RISC vs CISC

- **R**educed **I**nstruction **S**et **C**omputer vs **C**omplex **I**nstruction **S**et **C**omputers

- for a given benchmark the performance of a particular computer:

$$P = \frac{1}{I * C * \frac{1}{S}}$$

where *P = time to execute*
I = number of instructions executed
C = clock cycles per instruction
S = clock speed

- RISC approach attempts to reduce **C**
- CISC approach attempts to reduce **I**

- assuming identical clock speeds:

$C_{RISC} < C_{CISC}$ [both < 1 with superscalar designs]

a RISC will execute more instructions for a given benchmark than a CISC [≈10..30%]

RISC-I

- history
- RISC-1 designed by MSc students under the direction of David Patterson and Carlo H. Séquin at UCLA Berkeley
- released in 1982
- first RISC now accepted to be the IBM 801 [1980], but design not made public at the time
- John Cocke later won both the Turing award and the Presidential Medal of Science for his work on the 801
- RISC-1 similar to SPARC [Sun, Oracle] and DLX/MIPS [discussing its pipeline later]
- http://www.eecs.berkeley.edu/Pubs/TechRpts/1982/CSD-82-106.pdf

RISC-I Design Criteria

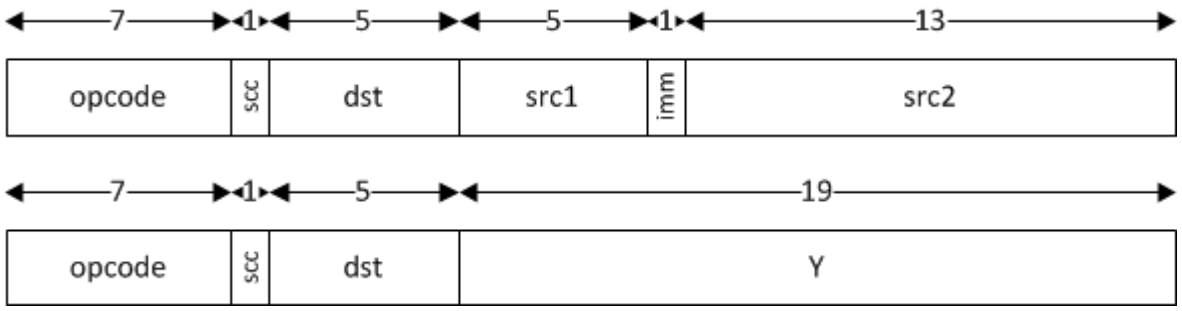
For an effective single chip solution artificially placed the following design constraints:

- execute one instruction per cycle [instructions must be simple to be executed in one clock cycle]
- make all instructions the same size [simplifies instruction decoding]
- access main memory with load and store instructions [load/store architecture]
- ONLY one addressing mode [indexed]
- limited support for high level languages [which means C and hence Unix]

  procedure calling, local variables, constants, …

RISC-I architecture

- 32 x 32 bit registers r0 .. r31 [R0 always 0]
- PC and PSW [status word]
- 31 different instructions [all 32 bits wide]
- instruction formats



- NB: 13 + 19 = 32

RISC-I architecture…

- opcode — 128 possible opcodes
- scc — if set, instruction updates the condition codes in PSW
- dst — specifies one of 32 registers r0..r31
- src1 — specifies one of 32 registers r0..r31
- imm, src2 — if (imm == 0) then 5 lower order bits of src2 specifies one of the 32 registers r0..r31

  if (imm == 1) then src2 is a sign extended 13 bit constant
- Y — 19 bit constant/offset used primarily by relative jumps and ldhi [load high immediate]

RISC-I Arithmetic Instructions

- 12 arithmetic instructions which take the form

  $R_{dst} = R_{src1}$ op S_2 NB: 3 address
  NB: S_2 specifies a register or an immediate constant

- operations

  add, add with carry, subtract, subtract with carry, reverse subtract, reverse subtract with carry

  and, or, xor

  sll, srl, sra [shifts register by S_2 bits where S_2 can be (i) an immediate constant or (ii) a value in a register]

  NB: NO mov, cmp, …

Synthesis of some IA32 instructions

set condition codes

mov	R_n, R_m	→	add R_0, R_m, R_n	
cmp	R_n, R_m	→	sub R_m, R_n, R_0, {C}	$R_m - R_n \rightarrow R_0$
test	R_n, R_n	→	and R_n, R_n, R_0, {C}	
mov	R_n, 0	→	add R_0, R_0, R_n	
neg	R_n	→	sub R_0, R_n, R_n	$R_0 - R_n \rightarrow R_n$ [twos complement]
not	R_n	→	xor R_n, #-1, R_n	[invert bits]
inc	R_n	→	add R_n, #1, R_n	

Synthesis of some IA32 instructions…

- loading constants $-2^{12} < N < 2^{12}-1$ [constant fits into src2 field]

