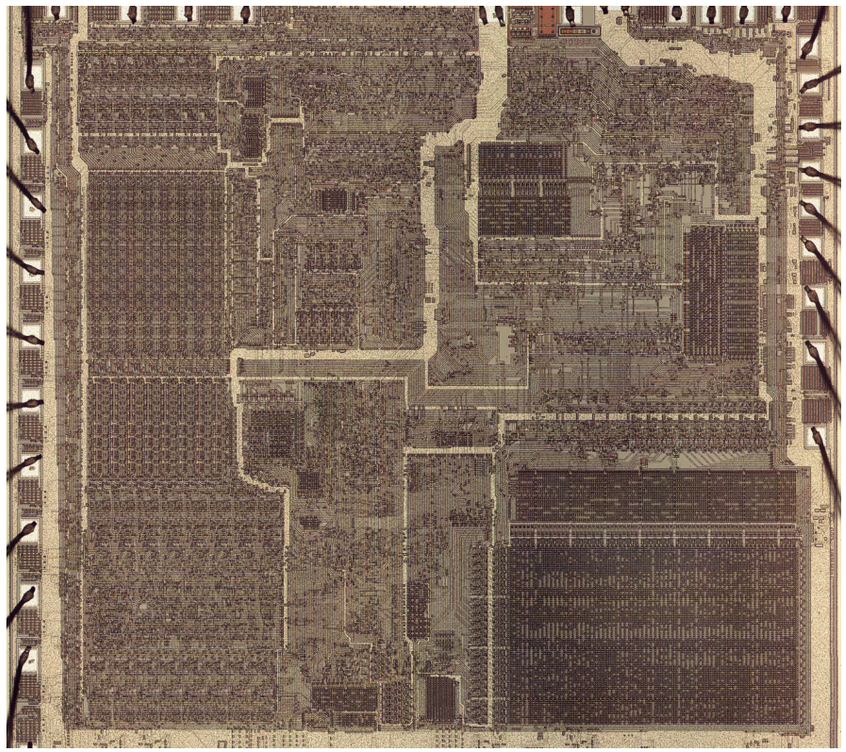
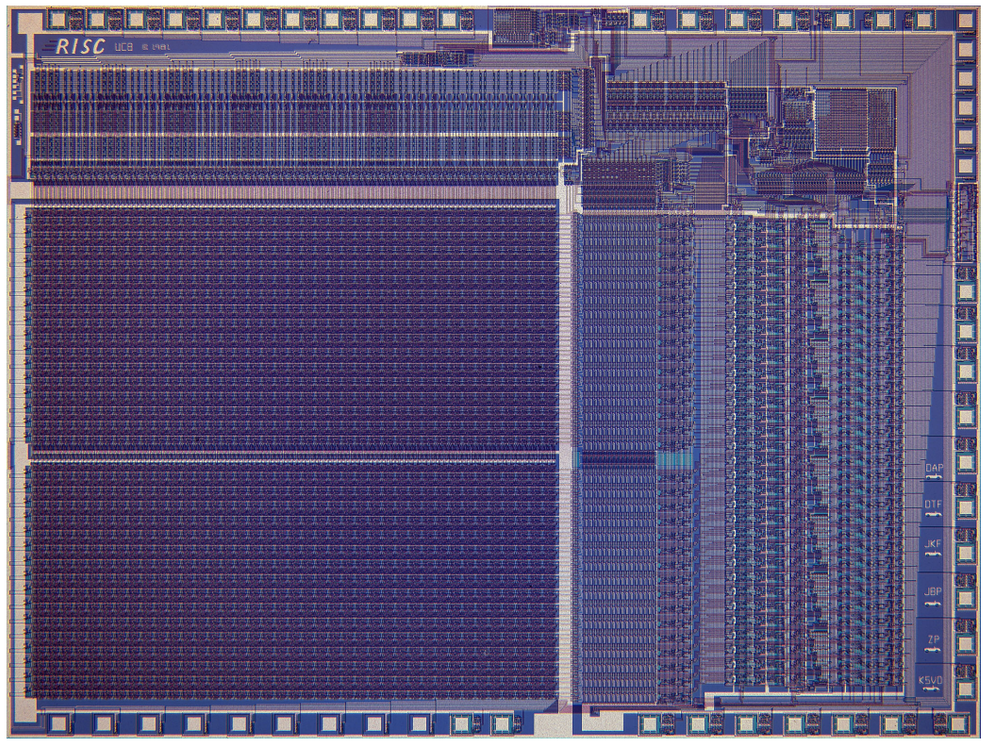
Reduced Instruction Set Computer

