RISC AND PIPELINING

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 AND PIPELINING

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

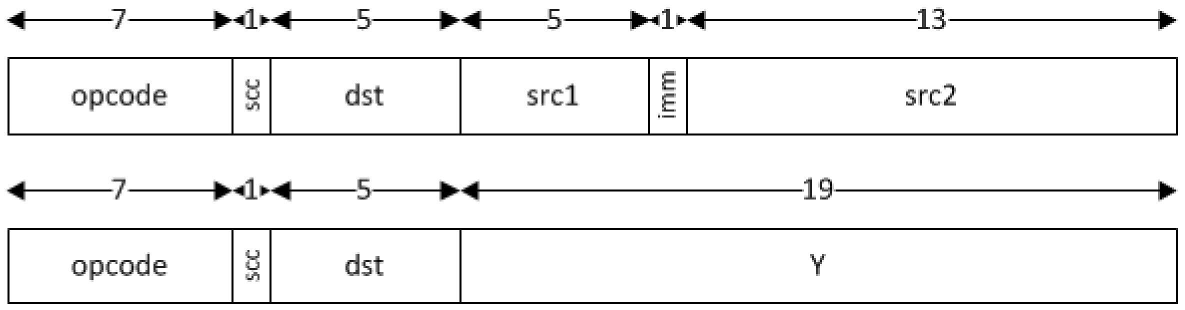
For an effective single chip solution artificially placed the following design constraints:

- 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 AND PIPELINING

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 <u>sign extended</u> 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	$\overleftarrow{R_n, R_m}$	$\rightarrow$	add $\overrightarrow{R_0, R_m}, R_n$	
cmp	R_n, R_m	$\rightarrow$	sub R_m, R_n, R_0, {C}	$R_m - R_n \rightarrow R_0$
test	R_n, R_n	$\rightarrow$	and R_n, R_n, R_0, {C}	
mov	R_n, 0	$\rightarrow$	add R_0, R_0, R_n	
neg	R_n	$\rightarrow$	sub R_0, R_n, R_n	$R_0 - R_n \rightarrow R_n$ [twos complement]
not	R_n	$\rightarrow$	xor R_n, #-1, R_n	[invert bits]
inc	R_n	$\rightarrow$	add R_n, #1, R_n	

Synthesis of some IA32 instructions…

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

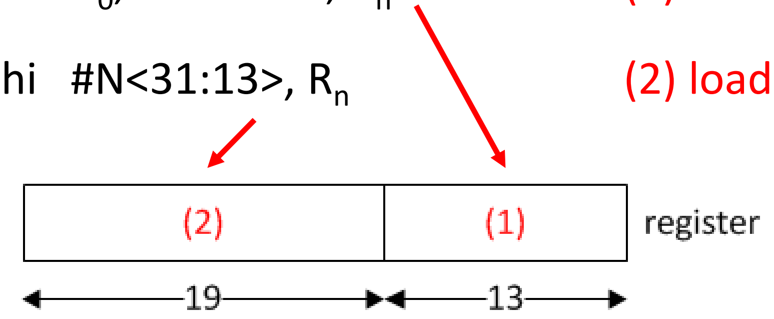
mov R_n, N $\rightarrow$ add R_0, #N, R_n

- 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 $\rightarrow$ 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 $\rightarrow$ add R_0, R_m, R_n
- immediate $\rightarrow$ add R_0, #N, R_n
- indexed $\rightarrow$ ldl (R_{src1})S_2, R_{dst}

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

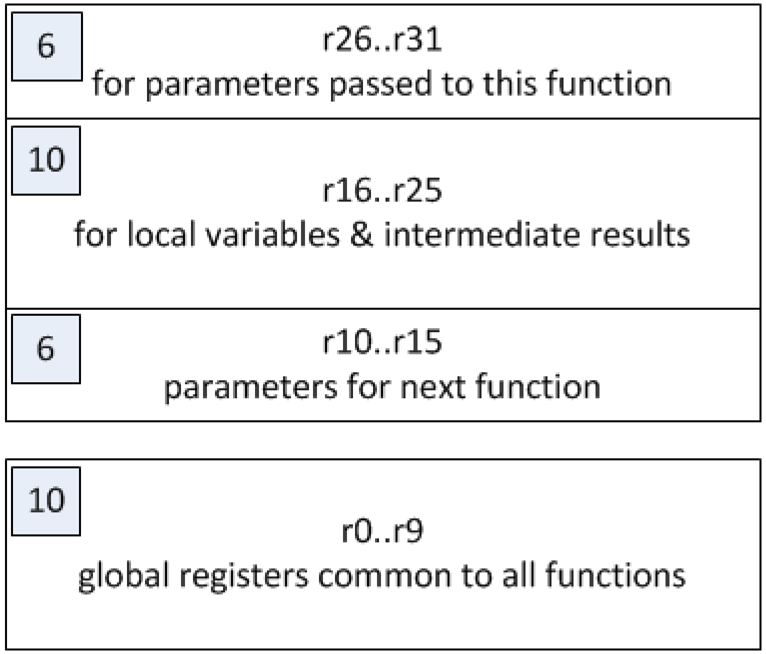
- absolute/direct $\rightarrow$ ldl (R_0)S_2, R_{dst}

  $R_{dst} = [S_2]$

  - 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

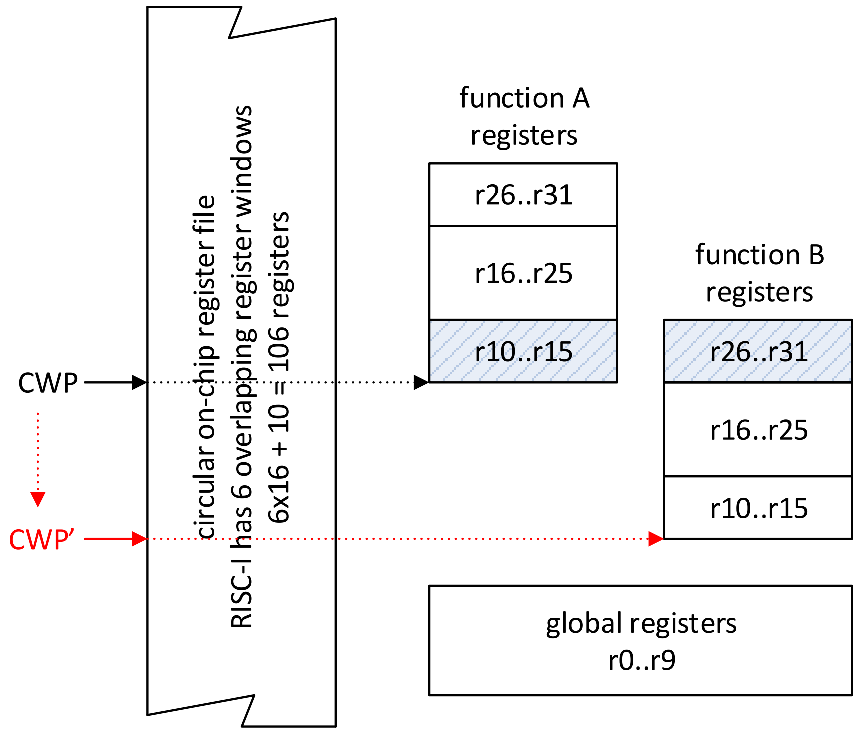
RISC-I Register Windows

- 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


6
r26..r31
for parameters passed to this function
10
r16..r25
for local variables & intermediate results
6
r10..r15
parameters for next function
10
r0..r9
global registers common to all functions

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?

