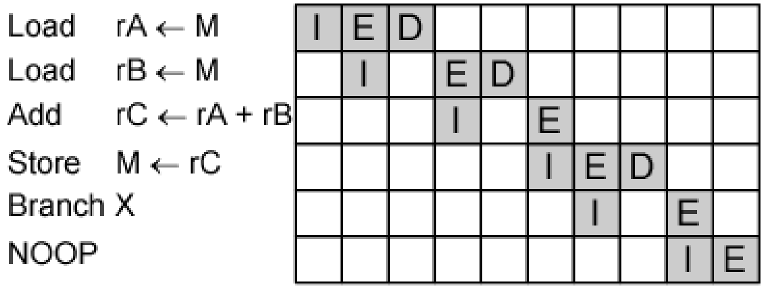
둘다 메모리 접근이 필요한 I,D 중첩불가

(b) I & E are executed at the same time
But, (1) only one memory access (per stage) is possible
(2) branch instruction interrupts the seq. flow of exe.



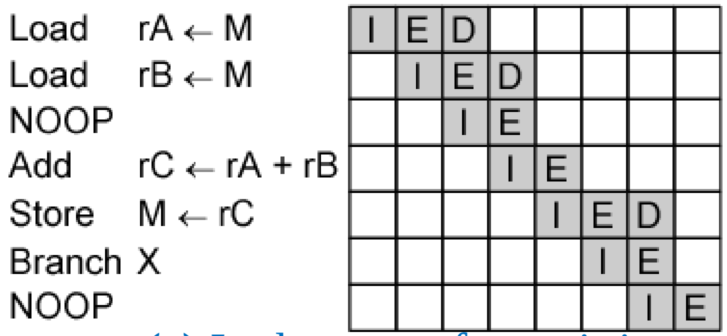
Load	rA ← M	I	E	D										
Load	rB ← M				I	E	D							
Add	rC ← rA + rB							I	E					
Store	M ← rC									I	E	D		
Branch X													I	E

(a) Sequential execution



Load	rA ← M	I	E	D							
Load	rB ← M		I		E	D					
Add	rC ← rA + rB				I		E				
Store	M ← rC						I	E	D		
Branch X								I		E	
NOOP										I	E

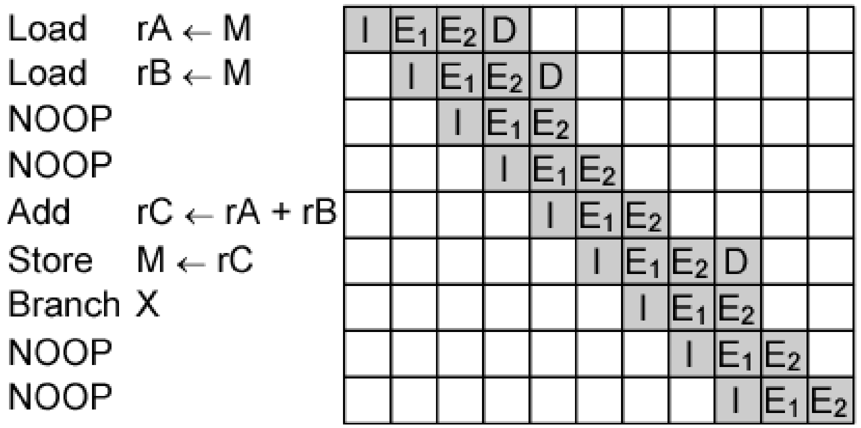
(b) Two-stage pipelined timing



Load	rA ← M	I	E	D					
Load	rB ← M		I	E	D				
NOOP				I	E				
Add	rC ← rA + rB				I	E			
Store	M ← rC					I	E	D	
Branch X							I	E	
NOOP								I	E

(c) In the case of permitting two mem. accesses per stage → three instructions can be overlapped

(c) Three-stage pipelined timing



Load	rA ← M	I	E_1	E_2	D							
Load	rB ← M		I	E_1	E_2	D						
NOOP				I	E_1	E_2						
NOOP					I	E_1	E_2					
Add	rC ← rA + rB					I	E_1	E_2				
Store	M ← rC						I	E_1	E_2	D		
Branch	X							I	E_1	E_2		
NOOP									I	E_1	E_2	
NOOP										I	E_1	E_2