CSU34021 - Computer Architecture II

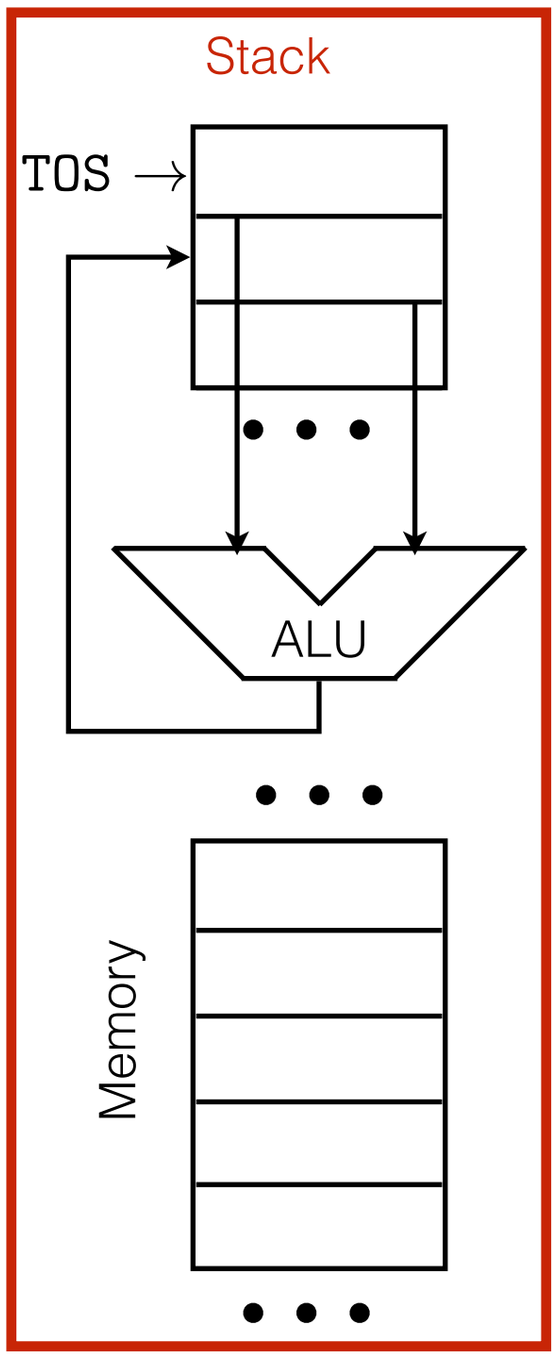
Some Background

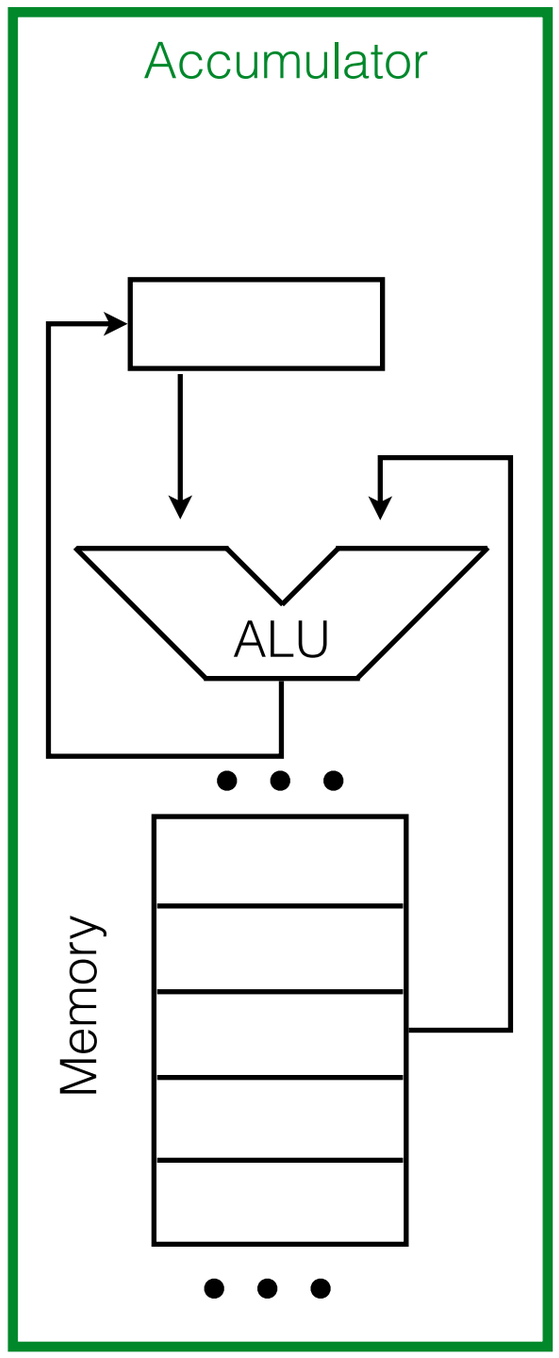
- One of the main reason of popularity for x86 and x64 is that they are backward compatible, all the way back to the 8086 (i.e., to 1976!).
- However this is also one of the main drawbacks of their design… Hardware and software technology changed a lot in the last 50 years.
- What made sense at the time doesn't necessarily make sense today… [High level languages vs. assembly; compilers; instruction cache; Component scaling; Software design principles...]