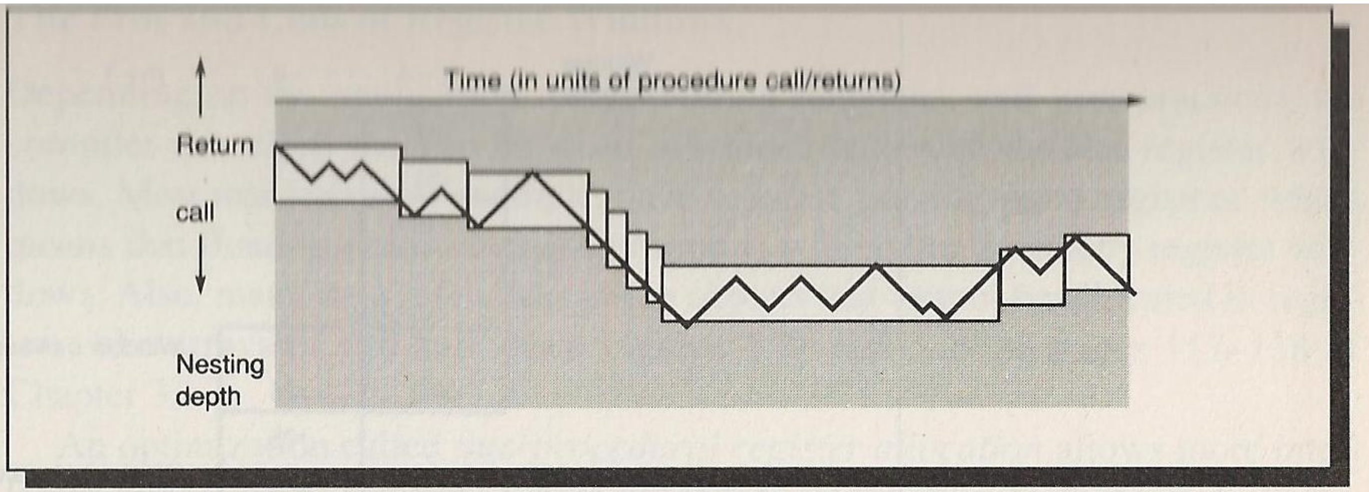


FIGURE 8.32 Change in procedure nesting depth over time. The boxes show procedure calls and returns inside the buffer before a window overflow or underflow. The program starts with three calls, a return, a call, a return, three calls, and then a window overflow.

- need a mechanism to handle register file overflow and underflow

[Hennessy and Patterson]

Register File Overflow/Underflow…

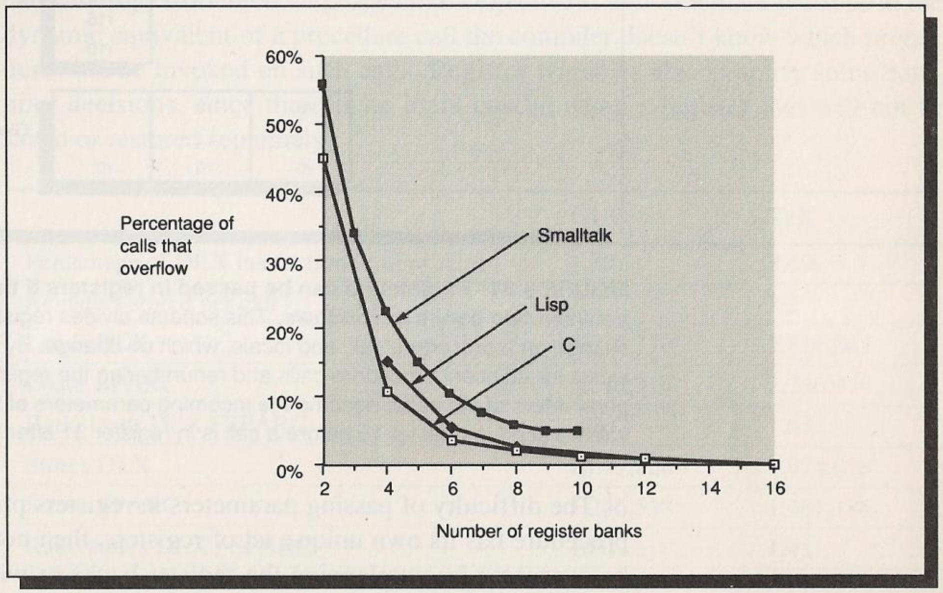


FIGURE 8.33 Number of banks or windows of registers versus overflow rate for several programs in C, LISP, and Smalltalk. The programs measured for C include a C compiler, a Pascal interpreter, troff, a sort program, and a few UNIX utilities [Halbert and Kessler 1980].The LISP measurements include a circuit 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].

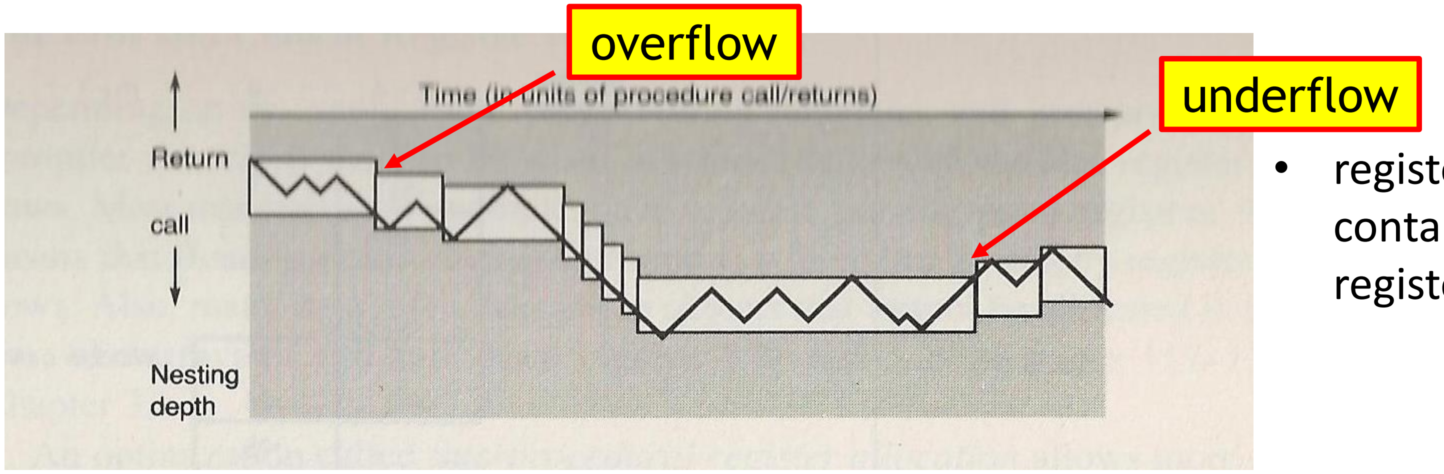
[Hennessy and Patterson]