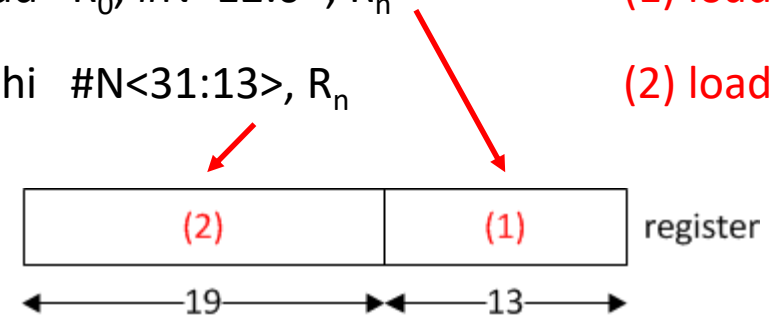
  mov R_n, N → add R_0, #N, R_n

- loading constants $(N < -2^{12}) \,||\, (N > 2^{12}-1)$ [constant too large for src2 field]

  construct large constants using two instructions

  mov** R_n, N → add R_0, #N<12:0>, R_n (1) load low 13 bits from src2 field

  ldhi #N<31:13>, R_n (2) load high 19 bits from Y field



** may not be correct

Load and Store Instructions

- 5 load and 3 store instructions

ldl	$(R_{src1})S_2$, R_{dst}	$R_{dst} = [R_{src1} + S_2]$	load 32 long [32 bits]
ldsu	$(R_{src1})S_2$, R_{dst}	$R_{dst} = [R_{src1} + S^2]$	load short unsigned [16 bits]
ldss	$(R_{src1})S_2$, R_{dst}	$R_{dst} = [R_{src1} + S_2]$	load short signed [16 bits]
ldbu	$(R_{src1})S_2$, R_{dst}	$R_{dst} = [R_{src1} + S_2]$	load byte unsigned
ldbs	$(R_{src1})S_2$, R_{dst}	$R_{dst} = [R_{src1} + S_2]$	load byte signed
stl	$(R_{src1})S_2$, R_{dst}	$[R_{src1} + S_2] = R_{dst}$	store long
sts	$(R_{src1})S_2$, R_{dst}	$[R_{src1} + S_2] = R_{dst}$	store short [low 16 bits of register]
stb	$(R_{src1})S_2$, R_{dst}	$[R_{src1} + S_2] = R_{dst}$	store byte [low 8 bits of register]

- load unsigned clears most significant bits of register
- load signed extends sign across most significant bits of register
- indexed addressing $[R_{src1} + S_2]$
- S_2 must be a constant [can also be a register in RISC II]

Synthesis of IA32 addressing modes

- register → add R_0, R_m, R_n
- immediate → add R_0, #N, R_n
- indexed → ldl (R_{src1})S_2, R_{dst}

  $R_{dst} = [R_{src1} + S_2]$

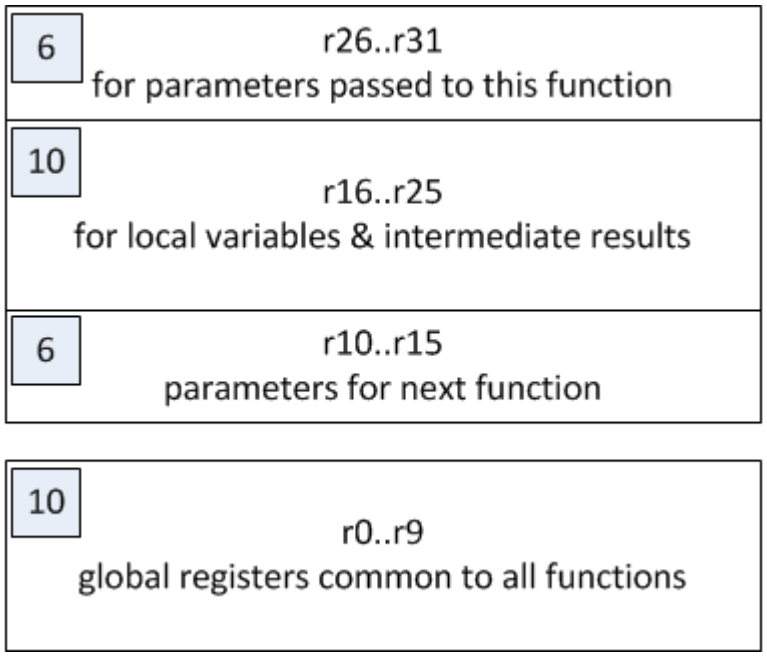
- absolute/direct → ldl (R_0)S_2, R_{dst}

  $R_{dst} = [S_2]$

  - since S_2 is a 13 bit signed constant this addressing mode is very limited
  - can ONLY access the top and bottom 4K (2^{12}) of the address space