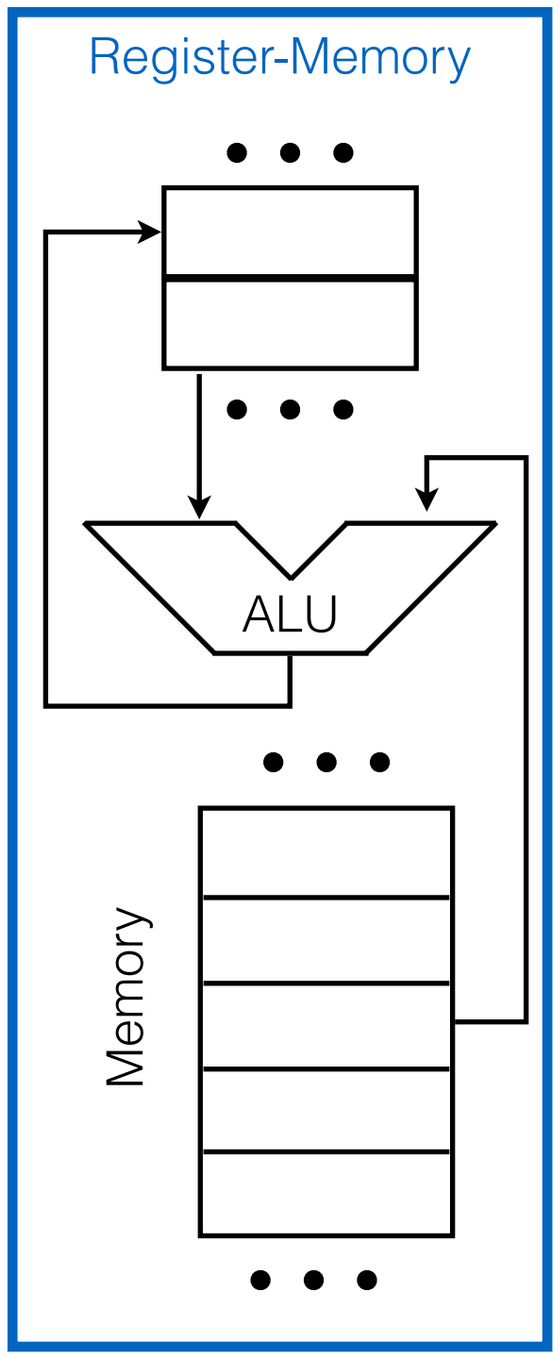


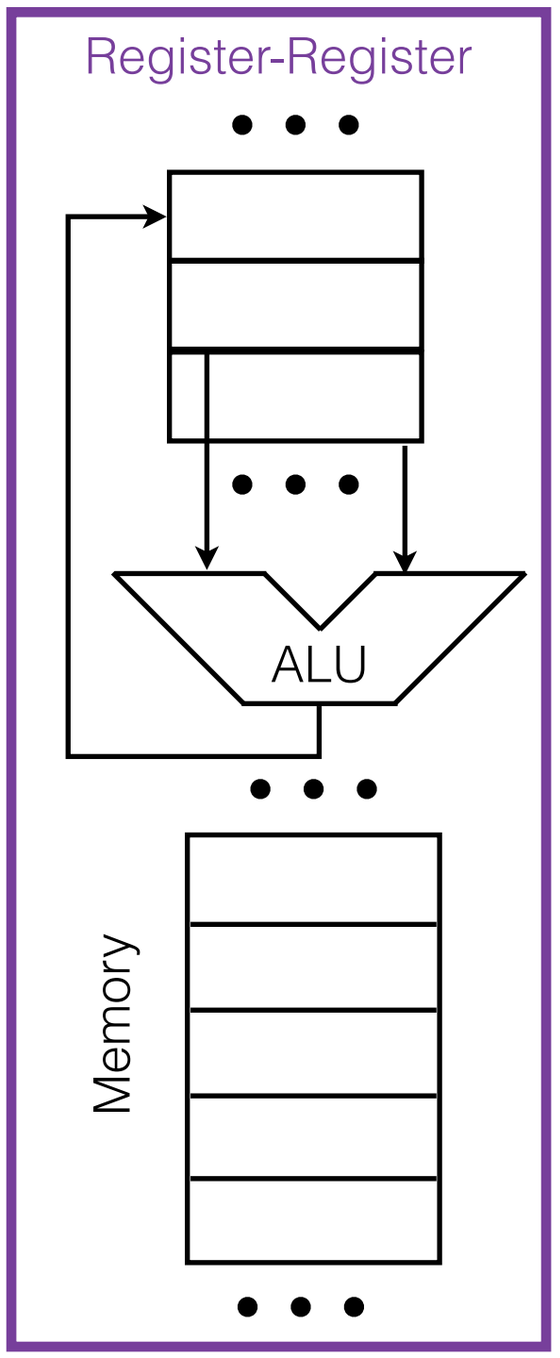


Example: Architecture's Internal Storage









Example: Architecture's Internal Storage

- Virtually all recent CPUs nowadays do Register-Register and/or Register/Memory… Why?

Example: Architecture's Internal Storage

- Virtually all recent CPUs nowadays do Register-Register and/or Register/Memory… Why?

- With only a few registers the x86 relies heavily on main memory.

- We have seen how x64 extends x86 with more registers, and how that is useful for speeding up function calls. [Technology, software design, compilers!].

- What else has changed since?

RISC - Reduced Instruction Set Computer

- Developed in the 80s to exploit new technology.
- General design principle: Only allow for simple instructions and addressing modes. Every operation to require the same cycles [optimising the average case and reducing chip complexity!]
- Only access memory with load/store operations.
- Need to increase the number of registers!
- A bit of a misnomer…

RISC vs CISC

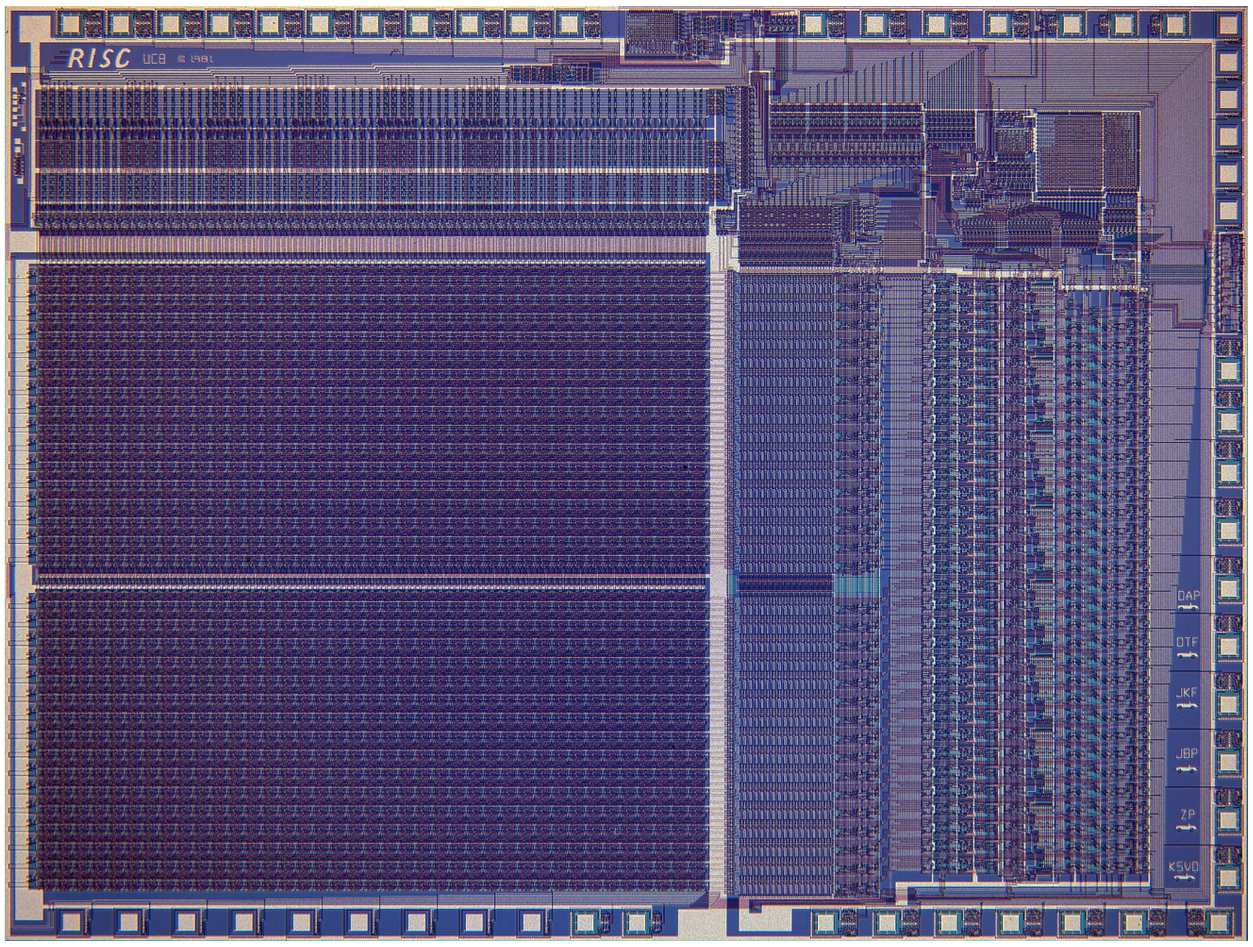
- CISC (Complex Instruction Set Computer) architectures try to minimise the number of instructions and give more high-level commands for the assembly.