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

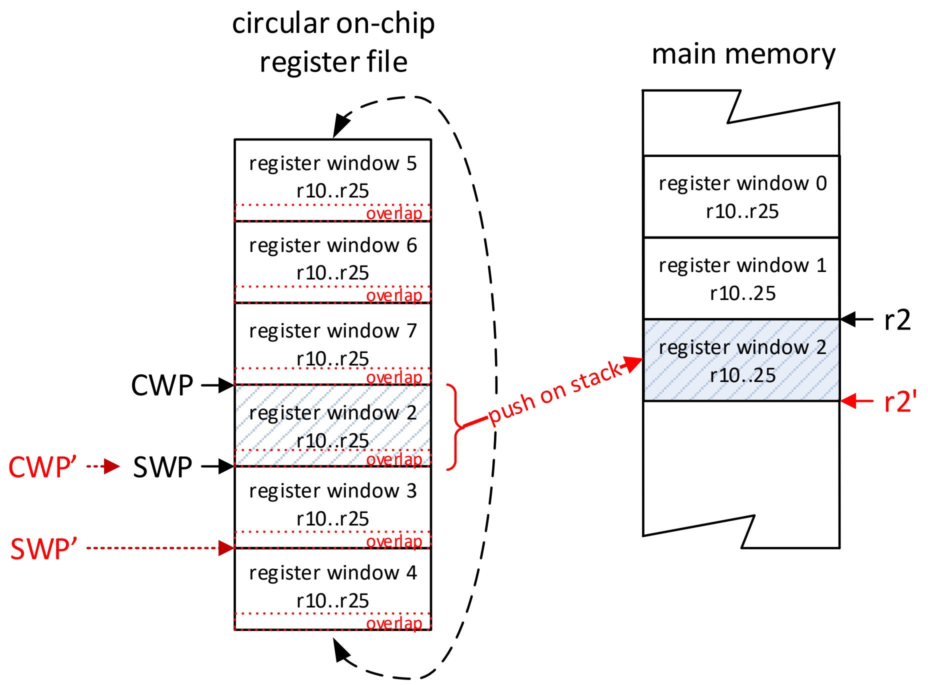
- register file overflows = 7
- register file underflows = 2

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

- max depth 13 windows

Register File Overflow

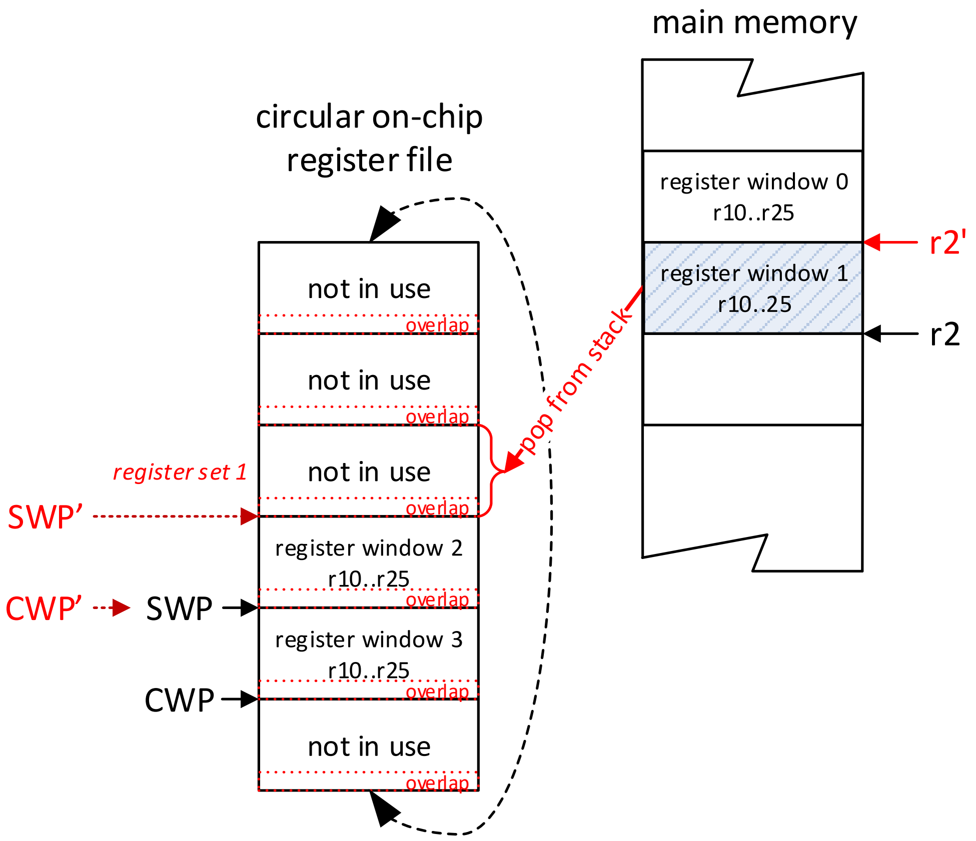
- typical register file overflow sequence
- SWP = save window pointer [points to oldest register window in register file]
- CWP++ and SWP++performed using modulo arithmetic as register file is *circular*
- r2 used as a stack pointer



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++]