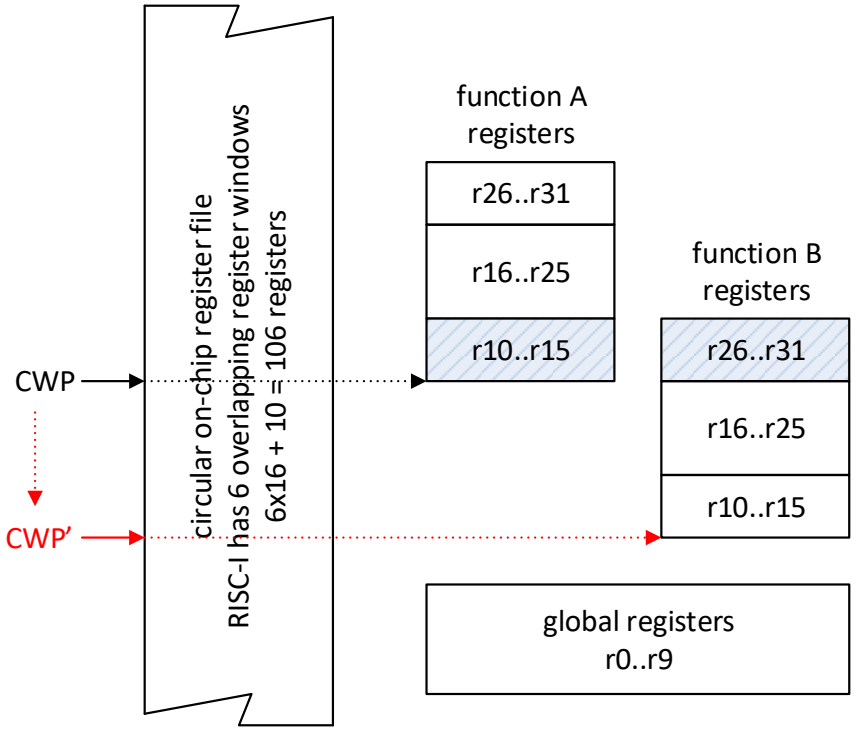
RISC-I Register Windows

- single cycle function call and return?
- need to consider parameter passing, allocation of local variables, saving of registers etc.
- "*since the RISC-I microprocessor core is so simple, there's plenty of chip area left for multiple register sets*"
- each function call allocates a new "window" of registers from a circular on-chip register file
- scheme based on the notion that the registers in a register window are used for specific purposes



RISC-I Register Windows Organisation

- example shows function A calling function B
- CWP [current window pointer] points to current register window in circular on-chip register file
- on a function call CWP moved so that a new window of registers r10..r25 [16 registers] allocated from the register file
  - r10..r15 of the calling function are now mapped onto r26..r31 of the called function [used to pass parameters]



RISC-I Function Call and Return

- CALL instruction

CALL $(R_{src1})\ S_2, R_{dst}$ *; called indexed*

CWP $\leftarrow$ *CWP + 1* *; move to next register window*
R_{dst} $\leftarrow$ *PC* *; return address saved in* R_{dst}
PC $\leftarrow$ $R_{src1} + S_2$ *; function start address*

- CALLR instruction

CALLR R_{dst}, Y *; call relative*

CWP $\leftarrow$ *CWP + 1* *; move to next register window*
R_{dst} $\leftarrow$ *PC* *; return address saved in* R_{dst}
PC $\leftarrow$ *PC + Y* *; relative jump to start address of function*

[NB: SPARC always uses r15 for the return address]

RISC-I Procedure Call and Return…

- the RET instruction takes the form

RET $(R_{dst})\ S_2$ *; return*

PC $\leftarrow R_{dst} + S_2$ *; return address + constant offset*
CWP $\leftarrow CWP - 1$ *; previous register window*

- CALL/CALLR and RET must use the same register for R_{dst}
- in most cases, functions can be called in a *"single cycle"*
  - *parameters stored directly in r10..r15*
  - *no need to save registers as a new register window allocated*
  - *use new registers for local variables*

CS3021/3421 agreed calling convention (for tutorial 3)

- need to agree how to use registers

```
int a(int i, int j) {  // parameter 1 will be in r26, parameter 2 in r27, …
   int k, l;           // local variables in r16, r17, …
   …
   b(1, 2);            // parameter 1 passed in r10, parameter 2 in r11, …
   …                   // CALLR saves return address in r25
   …
   return expr;        // return expr in r1, RET obtains return address from r25
}
```

- use r2 as a stack pointer and store global variables in r9, r8, … r3 where possible

Register File Overflow/Underflow

- what happens if functions nest too deeply and CPU runs out of register windows?

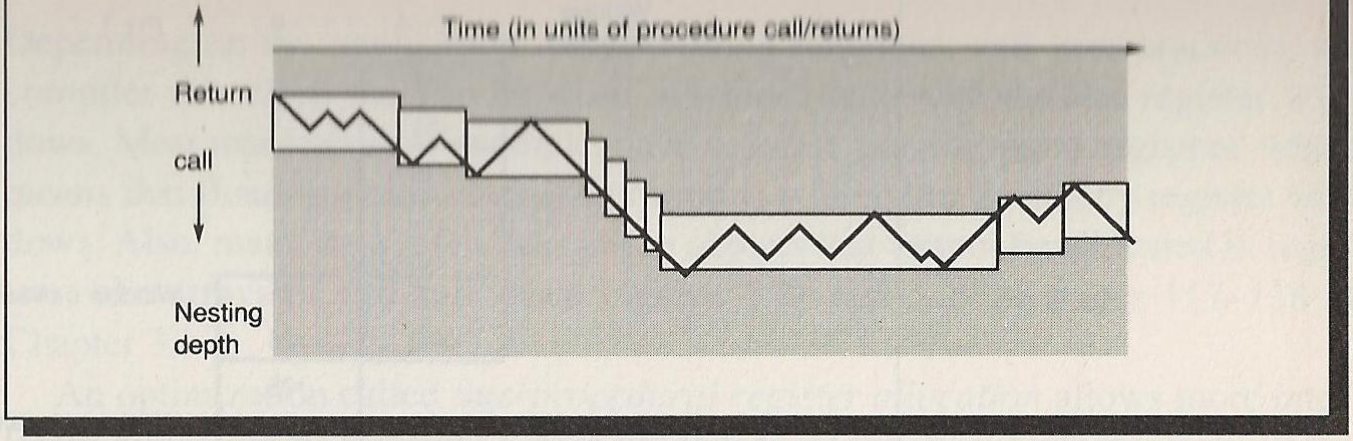


FIGURE 8.32 Change in procedure nesting depth over time. The boxes show procedure calls and returns inside the buffer before a window overflow or underflow. The program starts with three calls, a return, a call, a return, three calls, and then a window overflow.