(d) Four-stage pipelined timing

(d) E stage can be divided
- E1: register file read, E2:ALU operation/register write

Optimization of Pipelining

- Delayed branch
  - A delay slot is **an instruction slot that gets executed without the effects of a preceding instruction**
  - The most common form is **a single arbitrary instruction located immediately after a branch instruction on a RISC or DSP architecture**
    - this instruction will execute even if the preceding branch is taken
    - Thus, by design, the instructions appear to execute in an illogical or incorrect order
  - It is typical for assemblers to automatically reorder instructions by default, hiding the awkwardness from assembly developers and compilers.

https://en.wikipedia.org/wiki/Delay_slot

Normal and Delayed Branch

(1) NOOP이 들어갔으며, (2) ADD 1,rA이 delayed branch용 코드로 삽입됨. (1)은 pipeline을 regularize하기 위한 것 외에는 없음. (2)는 ADD 1, rA 코드를 실행할 수 있게 되어, 결국 성능향상 효과를 얻음.

To regularize the pipeline, a NOOP is inserted after the branch

Address	Normal Branch	Delayed Branch	Optimized Delayed Branch
100	LOAD X, rA	LOAD X, rA	LOAD X, rA
101	ADD 1, rA	ADD 1, rA	JUMP 105
102	JUMP 105	JUMP 106	(2) ADD 1, rA
103	ADD rA, rB	(1) NOOP	ADD rA, rB
104	SUB rC, rB	ADD rA, rB	SUB rC, rB
105	STORE rA, Z	SUB rC, rB	STORE rA, Z
106		STORE rA, Z	

Perf. is increased when two instructions (101, 102) are interchanged

Use of Delayed Branch

The ADD (103) instruction will be cleared by JUMP(102) instruction

Time

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

(a) Traditional Pipeline

The inserted NOOP (103) do not need special circuitry to clear the pipeline

	1	2	3	4	5	6	7
100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC Pipeline with Inserted NOOP

JUMP instruction (101) is fetched before the ADD (102). The ADD(102) is also executed and STORE(105) is successfully fetched.
Original semantics of the program is not altered but two less clock cycles are required for execution

	1	2	3	4	5	6
100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed Instructions

RISC vs. CISC Controversy (1)

- Has been 15+ years
- Quantitative assessment
  - Compare program sizes and execution speeds
- Qualitative assessment
  - Examine issues of high level language support and use of VLSI real estate

RISC vs. CISC Controversy (2)

- Problems
  - No pair of RISC and CISC that are directly comparable
  - No definitive set of test programs
  - Difficult to separate hardware effects from complier effects
  - Most comparisons done on “toy” rather than production machines
  - Most commercial devices are a mixture

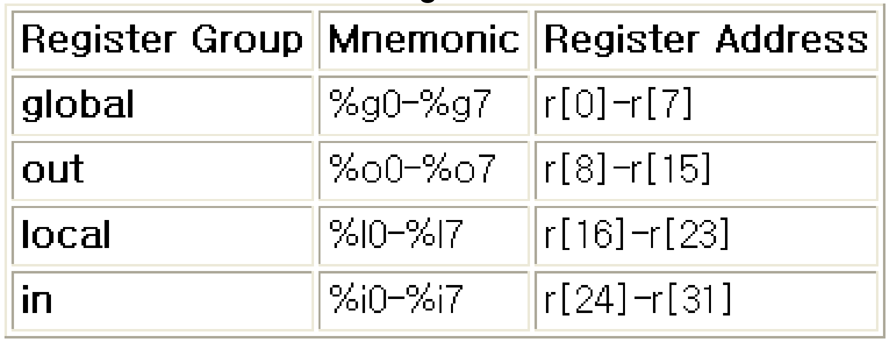
RISC vs. CISC Controversy (3)

- Has died down because of a gradual convergence of technologies
  - RISC systems become more complex
  - CISC designs have focused on issues traditionally associated with RISC

참고) Registers in the SPARC

- SPARC has 32 general purpose integer registers visible to the program at any given time.
  - Of these, 8 registers are *global* registers and 24 registers are in a register *window*.
  - A window consists of three groups of 8 registers, the *out*, *local*, and *in* registers.
  - See the following table.
  - A SPARC implementation can have from 2 to 32 windows, thus varying the number of registers from 40 to 520.
  - Most implementations have 7 or 8 windows. The variable number of registers is the principal reason for the SPARC being "scalable".