- RISC architectures try to minimise the number of clock cycles needed by each instruction and simplifies decoding, at the cost of increasing the number of instructions needed to perform an operation.

- The distinction is not always so marked as it was at early stages. Many architectures nowadays implement design principles from both [modern implementation of x64 actually use RISC and only give the appearance of the x64 ISA to the programmer through microprogramming].

RISC-1

- Designed by MSc students led by D. Patterson and C.H. Séquin.
- Report: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/CSD-82-106.pdf.



RISC-1 Design Principles

- One Instruction per cycle [can't allow memory instruction! Why?]
- All instructions to be of the same size [Simplifies decoding!]
- Main memory access through simple load/store instructions.
- Only one addressing mode.
- Compiler in mind!
- Optimising the most frequent case, ensuring correctness for other cases.

Most frequent case?

Table 1.
Relative Frequency of HLL Statements.
(ordered by memory references)

statements	HLL (# occurrence)		WEIGHTED (# instr.)		WEIGHTED (# mem. ref.)	
HLL	P	C	P	C	P	C
call/return	12±1	12±5	30±3	33±14	43±4	45±19
loops	4±0	3±1	40±3	32±6	32±2	26±5
assign	36±5	38±15	12±2	13±5	14±2	15±6
if	24±7	43±17	11±3	21±8	7±2	13±5
begin	20±1	-	5±0	-	2±0	-
with	4±1	-	1±0	-	1±0	-
case	1±1	<1±1	1±1	1±1	1±1	1±1
goto	-	3±1	-	0±0	-	0±0

- Quantitative analysis of common programmes.
- Function calls/returns account for majority of time and memory access! Extra care needed to account for this case.

RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

Operation code: Specifies one of 31 operations

RISC-1 Architecture

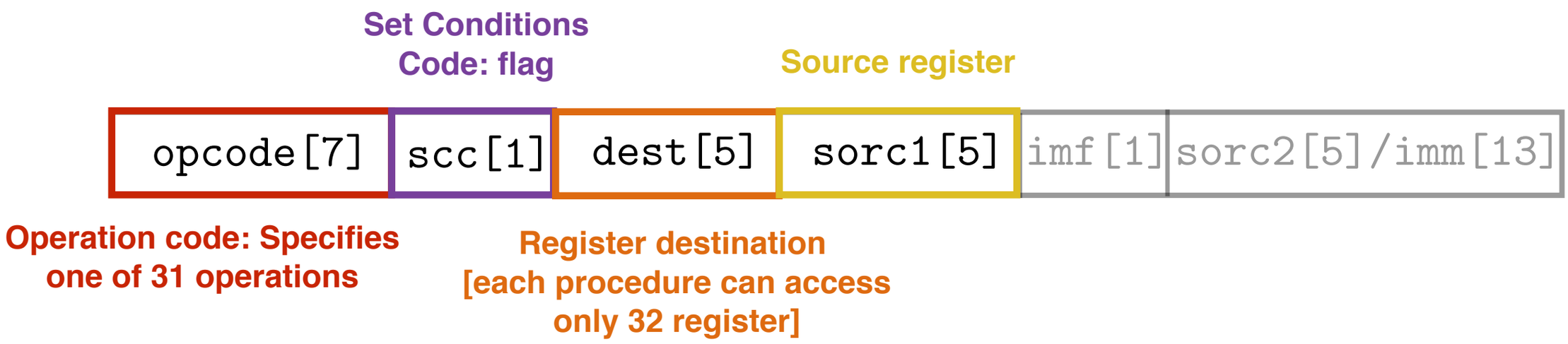
- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