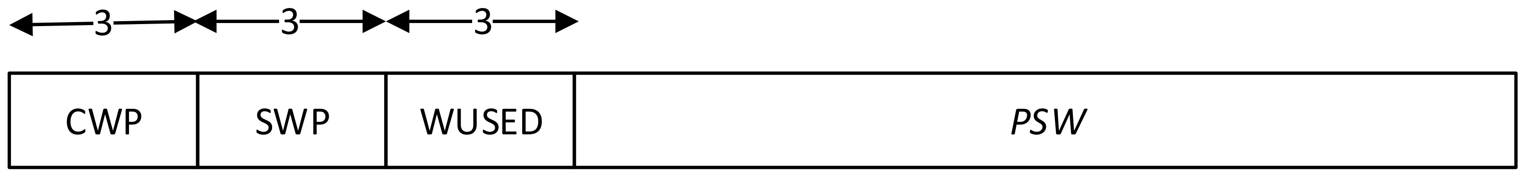
Register File Underflow

- 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
```

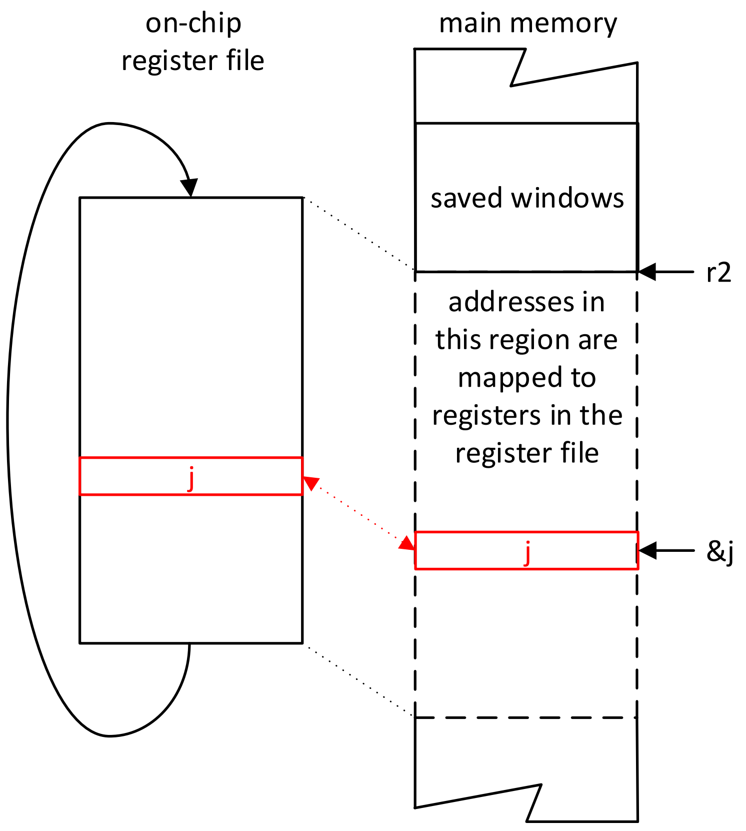
- a solution was proposed in original paper

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


on-chip register file
main memory
saved windows
r2
addresses in this region are mapped to registers in the register file
j
j
&j

RISC AND PIPELINING

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

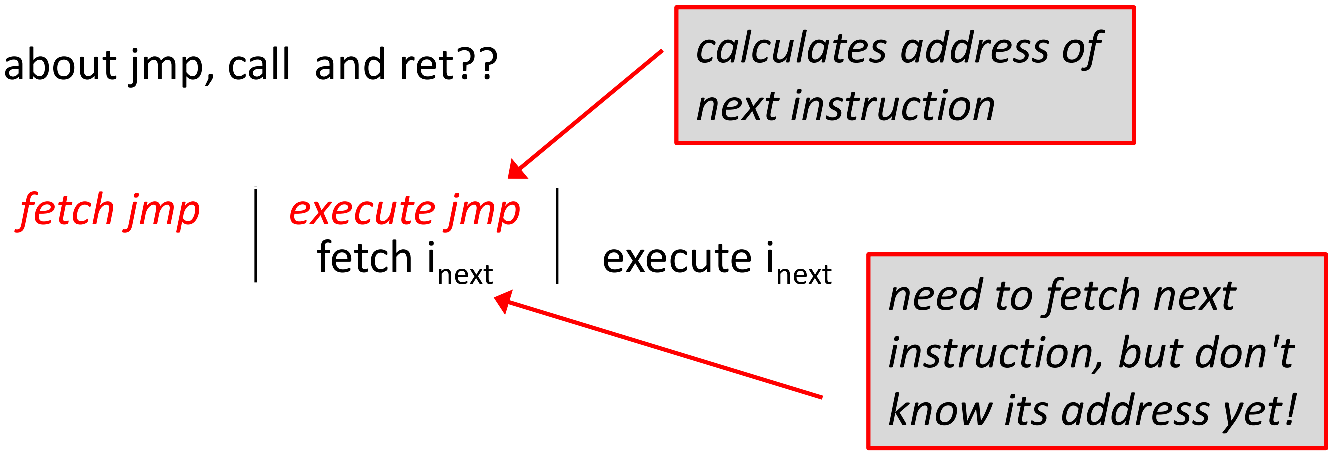
- pipeline stall arises because it is NOT possible to access memory twice in the same clock cycle [fetch the next instruction and read/write target of load/store]

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??


calculates address of next instruction
fetch jmp
execute jmp
fetch i_{next}
execute i_{next}
need to fetch next instruction, but don't know its address yet!

- 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

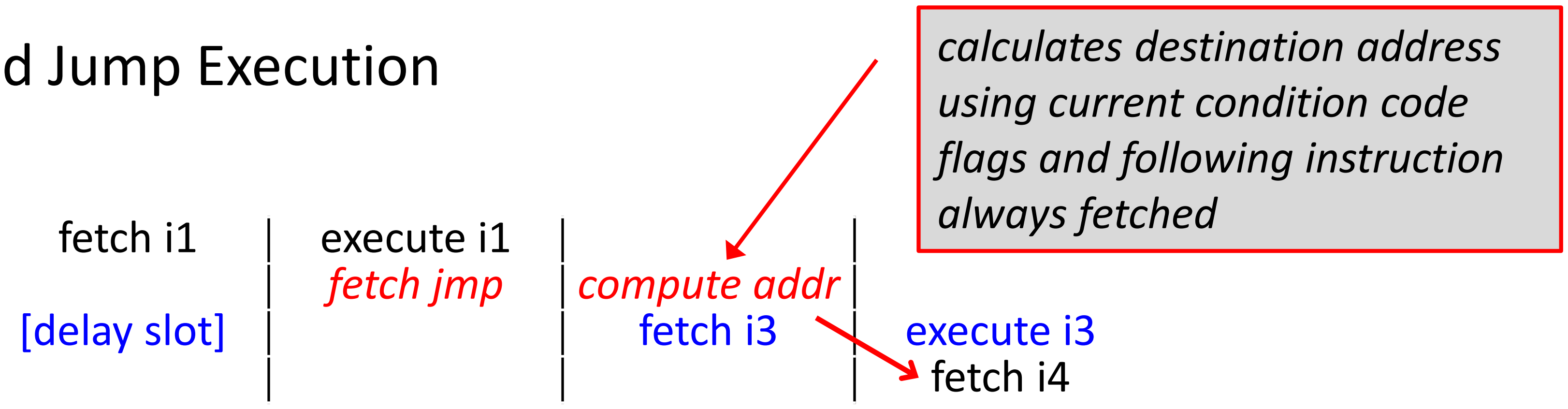
- reorganised and optimized

```
        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

RISC AND PIPELINING

Delayed Jump Execution



calculates destination address using current condition code flags and following instruction always fetched

- 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]

RISC AND PIPELINING

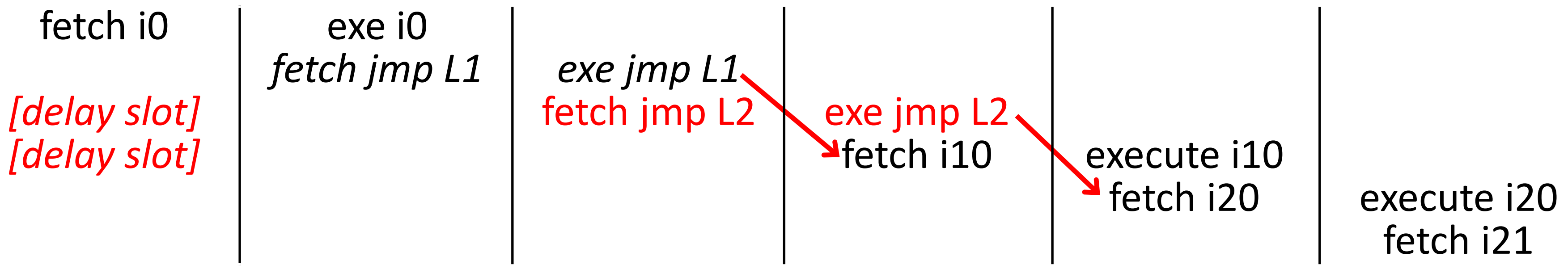
What about??

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

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

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

- best approach is to draw a pipeline diagram

RISC and Pipelining

What about?...



- order i0, i10, i20, i21...