- need a mechanism to handle register file overflow and underflow

[Hennessy and Patterson]

Register File Overflow/Underflow…

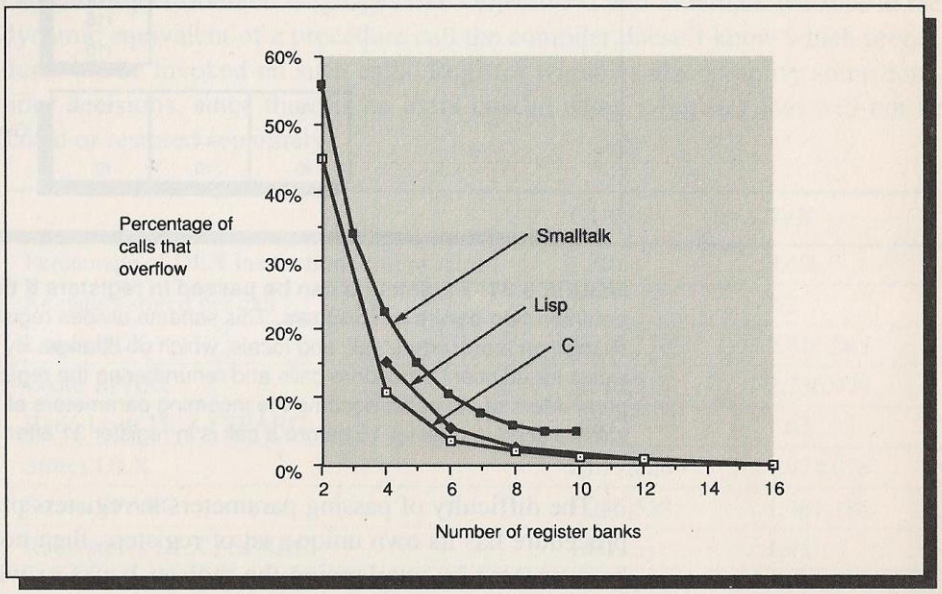


FIGURE 8.33 Number of banks or windows of registers versus overflow rate for several programs in C, LISP, and Smalltalk. The programs measured for C include a C compiler, a Pascal interpreter, troff, a sort program, and a few UNIX utilities [Halbert and Kessler 1980].The LISP measurements include a circuit simulator, a theorem prover, and several small LISP benchmarks [Taylor et al. 1986]. The Smalltalk programs come from the Smalltalk macro benchmarks [McCall 1983] which include a compiler, browser, and decompiler [Blakken 1983 and Ungar 1987].

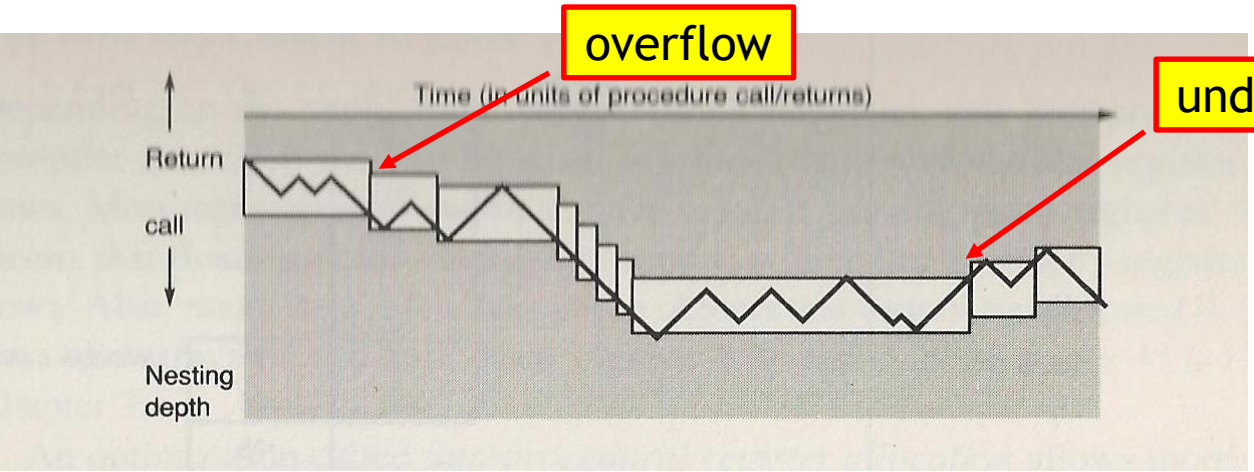
[Hennessy and Patterson]