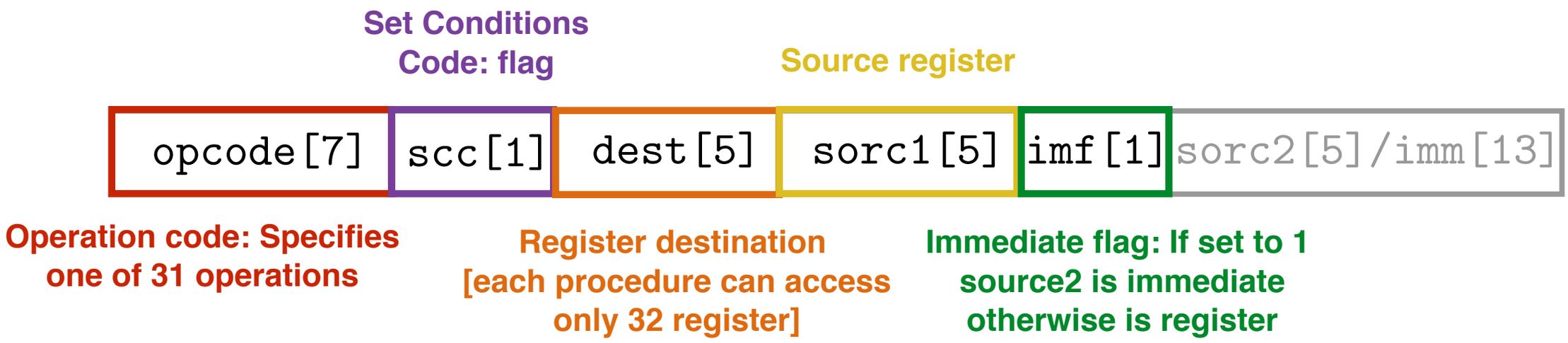
RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:



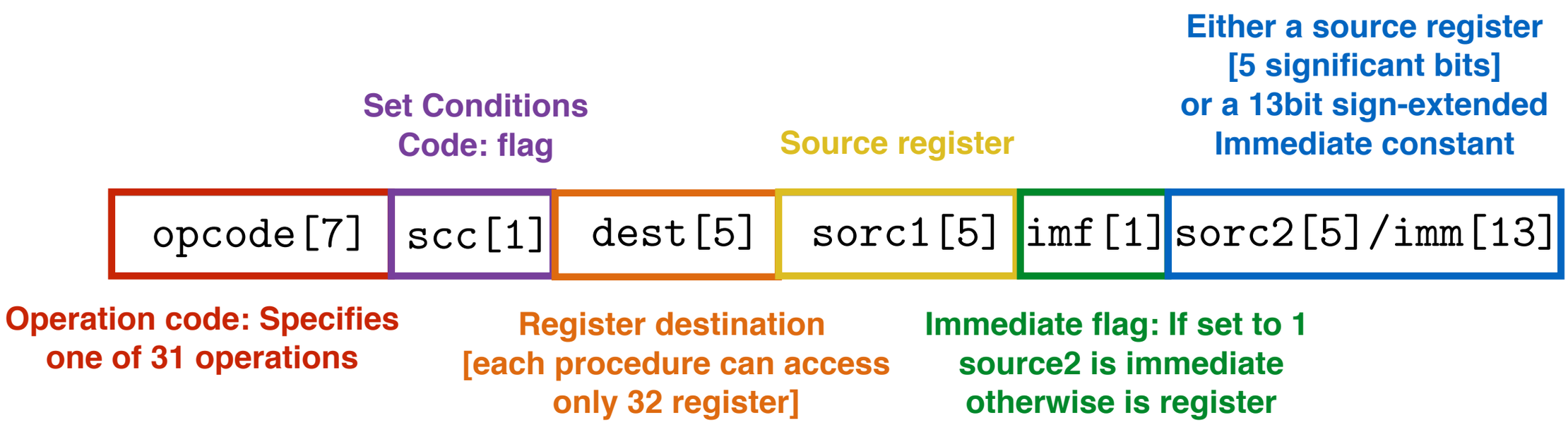
RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:



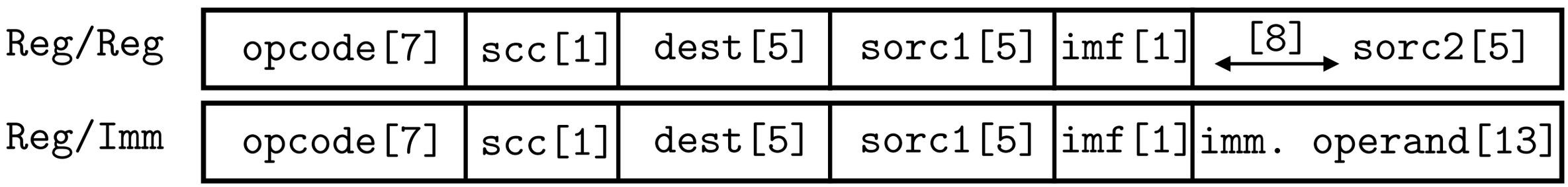
RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:



RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

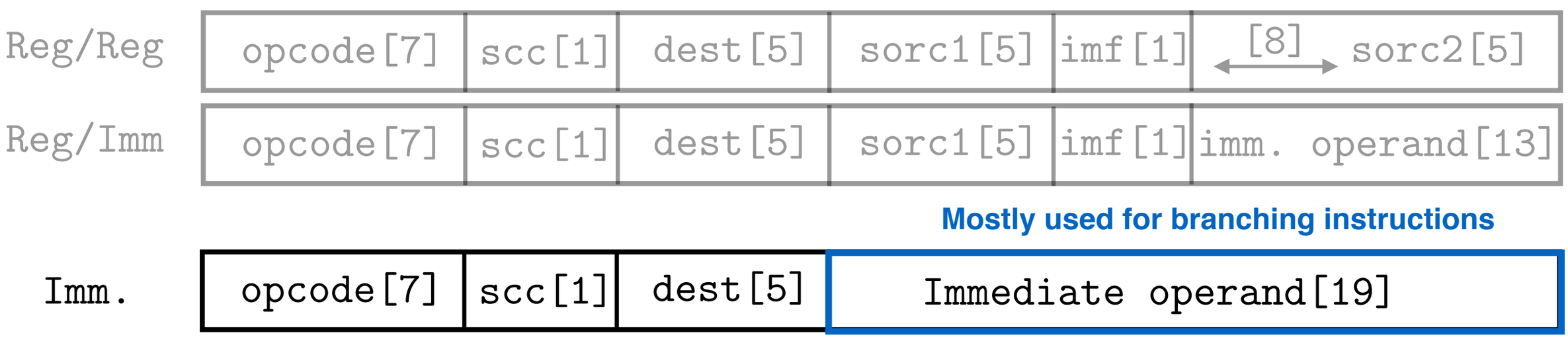


RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:

RISC-1 Architecture

- 138 32-bit registers [at least in theory] – r0 is hard-wired to 0 [each procedure assign a 32 register bank – more on this later].
- Additionally a set of Program Counter registers (PC), Program Status Word (PSW) and various control registers.
- Only 31 instruction in the RISC-1 ISA. All of them are 32-bit wide, with the following format:



Instruction Set

- Instruction set pruned to the essential [NB: RISC can have a lot of instructions, what really matter is their complexity…]

Arithmetic Operations

Load/Store Instructions

Call/Return & Jump Instructions

Instr.	Operands	Comments	
ADD	Rs,S2,Rd	Rd ← Rs + S2	integer add
ADDC	Rs,S2,Rd	Rd ← Rs + S2 + carry	add with carry
SUB	Rs,S2,Rd	Rd ← Rs - S2	integer subtract
SUBC	Rs,S2,Rd	Rd ← Rs - S2 - carry	subtract with carry
SUBR	Rs,S2,Rd	Rd ← S2 - Rs	integer subtract
SUBCR	Rs,S2,Rd	Rd ← S2 - Rs - carry	subtract with carry
AND	Rs,S2,Rd	Rd ← Rs & S2	logical AND
OR	Rs,S2,Rd	Rd ← Rs | S2	logical OR
XOR	Rs,S2,Rd	Rd ← Rs xor S2	logical EXCLUSIVE OR
SLL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift left
SRL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right logical
SRA	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right arithmetic
LDL	(Rx)S2,Rd	Rd ← M[Rx+S2]	load long
LDSU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short unsigned
LDSS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short signed
LDBU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte unsigned
LDBS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte signed
STL	(Rx)S2,Rm	M[Rx+S2] ← Rm	store long
STS	(Rx)S2,Rm	M[Rx+S2] ← Rm	store short
STB	(Rx)S2,Rm	M[Rx+S2] ← Rm	store byte
JMP	CON,S2(Rx)	pc ← Rx+S2	conditional jump
JMPR	CON,Y	pc ← pc + Y	conditional relative
CALL	S2(Rx),Rd	CWP - -; Rd ← pc, next pc ← Rx+S2	call reg.-indexed and change window
CALLR	Y,Rd	CWP - -; Rd ← pc, next pc ← pc + Y	call relative and change window
RET	(Rx)S2	pc ← Rx+S2, next CWP++	return, change window
RETINT	(Rx)S2	pc ← Rx+S2; next CWP++	also enable interrupts
CALLINT	Rd	CWP - -; Rd ← last pc	also disable interrupts
LDHI	Y,Rd	Rd<31:13>←Y; Rd<12:0>←0	load immediate high
GTLPC	Rd	Rd ← last pc	to restart delayed jump
GETPSW	Rd	Rd ← PSW	read status word
PUTPSW	Rm	PSW ← Rm	set status word

3-address machine!

ISAs Equivalence

- Can been shown that the RISC-1 ISA is as powerful as IA-32

IA-32 **RISC-1**

```
mov r_dst, r_src   →   add r0,    r_src, r_dst
not r_dst          →   xor r_dst, #-1,   r_dst
cmp r_dst, r_src   →   sub r_src, r_dst, r0,   {C}
                   →
                   •
                   •
                   •
```

Set SSC flag [hence condition codes]

ISAs Equivalence continued

IA-32		RISC-1
`test r_dst, r_src`	$\rightarrow$	`and r_src, r_dst, r_0, {C}`
`mov r_dst, 0`	$\rightarrow$	`add r_0, r_0, r_dst`
`inc r_dst`	$\rightarrow$	`add r_dst, #1, r_dst`

ISAs Equivalence continued

IA-32		RISC-1
`test r_dst, r_src`	$\rightarrow$	`and r_src, r_dst, r_0, {C}`
`mov r_dst, 0`	$\rightarrow$	`add r_0, r_0, r_dst`
`inc r_dst`	$\rightarrow$	`add r_dst, #1, r_dst`

- Loading constants that fit into src2 (i.e., 13 bits): $-2^{12} \leq N \leq 2^{12} - 1$

`mov r_dst, N` $\rightarrow$ `add r0, #N, r_dst`

Loading/Storing

```
ldl  (r_src1)src2, r_dst    ;r_dst=[r_src1 + src2] -> load long (32 bits)
ldsu (r_src1)src2, r_dst    ;r_dst=[r_src1 + src2] -> load short unsigned (16 bits)
ldss (r_src1)src2, r_dst    ;r_dst=[r_src1 + src2] -> load short signed (16 bits)
ldbu (r_src1)src2, r_dst    ;r_dst=[r_src1 + src2] -> load byte unsigned
ldbs (r_src1)src2, r_dst    ;r_dst=[r_src1 + src2] -> load byte signed

stl  (r_src1)src2, r_dst    ;[r_src1 + src2]=r_dst -> store long
sts  (r_src1)src2, r_dst    ;[r_src1 + src2]=r_dst -> store short (lower 16 bits of r_dst)
stb  (r_src1)src2, r_dst    ;[r_src1 + src2]=r_dst -> store byte (lower 8 bits of r_dst)
```

- The load unsigned operators work by setting the higher bits to zero.
- The load unsigned operators work by sign-extending the values to the register.
- `src2` must be an immediate value.