Table. Visible registers in SPARC



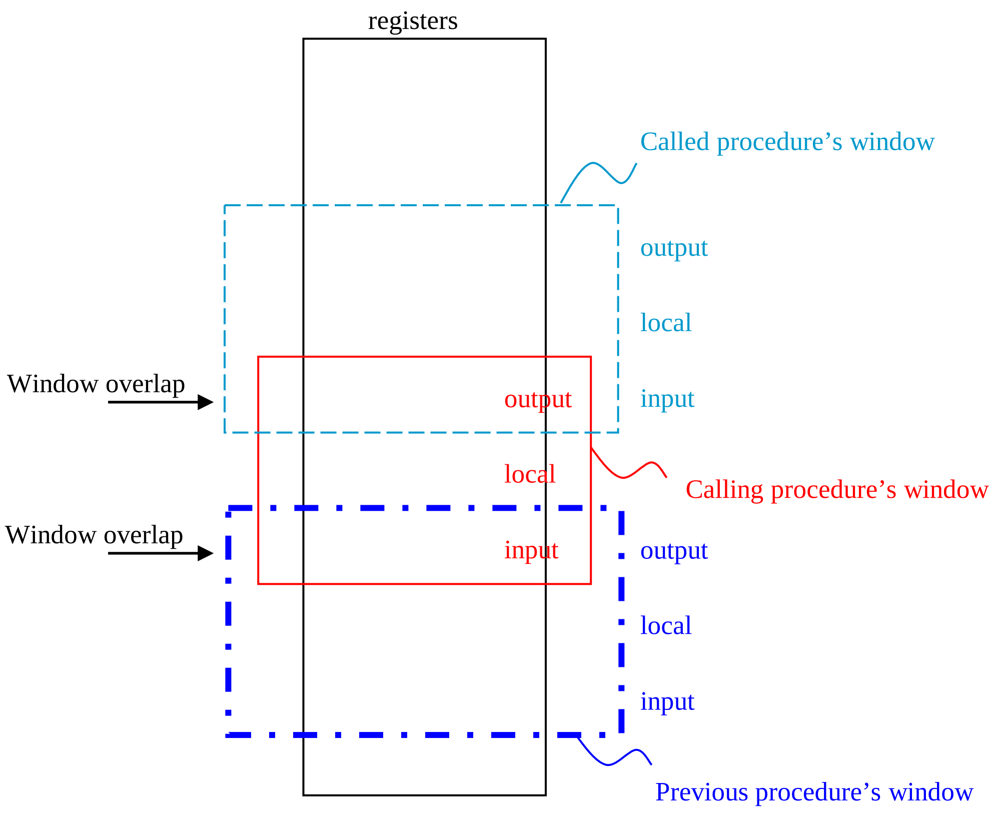
Register Group	Mnemonic	Register Address
global	%g0-%g7	r[0]-r[7]
out	%o0-%o7	r[8]-r[15]
local	%l0-%l7	r[16]-r[23]
in	%i0-%i7	r[24]-r[31]

참고) Registers in the SPARC

- At any given time, only one window is visible, as determined by the *current window pointer* (CWP) which is part of the processor status register (PSR).
  - This is a five bit value that can be decremented or incremented by the SAVE and RESTORE instructions, respectively.
  - These instructions are generally executed on procedure call and return (respectively).
  - The idea is that the *in* registers contain incoming parameters, the *local* register constitute scratch registers, the *out* registers contain outgoing parameters, and the *global* registers contain values that vary little between executions.
  - The register windows overlap partially, thus the *out* registers become renamed by SAVE to become the *in* registers of the called procedure. Thus, the memory traffic is reduced when going up and down the procedure call. Since this is a frequent operation, performance is improved.