RISC and Pipelining

Pipelining

- key implementation technique for speeding up CPUs [see Hennessy & Patterson]
- break each instruction into a series of small steps and execute them in parallel [steps from different instructions]
  - think of a car assembly line!
- clock rate set by the time needed for the longest step - ideally time for each step should be equal
- consider a 5 stage instruction pipeline for the hypothetical DLX microprocessor [after Knuth's MIX]

IF	instruction fetch
ID	instruction decode and register fetch [operands]
EX	execution and effective address calculation
MA	memory access
WB	write back [into a register]

RISC AND PIPELINING

Pipelining...

i	IF	ID	EX	MA	WB				
i+1		IF	ID	EX	MA	WB			
i+2			IF	ID	EX	MA	WB		
i+3				IF	ID	EX	MA	WB	
i+4					IF	ID	EX	MA	WB

- execution time of an individual instruction remains the same...
- BUT throughput increased by the depth of the pipeline [5 times in this case]
- clock frequency 5 times faster than non-pipelined implementation
- good performance if pipeline runs without stalling

RISC AND PIPELINING

Pipelining…

- for example, pipeline stalled while data is read from memory if memory access causes a cache miss [cache hit: 1 cycle, cache miss: 3 cycles]

i	IF	ID	EX	MA	MA	MA	WB				
i+1		IF	ID	stall	stall	EX	MA	WB			
i+2			IF	IF	IF	ID	EX	MA	WB		
i+3						IF	ID	EX	MA	WB	
i+4							IF	ID	EX	MA	WB

- stall normally between ID and EX phases

- instruction issued [from ID to EX phases] when it can be executed without stalling

  - *2 cycle cache miss penalty*

Pipelining…

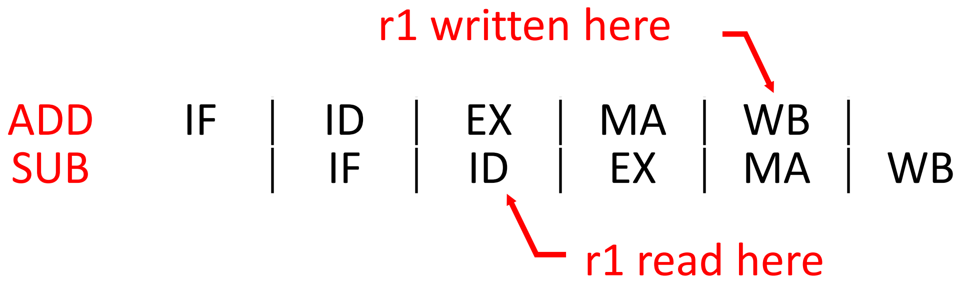
- ALSO note that a non-pipelined DLX requires 2 memory access every 5 clock cycles <u>while</u> a pipelined DLX requires 2 memory accesses per clock cycle
  - IF: fetch instruction from memory
  - MA: read/write data from/to memory
  - helped by separate instruction and data caches internally [Harvard Architecture]

Data Hazards

- consider the execution of the following instructions

r1 = r2 + r3	[ADD]
r4 = r1 - r5	[SUB]

- ADD instruction writes r1 in the WB stage, but SUB reads r1 in the ID stage

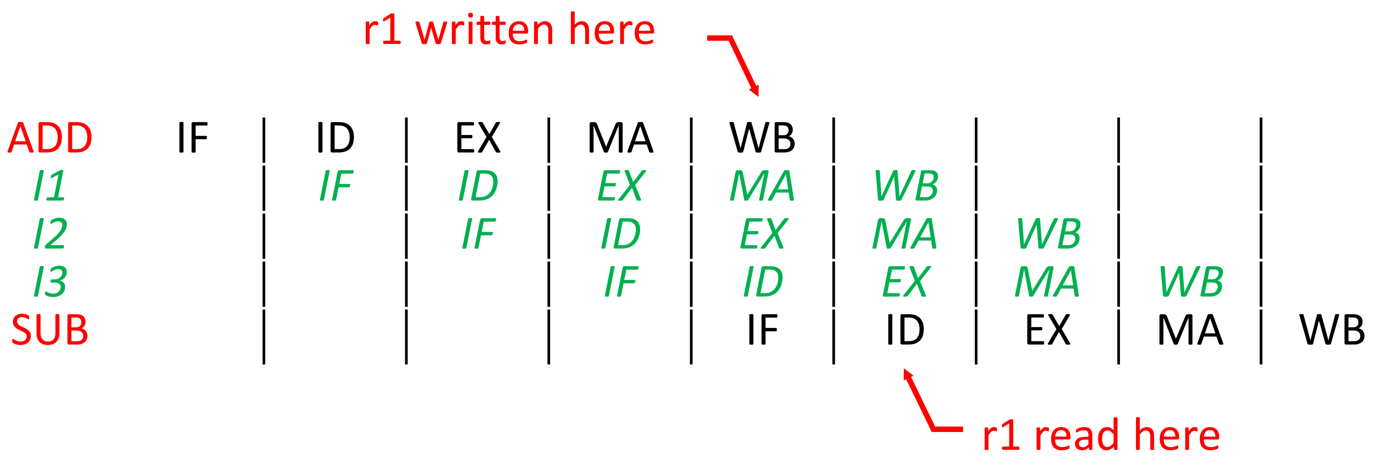


ADD	IF	ID	EX	MA	WB	
SUB		IF	ID	EX	MA	WB

- problem solved in DLX by
  - pipeline forwarding [or bypassing] and...
  - two phase access to the register file

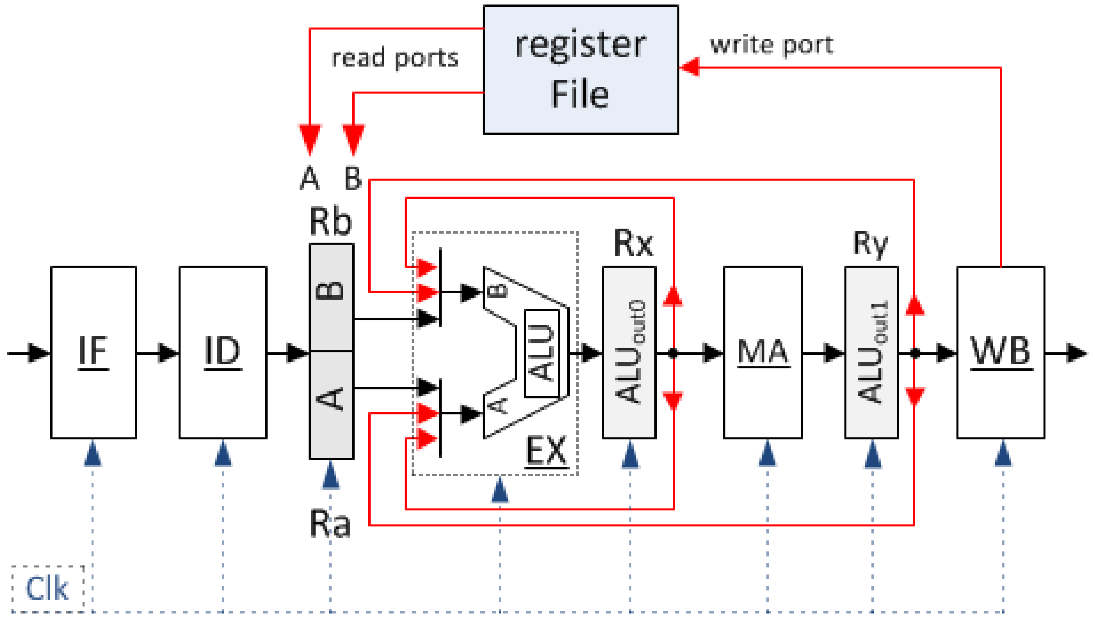
Data Hazards…

- alternative approach is to expose pipeline to programmers
- programmers would need to insert three instructions between ADD and SUB to get the *expected* result