Addressing Modes

- Registers and immediate can be accessed as per the instruction format.
- For the memory, only indexed addressing mode implemented directly. However other addressing modes can be derived:

Indexed:

```
ldl    (r_src1)src2, r_dst  //r_dst:=[r_src1+src2]
```

Direct:

```
ldl    (r0)src2, r_dst  //r_dst:=[src2]
```

Indirect:

```
ldl    (r_src1)#0, r_dst  //r_dst:=[r_src1]
```

Function calls

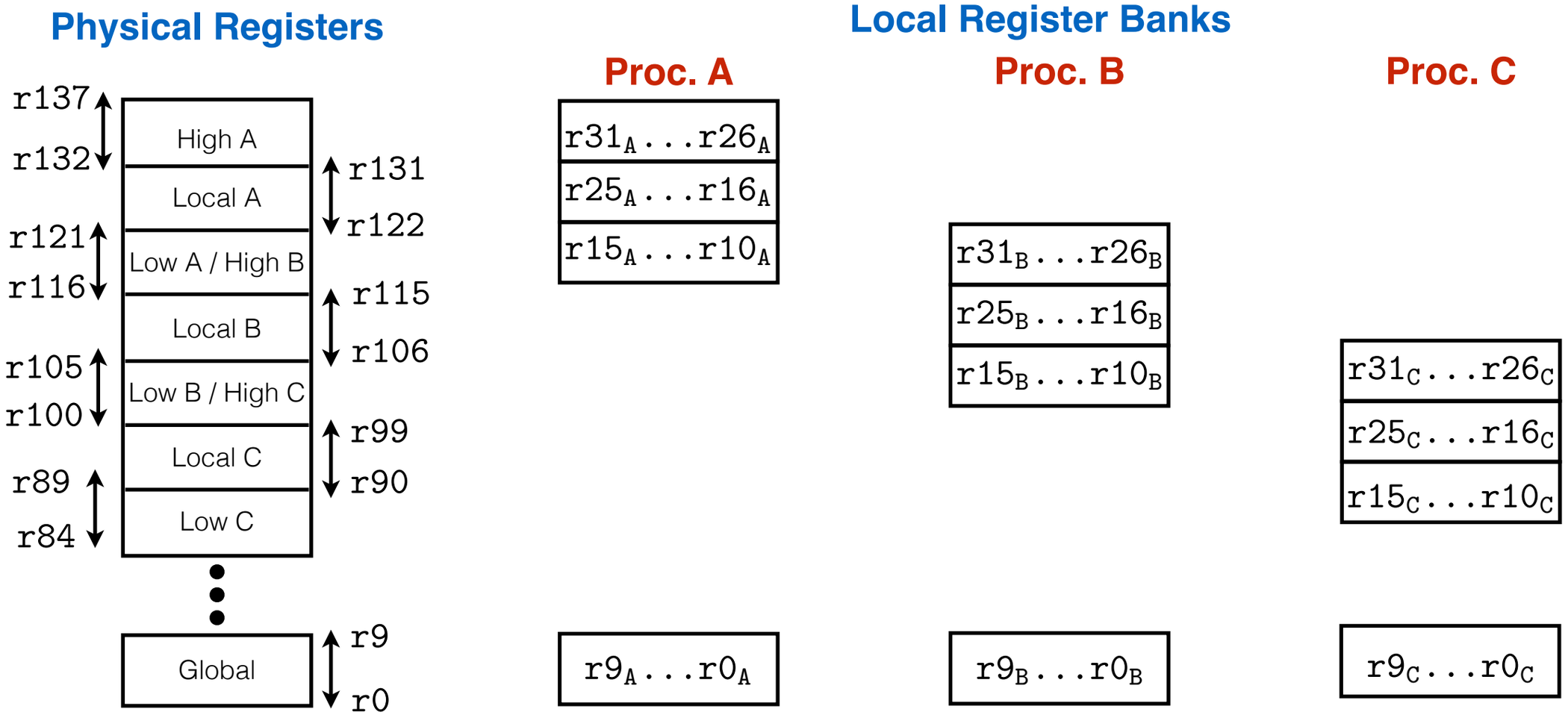
- Simplified control unit allows for more space in the chip (6% versus ~50% of most chips of the time).

- Can be used for more registers and to speed up function calls.

- 138 register [at least in theory… in practice only 78 – full set of registers in RISC-2].

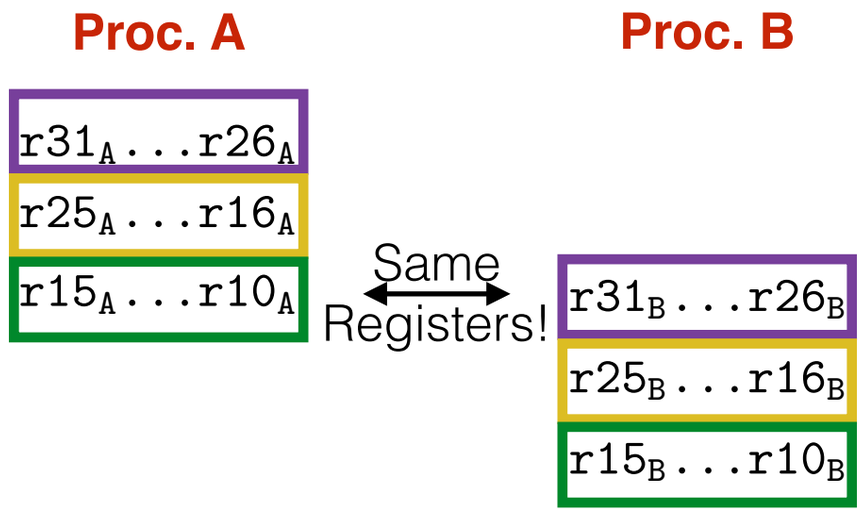
Register Banks

- Each procedure can access a bank of 32 registers. These are used for local variables and parameter passing.
- First 10 registers are global and shared across all the procedures [easy access to global variables e.g., pointers of big array and data structure].
- Overlapping of data banks for passing parameters.