Register File Overflow/Underflow…

- can run out of register windows if functions nest deep enough [overflow]
- register window overflow can ONLY occur on a CALL/CALLR
  - need to save [spill] oldest register window onto a stack maintained in main memory
- register window underflow can ONLY occur on a RET
  - there must always be at least two valid register windows in register file [window CWP contains registers r10..r25 and window CWP-1 contains r26..r31]
  - need to restore register window from stack maintained in main memory

Register File Overflow/Underflow …

- calls, returns, register file overflows, register file underflows and max depth



- register file contains 6 register windows

- call indicated by ＼ ≈43
- return indicated by ／ ≈36

} a completed program will have the same number of calls and returns

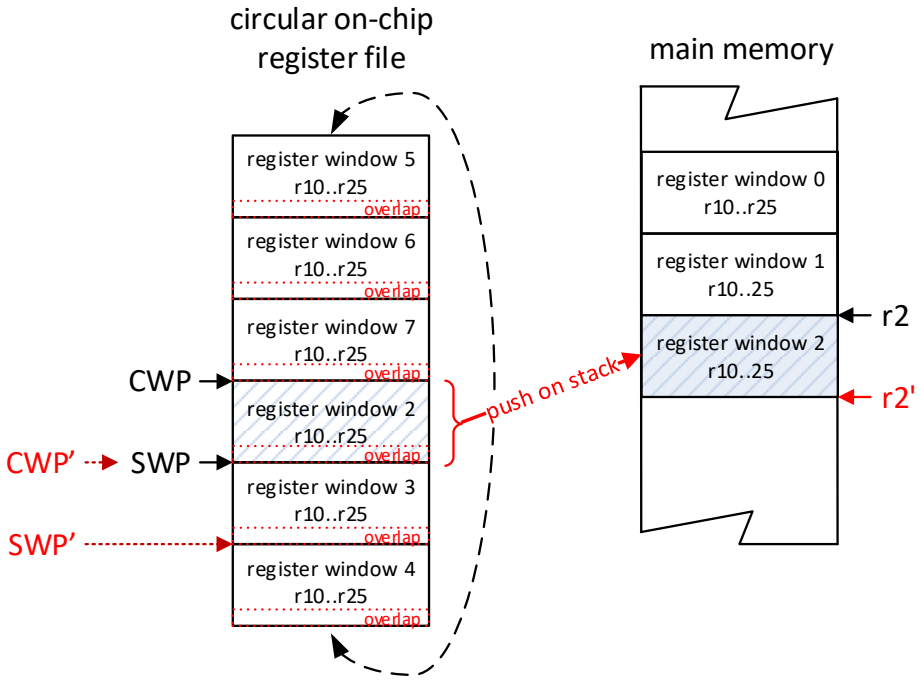
- register file overflows = 7
- register file underflows = 2

} a completed program will have the same number of register file overflows and underflows

- max depth 13 windows

Register File Overflow

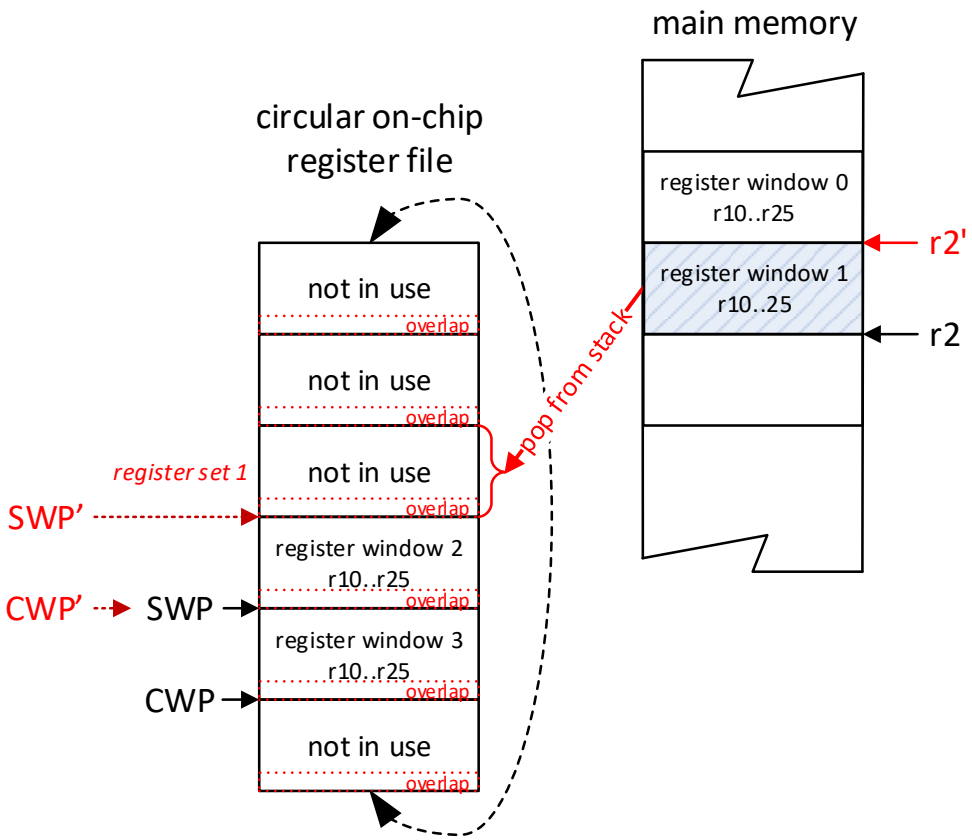
- typical register file overflow sequence
- SWP = save window pointer [points to oldest register window in register file]
- CWP++ and SWP++performed using modulo arithmetic as register file is *circular*
- r2 used as a stack pointer