참고) Registers in the SPARC

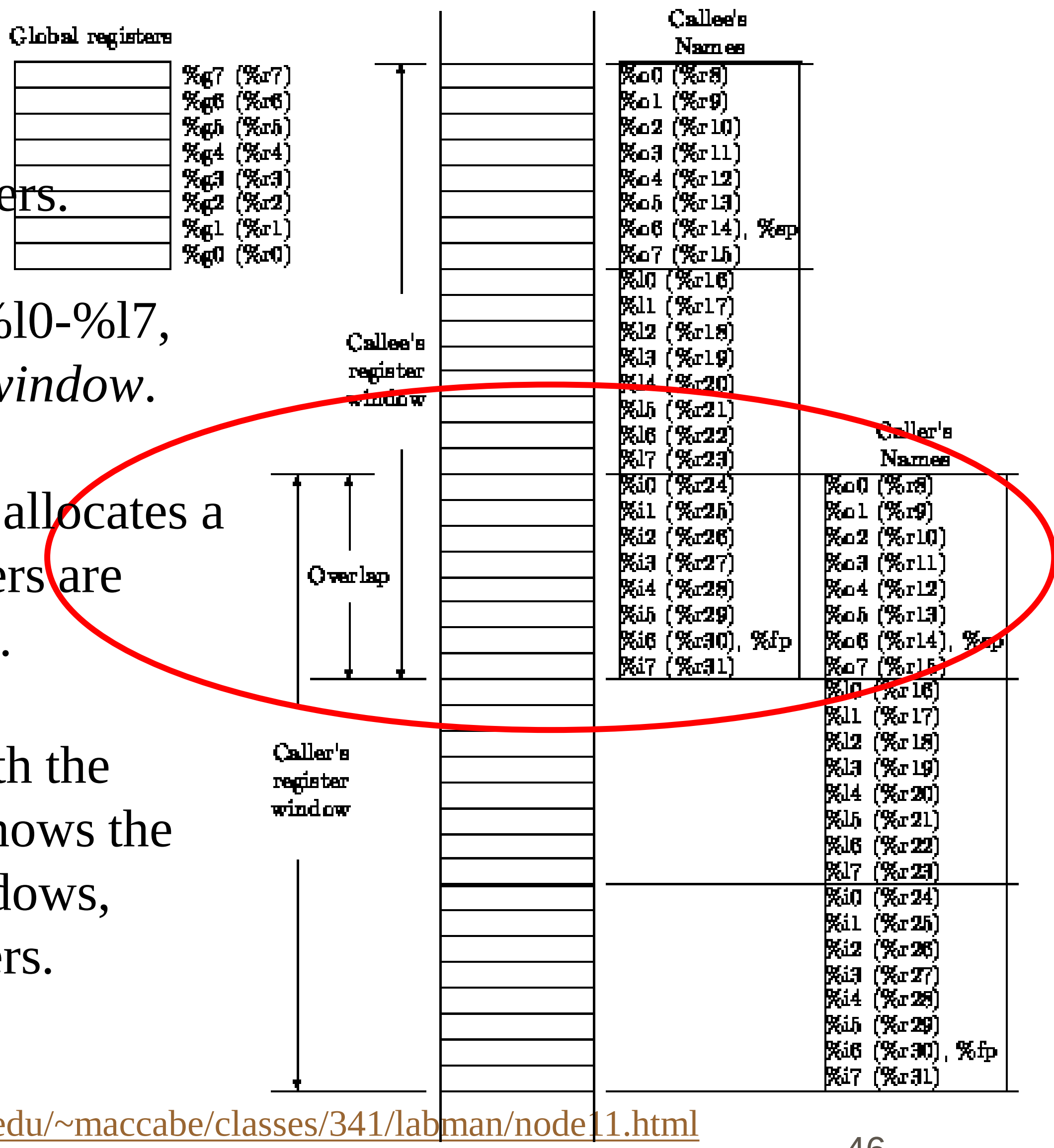
- Example



http://www.cs.unm.edu/~maccabe/classes/341/labman/node11.html

참고) Registers in the SPARC

- Example



All procedures share the global registers.

The remaining registers, %o0-%o7, %l0-%l7, and %i0-%i7, constitute the *register window*.