ADD	IF	ID	EX	MA	WB				
I1		*IF*	*ID*	*EX*	*MA*	*WB*			
I2			*IF*	*ID*	*EX*	*MA*	*WB*		
I3				*IF*	*ID*	*EX*	*MA*	*WB*	
SUB					IF	ID	EX	MA	WB

Pipeline Forwarding



- registers between each pipeline stage R_a, R_b, ALU_{out0}, ALU_{out1} etc.
- all registers clocked synchronously

- the ALU results from the "previous" two instructions can be forwarded to the ALU inputs from the ALU_{OUT0} & ALU_{OUT1} pipeline registers before the results are written back to the register file

- tag ALU_{out0} and ALU_{out1} with the destination register
- EX stage checks for source register in order ALU_{out0}, ALU_{out1} and then A/B

Two Phase Clocking

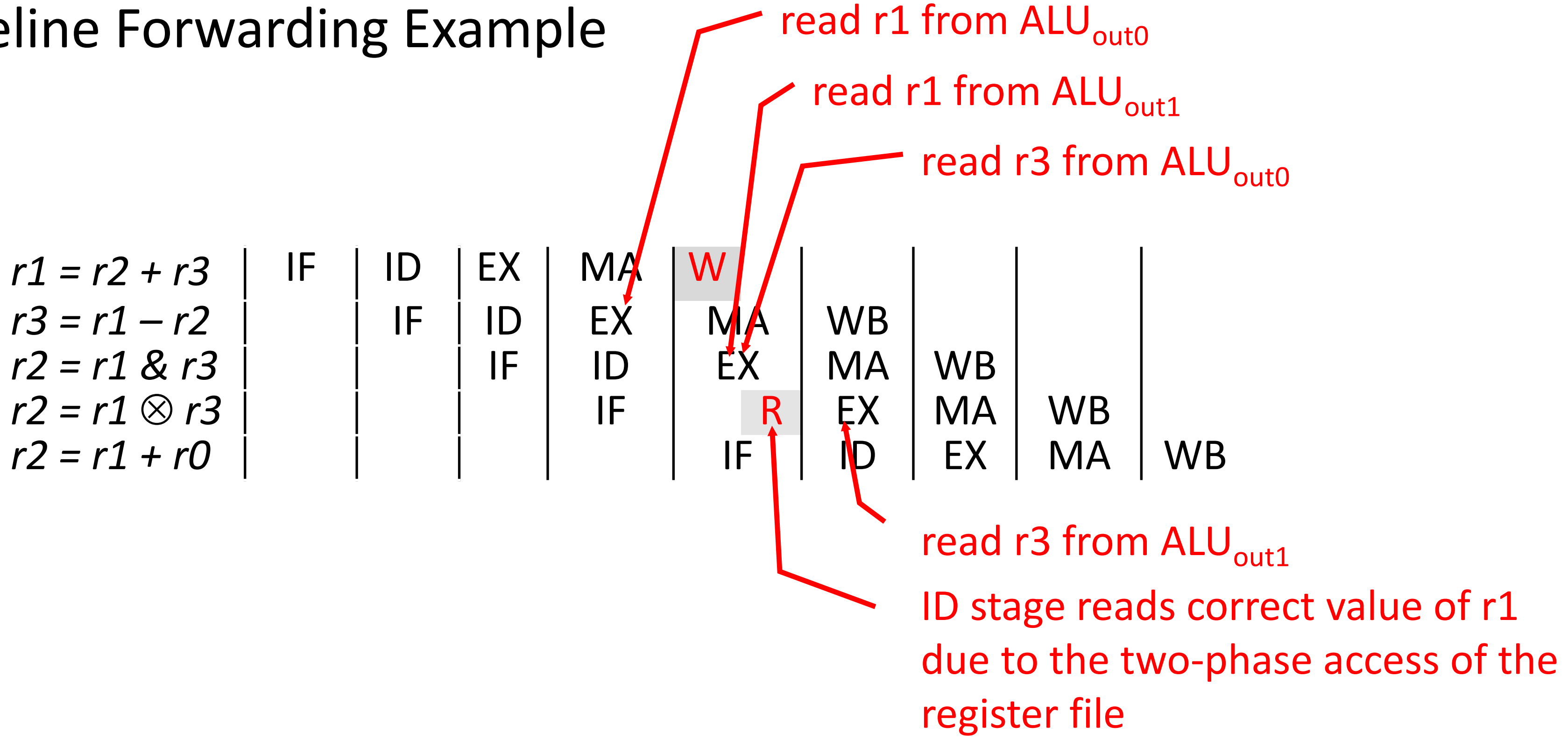
- DLX register file can be written then read in a single clock cycle
  - written during first half of cycle [WB phase]
  - read during second half of cycle [ID phase]
  - hence NO need for a third forwarding register [see slide 38]

WB and ID stages can access register file during same cycle

i	IF	ID	EX	MA	W				
i+1		IF	ID	EX	MA	WB			
i+2			IF	ID	EX	MA	WB		
i+3				IF	R	EX	MA	WB	
i+4					IF	ID	EX	MA	WB

RISC and Pipelining

Pipeline Forwarding Example


read r1 from ALU_{out0}
read r1 from ALU_{out1}
read r3 from ALU_{out0}
read r3 from ALU_{out1}
ID stage reads correct value of r1 due to the two-phase access of the register file

r1 = r2 + r3	IF	ID	EX	MA	W				
r3 = r1 – r2		IF	ID	EX	MA	WB			
r2 = r1 & r3			IF	ID	EX	MA	WB		
r2 = r1 ⊗ r3				IF	R	EX	MA	WB	
r2 = r1 + r0					IF	ID	EX	MA	WB

- first instruction writes to r1 and the next four instructions use r1 as a source operand
- second instruction writes to r3 which is used as a source operand by the third and fourth instructions

- NB: the *intelligence* is in the EX phase, not the ID phase

Load Hazards

- consider the following instruction sequence

```
r1 = M[a]          // load
r4 = r1 + r7       // add
r5 = r1 - r8       // subtract
r6 = r2 & r7       // and
```

dependency between load and ADD results in a one cycle pipeline stall

r1 = M[a]	IF	ID	EX	MA	WB				
r4 = r1 + r7		IF	ID	stall	EX	MA	WB		
r5 = r1 – r8			IF	IF	ID	EX	MA	WB	
r6 = r2 & r7					IF	ID	EX	MA	WB

- can't used result of load until data read from memory in MA phase
- a pipeline interlock occurs when a load hazard is detected, resulting in a pipeline stall
- loaded data must be forwarded to EX stage from ALU$_{out1}$
- could remove stall by moving "&" instruction and placing it between load and add
- often possible to reschedule instructions to avoid this type of pipeline stall