1. function calls already 8 deep [register windows 0 to 7]
2. CWP -> register window 7, SWP -> register window 2 [oldest window]
3. two register windows already pushed onto stack [register windows 0 and 1]
4. another call will result in a register file overflow
5. register window 2 pushed onto stack [pointed to by SWP]
6. CWP and SWP *move down* one window [CWP++ and SWP++]

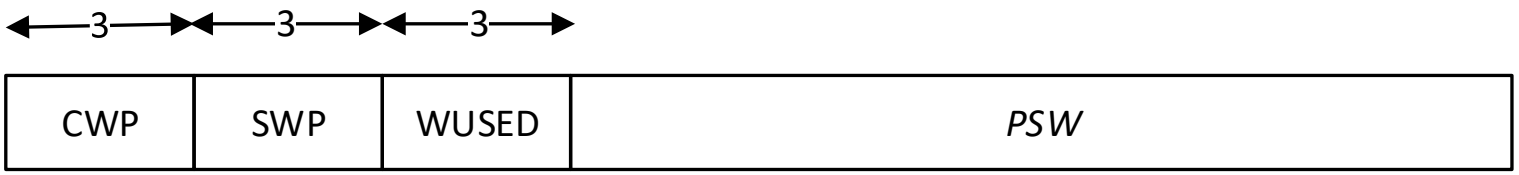
Register File Underflow

- typical register file underflow sequence
- always need 2 valid register windows in register file
- window CWP-1 contains CWP's r26..31 [on underflow SWP == CWP - 1]
- must restore window SWP-1
- CWP and SWP *move up* one window [CWP-- and SWP--]



Register File Overflow

- imagine the PSW maintains CWP, SWP and WUSED (number of windows in use)



- before a CALL/CALLR instruction is executed, the following test is made

```
if (WUSED == NWINDOWS)
    overflowTrapHandler();
    SWP++;
} else {
    WUSED++;
}
CWP++;
```

- CWP++ and SWP++ must handle wrap around
- NWINDOWS is the number of register windows in register file

Register File Overflow

- before a RET instruction is executed, the following test is made

```
if (WUSED == 2) {
    SWP--;
    underflowTrapHandler();
} else {
    WUSED--;
}
CWP--;
```

- CWP-- and SWP– must handle wrap around

- How might overflow and underflow be handled?? (i) instruction to switch to the SWP window so that r10..r25 can be saved or restored from stack using standard instructions (ii) instructions to increment/decrement SWP and (iii) an instruction to move back to the CWP window so the CALL/RET can be executed without generating an overflow/underflow.

Problems with Multiple Register Sets?

- must save/restore 16 registers on an overflow/underflow even though only a few may be in use
- saving multiple register sets on a context switch [between threads and processes]
- referencing variables held in registers by address [a register does NOT normally have an address]

```
p(int i, int *j) {
    *j = ...          // j passed by address
}
```

```
p(int i, int &j) {
    j = ...           // j passed by address
}
```

```
q() {
    int i, j;         // can j be allocated to a register as it is passed to p by address?
    ...               // i in r16 and j in r17?
    p(i, &j);         // pass i by value and j by address?
    ...
}
```

```
p(i, j)           // j passed by address
```

- a solution was proposed in original paper

Proposed Solution

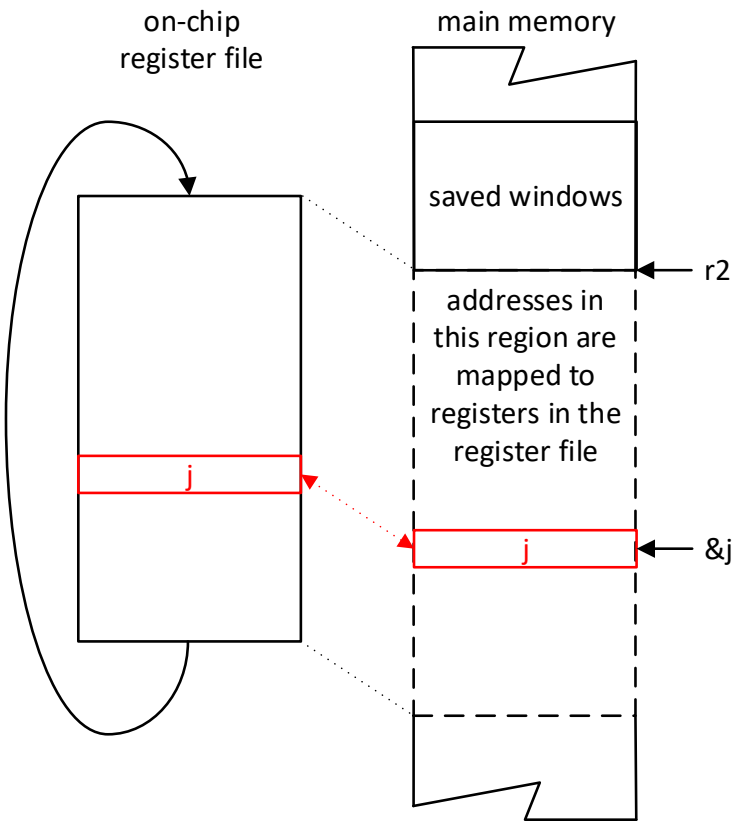
- solution proposed in original Computer IEEE paper