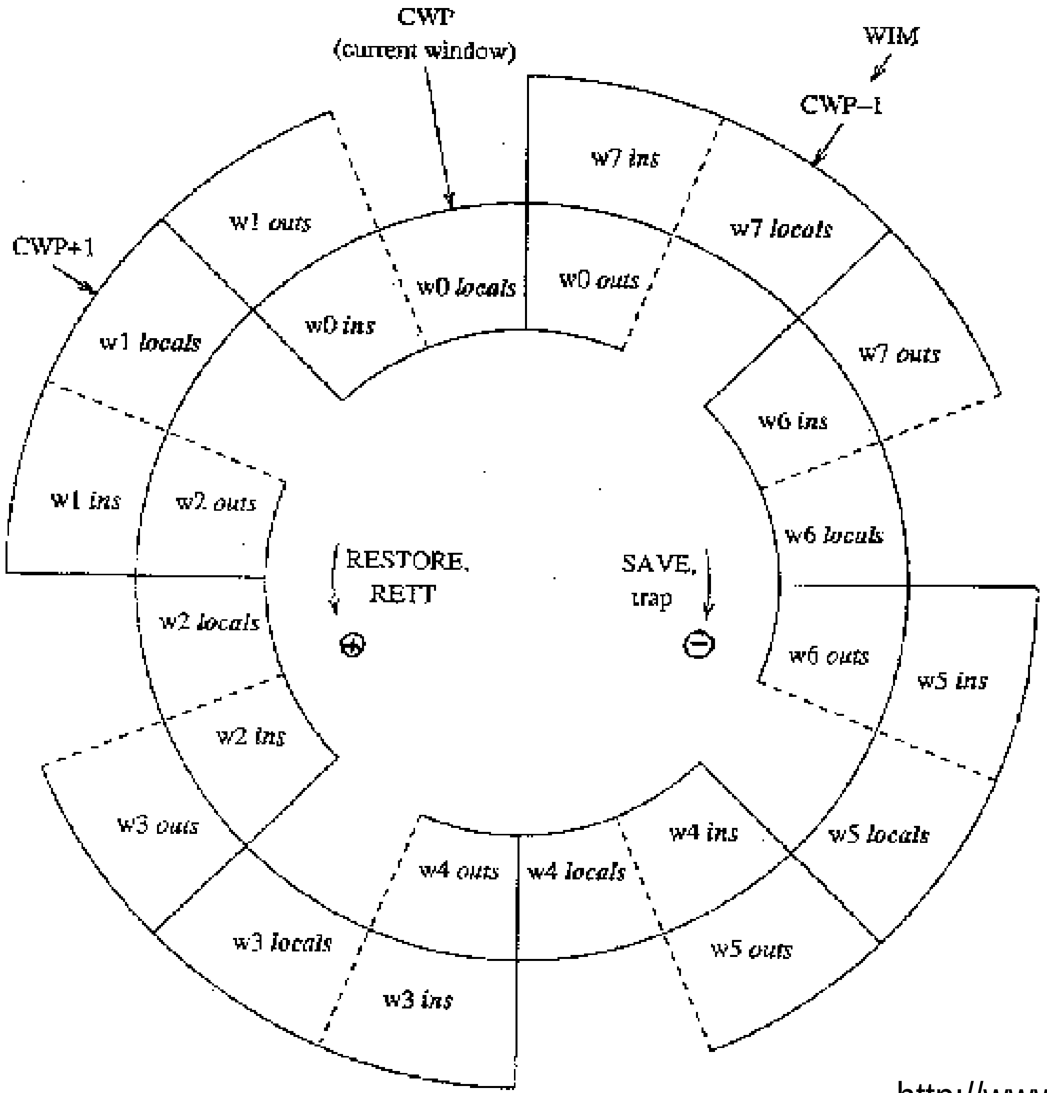
When a procedure starts execution, it allocates a set of 16 registers. The 16 new registers are used for the output and local registers.

The input registers are overlapped with the caller's output registers. This figure shows the details of overlapping of register windows, emphasizing the names for the registers.

http://www.cs.unm.edu/~maccabe/classes/341/labman/node11.html

참고) Registers in the SPARC

- Windowed registers in SPARC



CWP: current window pointer
WIM: window invalid mask

http://www.sics.se/~psm/sparcstack.html