Instruction Scheduling Example

- consider the following instruction sequence where a .. f are memory locations

```
a ← b + c
d ← e - f
```

- compiler generated scheduled code would be as follows

```
r2 ← M[b]
r3 ← M[c]
r5 ← M[e]                ; swapped with add to avoid stall
r1 ← r2 + r3
r6 ← M[f]
M[a] ← r1                ; load/store swapped to avoid stall in sub
r4 ← r5 - r6
M[d] ← r4
```

- access to many registers critical for a legal schedule
- pipeline scheduling generally increases registers usage

DLX Pipeline Operation

- register transfer description
- ALU instructions

IF	IR $\leftarrow$ M[PC]; PC $\leftarrow$ PC+4
ID	A $\leftarrow$ R_{SRC1}; B $\leftarrow$ R_{SRC2}; PC1 $\leftarrow$ PC; IR1 $\leftarrow$ IR
EX	ALU_{OUT0} $\leftarrow$ result of ALU operation
MA	ALU_{OUT1} $\leftarrow$ ALU_{OUT0}
WB	R_{DST} $\leftarrow$ ALU_{OUT1}

- Load/Store instructions

IF	IR $\leftarrow$ M[PC]; PC $\leftarrow$ PC+4
ID	A $\leftarrow$ R_{SRC1}; B $\leftarrow$ R_{DST}; PC1 $\leftarrow$ PC; IR1 $\leftarrow$ IR
EX	MAR $\leftarrow$ effective address; SMDR $\leftarrow$ B
MA	LMDR $\leftarrow$ M[MAR] or M[MAR] $\leftarrow$ SMDR
WB	R_{DST} $\leftarrow$ LMDR

DLX Pipeline Operation…

- BNEZ/BEQZ instructions [conditional branch]

IF	IR $\leftarrow$ M[PC]; PC $\leftarrow$ PC+4
ID	A $\leftarrow$ R_{SRC1}; B $\leftarrow$ R_{SRC2}; PC1 $\leftarrow$ PC; IR1 $\leftarrow$ IR
EX	ALU_{OUT0} $\leftarrow$ PC1+ offset; cond $\leftarrow$ R_{SRC1} op 0
MA	if (cond) PC $\leftarrow$ ALU_{OUT0}
WB	idle

Control Hazards

- a simple DLX branch implementation results in a 3 cycle stall per branch instruction

branch	*IF*	*ID*	*EX*	*MA*	WB						
i1		IF	stall	stall	IF	ID	EX	MA	WB		
i2						IF	ID	EX	MA	WB	
i3							IF	ID	EX	MA	WB

  - new PC not known until the end of MA
  - 3 cycle penalty whether branch is taken or NOT

- a 30% branch frequency and a 3 cycle stall results in ONLY ≈ 50% of the potential pipeline speed up [consider 100 instructions: non-pipelined 500; perfectly pipelined 100; 3 cycle branch stall 30 x 4 + 70 = 190]

- need to (1) determine if branch is taken or not taken earlier in pipeline and (2) compute target address earlier in pipeline

DLX Branches

- DLX doesn't have a conventional condition code register
- uses a "set conditional" instruction followed by a BEQZ or BNEZ instruction
  - sets register with 0 or 1 depending on the comparison of two source operands

```
SLT     r1, r2, r3  ; r1 = (r2 < r3) ? 1 : 0
BEQZ    r1, L       ; branch to L if (r1 == 0)
```

  - also SGT, SLE, SGE, SEQ and SNE [NB: also need unsigned comparisons]
- may need additional instructions compared with an instruction set where instructions implicitly set the condition codes

DLX Branches…

- add additional hardware to resolve branches during the ID stage
- test if a register == /!= 0 and adds offset to PC if condition true

IF	IR $\leftarrow$ M[PC]; PC $\leftarrow$ PC+4
ID	if (R_{SRC1} == / != 0) PC $\leftarrow$ PC + offset
EX	idle
MA	idle
WB	idle

- now a ONE cycle branch penalty

branch	*IF*	*ID*	*EX*	*MA*	*WB*				
i1		IF	IF	ID	EX	MA	WB		
i2				IF	ID	EX	MA	WB	
i3					IF	ID	EX	MA	WB

- stalls pipeline until branch target known

DLX Branches…

- improve further by assuming branch NOT taken
  - pipeline stalled ONLY if branch taken
  - must undo any side effects if branch taken [which should be minimal]

branch NOT taken	*IF*	*ID*	*EX*	*MA*	*WB*	
i1		IF	ID	EX	MA	WB
i2			IF	ID	EX	MA	WB

branch taken	*IF*	*ID*	*EX*	*MA*	*WB*		
i1		IF	IF	ID	EX	MA	WB
i2				IF	ID	EX	MA	WB

incorrect instruction fetched as branch taken

Comparison of DLX Branch Strategies

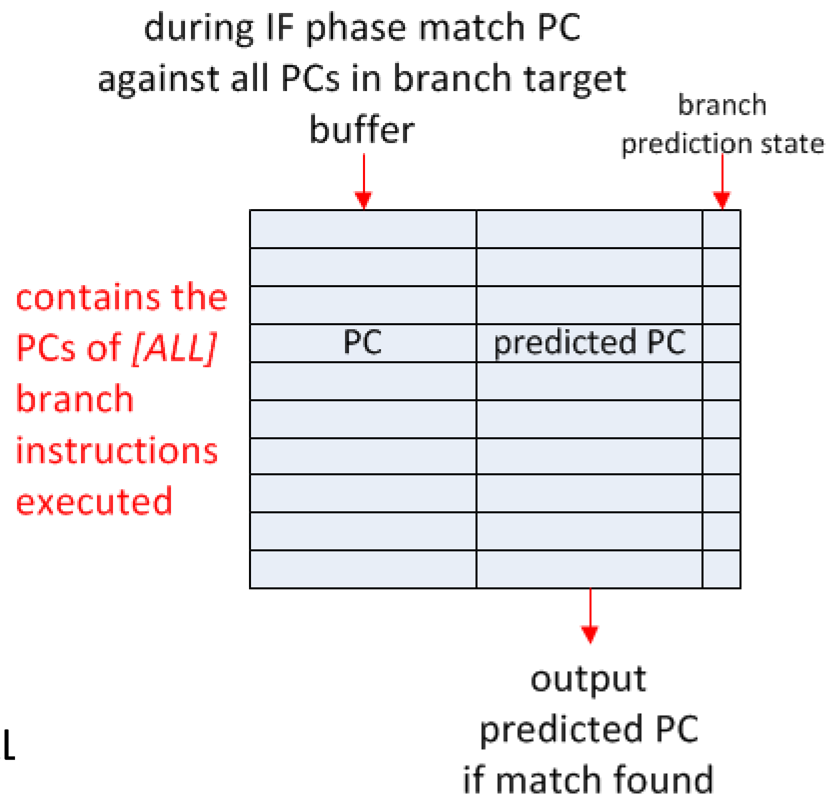
- also compare with delayed branches [as per RISC-I and SPARC]
- assume
  - 14% branch instructions
  - 65% of branches change PC
  - 50% probability of filling branch delay slot with a useful instruction

method	branch penalty	effective clocks per instruction (CPI)
stall pipeline 3	3	0.86 + 0.14 x 4 = 1.42
stall pipeline 1	1	0.86 + 0.14 x 2 = 1.14
predict NOT taken	1	0.86 + 0.14 x (0.35 + 2 x 0.65) = 1.09
delayed branch	0.5	0.86 + 0.14 x 1.5= 1.07