- *"RISC-I solves that problem by giving addresses to the window registers. By reserving a portion of the address space, we can determine, with one comparison, whether a register address points to a CPU register or to one that has overflowed into memory. Because the only instructions accessing memory (load & store) already take an extra cycle, we can add this feature without reducing their performance."*

- NOT implemented in RISC-I

Proposed Solution..

- register file can be thought of as sitting on the top of the stack in memory
- can then assign a notional address to each register in register file [where it would be stored on stack if spilled]
- inside Q, a register can be used for j
- address of j passed to p(), compiler able to generation instructions to calculate its address (relative to stack pointer r2) [the address is where the register would be stored in memory if spilled to the stack]
- *j in p() will be mapped by load and store instructions onto a register if the address "*maps*" to the register file otherwise memory will be accessed



RISC-I Pipeline

- two stage pipeline - fetch unit and execute unit
- normal instructions

fetch i1	execute i1 fetch i2	execute i2 fetch i3	execute i3

- *load/store instructions*

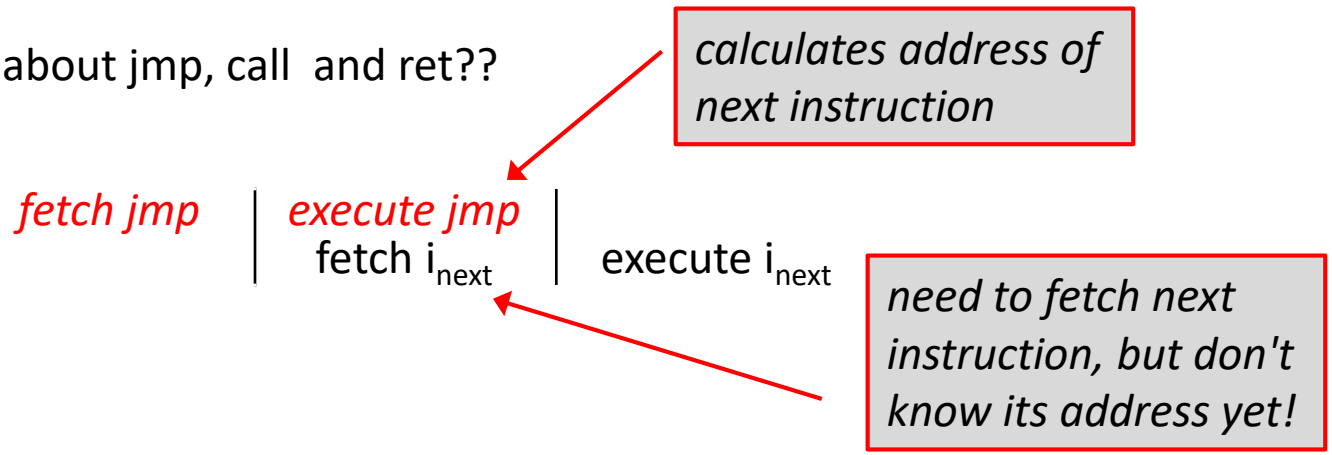
fetch load	*compute addr* fetch i2	*mem access* stall	execute 2 fetch i3

- pipeline stall arises because it is NOT possible to access memory twice in the same clock cycle [fetch the next instruction and read/write target of load/store]
  - load/store 2 cycles [latency 3 cycles]
  - others 1 cycle [latency 2 cycles]

Delayed Jumps

- RISC-I cycle long enough to (1) read registers, perform ALU operation and store result back in a register OR (2) read instruction from memory, BUT not both sequentially

- what about jmp, call and ret??



- jmp/call/ret instructions are problematic since it is NOT possible [during one clock cycle] to calculate the destination address and ALSO fetch the destination instruction

- RISC-I solution is to use "delayed jumps"

Delayed Jumps…

- jmp/call/ret effectively take place AFTER the following instruction [in the code] is executed

```
1            sub     r16, #1, r16 {C}        ;
2            jne     L                       ; conditional jmp
3            xor     r0, r0, r16             ;
4            sub     r17, #1, r17            ;

10     L:    sll     r16, 2, r16             ;
```

- if conditional jmp taken

  effective execution order 1, 3, 2, 10, …

- if conditional jmp NOT taken

  effective execution order 1, 3, 2, 4, …

NB: jmp condition evaluated at the *normal time* [condition codes set by instruction 1 in this case]

Delayed Jump Example

- consider the RISC-I code for the following code segment

```
i = 0;          // assume i in r16
while (i<j)     // assume j in r17
    i += f(i);  // parameter in r10, return address saved in r25 and result returned in r1
k = 0;          // assume k in r18
```

Delayed Jump Example…

- unoptimised
- place nop [xor r0, r0, r0] after each jmp/call/ret [in the delay slot]

```
        add     r0, r0, r16          // i = 0
L0:     sub     r16, r17, r0 {C}     // i < j ?
        jge     L1                   //
        xor     r0, r0, r0           // nop
swap    add     r0, r16, r10         // set up parameter in r10
        callr   r25, f               // return address saved in r25
        xor     r0, r0, r0           // nop
swap    add     r1, r16, r16         // i += f(i)
        jmp     L0                   //
        xor     r0, r0, r0           // nop
L1:     add     r0, r0, r18          // k = 0
```

what about this nop?

instruction at L1?

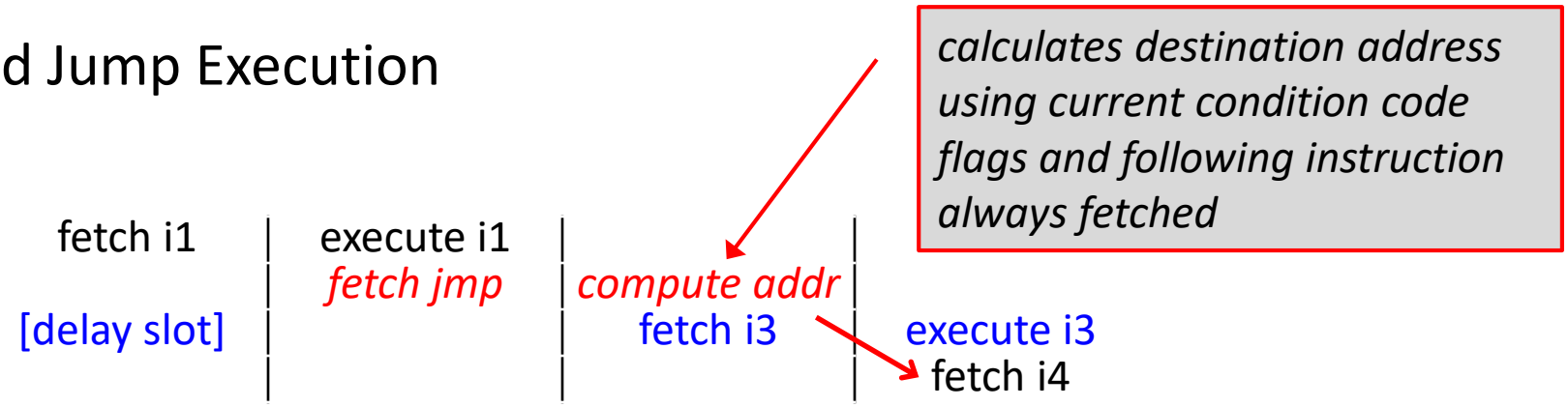
Delayed Jump Example

- reorganised and optimized

```
        add     r0, r0, r16          // i = 0
L0:     sub     r16, r17, r0 {C}     // i < j ?
        jge     L1                   //
        add     r0, r0, r18          // k can be zeroed many times as…
        callr   r25, f               // operation idempotent
        add     r0, r16, r10         // set up parameter in r10
        jmp     L0                   //
        add     r1, r16, r16         // i = i + f(i)
L1:
```

- managed to place useful instructions in each delay slot
- setting up parameter in instruction after call to f() appears strange at first

Delayed Jump Execution



- destination of jmp instruction is i4 [if jump NOT taken this will be the instruction after the delay slot]
- i3 executed in the delay slot
- *better* to execute an instruction in the delay slot than leaving execution unit idle
- since the instruction in the delay slot is fetched anyway, might as well execute it
- 60% of delay slots can be filled with useful instructions [Hennessy & Patterson]

What about??

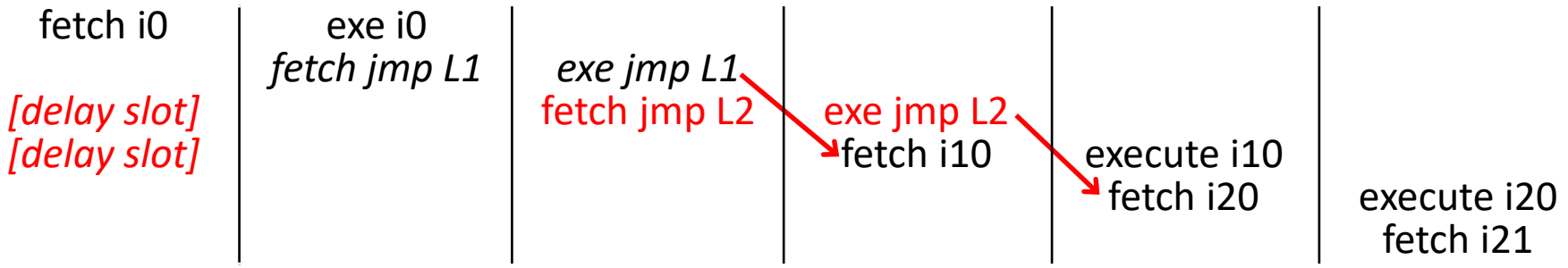
```
        i0      ....
        jmp     L1      // unconditional jump
        jmp     L2      // unconditional jump

L1:     i10     ....
        i11     ....

L2:     i20     ....
        i21     ....
```

- best approach is to draw a pipeline diagram

What about?...



- order i0, i10, i20, i21…