RISC AND PIPELINING

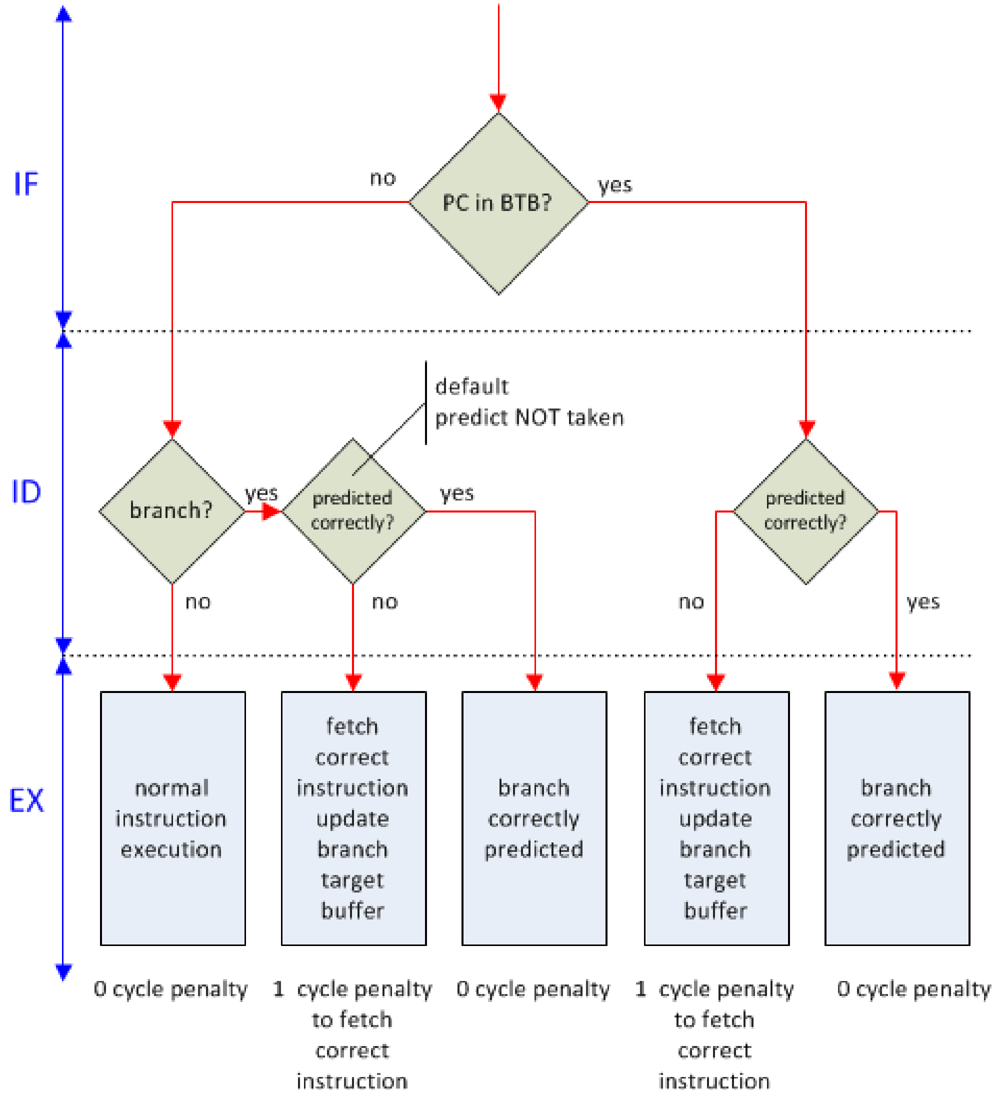
Branch Prediction

- need to resolve branches during IF phase
- use a branch target buffer [BTB]
- during IF fetch, look for PC in branch target buffer [compare against all PCs in parallel]
- if match found, use predicted PC to fetch next instruction
- if branch <u>correctly</u> predicted, NO pipeline STALL
- if branch <u>incorrectly</u> predicted, must "abort" fetched instruction and fetch correct one [pipeline stalled for one clock cycle]
- must update BTB if a *new* branch fetched or prediction changes


during IF phase match PC against all PCs in branch target buffer
branch prediction state
contains the PCs of *[ALL]* branch instructions executed
PC
predicted PC
output predicted PC if match found

Branch Prediction Flowchart

- flowchart showing the execution of a branch instruction
- 1 cycle penalty to fetch correct instruction if incorrect instruction fetched [ie. branch incorrectly predicted]
- BTB is a cache [has a limited capacity – may need to replace entries]
- need ONLY place taken branches in BTB as following instruction will be fetched by default anyway
- NB: default *predict NOT taken*


IF
ID
EX
PC in BTB?
no
yes
default
predict NOT taken
branch?
yes
no
predicted correctly?
yes
no
predicted correctly?
no
yes
normal instruction execution
fetch correct instruction update branch target buffer
branch correctly predicted
fetch correct instruction update branch target buffer
branch correctly predicted
0 cycle penalty
1 cycle penalty to fetch correct instruction
0 cycle penalty
1 cycle penalty to fetch correct instruction
0 cycle penalty

Two Bit Branch Prediction

- consider the following loop

```
        add     r0, #10, r1
L1:     …
        …
        sub     r1, #1, r1
        bnez    r1, L1
```

- assume BTB empty and predict that a branch will branch the same way as it did last time
- first time bnez executed, predicted incorrectly [as default prediction branch NOT taken]
- next 8 times, bnez predicted correctly [predict taken, branch taken]
- next time, bnez predicted incorrectly [predict taken, branch NOT taken]
- now assume branch remains in BTB and loop executed again
- bnez will be predicted incorrectly [predict NOT taken, branch taken] and so on…
- 80% prediction accuracy

- a two bit scheme which changes prediction only when prediction is incorrect twice in succession gives a 90% prediction accuracy [with this example]

Branch Prediction Analysis

- assume
  - 14% branch instructions
  - 90% probability of branch hitting the branch target buffer [as it has a finite capacity]
  - 90% probability of a correct prediction
  - 1 cycle penalty if branch target buffer needs updating
- branch penalty calculation
  - penalty = %hits x %mispredictions x 1 + %misses x 1
  - penalty = (0.9 x 0.1 x 1) + (0.1 x 1)
  - penalty = 0.19 cycle
  - branches take 1.19 cycles
  - as 14% branches results in 1.0266 effective clocks per instruction [CPI]
- compares favourably with delayed branching [1.07]

Summary

- you are now able to:
  - outline the history of RISCs

  - outline the design criteria and architecture of the RISC-1 microprocessor
  - analyse the operation and limitations of register windows
  - analyse the operation and motivation for delayed jumps
  - develop simple RISC-1 assembly language programs

  - describe the operation of the 5 stage DLX/MIPS pipeline
  - explain how data and load hazards are resolved
  - analyse a number of different approaches for handling control hazards
  - explain how branch prediction is implemented

  - use a DLX/MIPS pipeline simulator
  - predict the number of clock cycles needed to execute DLX/MIPS code segments

RISC AND PIPELINING