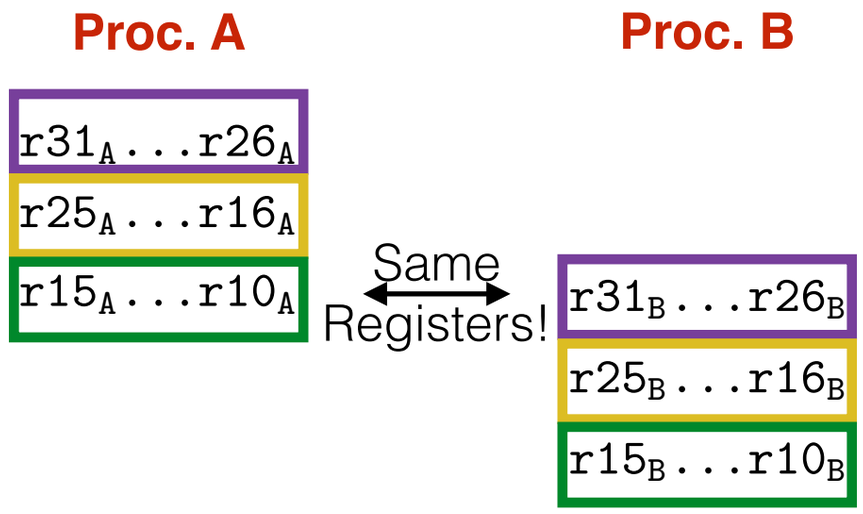
Parameter Passing

- Parameters passed to the current function are stored in **high** (r31 to r26) registers.
- Local variables stored in **middle** registers (r25 to r16).
- Parameters to pass to the next function are stored in the **lower** (r15 to r10) registers.
- No overhead for passing up to 6 parameters to functions! [Majority of the cases.]



Parameter Passing

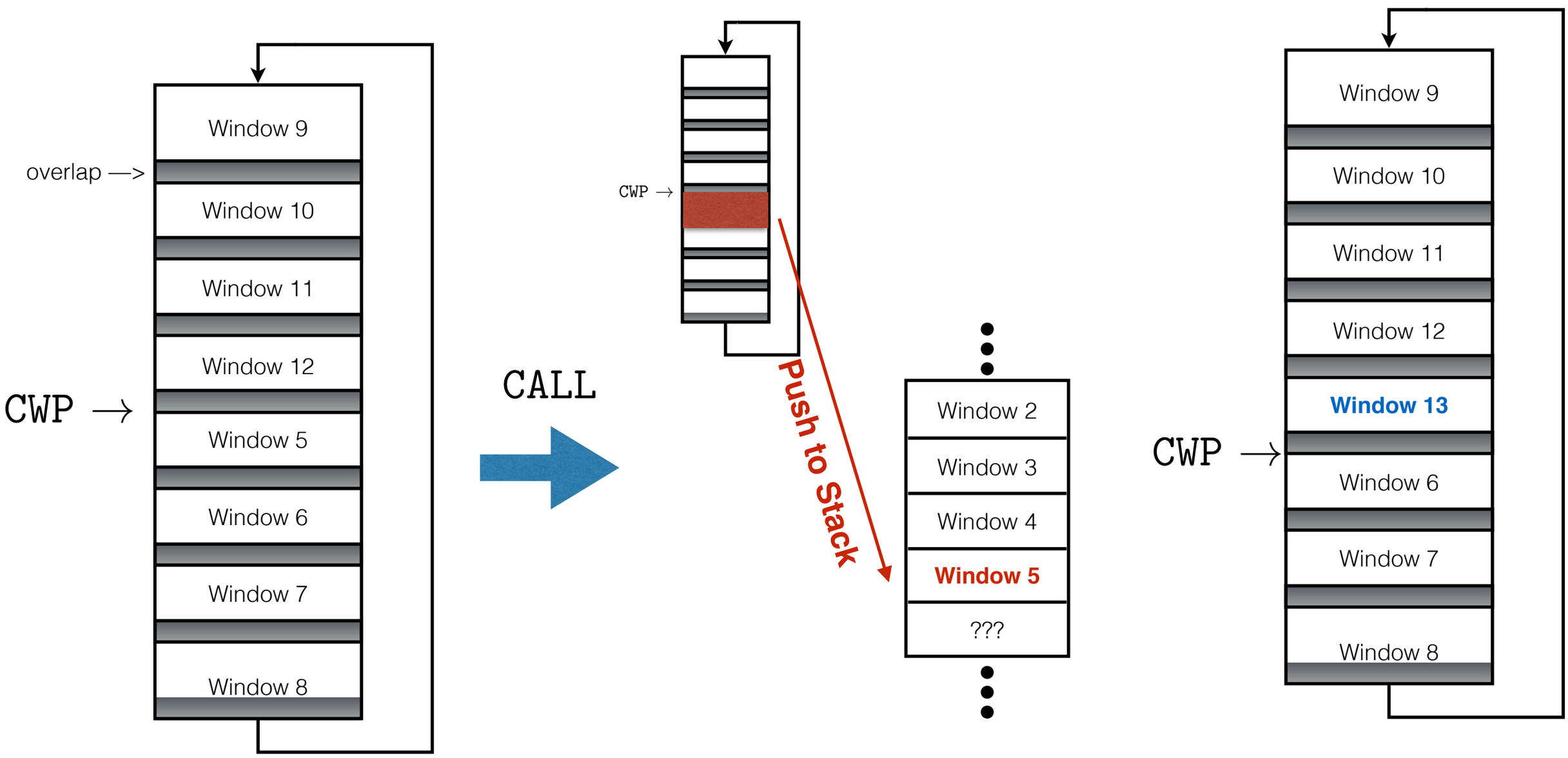
- Parameters passed to the current function are stored in **high** (r31 to r26) registers.
- Local variables stored in **middle** registers (r25 to r16).
- Parameters to pass to the next function are stored in the **lower** (r15 to r10) registers.
- No overhead for passing up to 6 parameters to functions! [Majority of the cases.]



- Even with 138 registers there are only 8 register banks, what happens when there are more than 8 nested function? (Very common!)

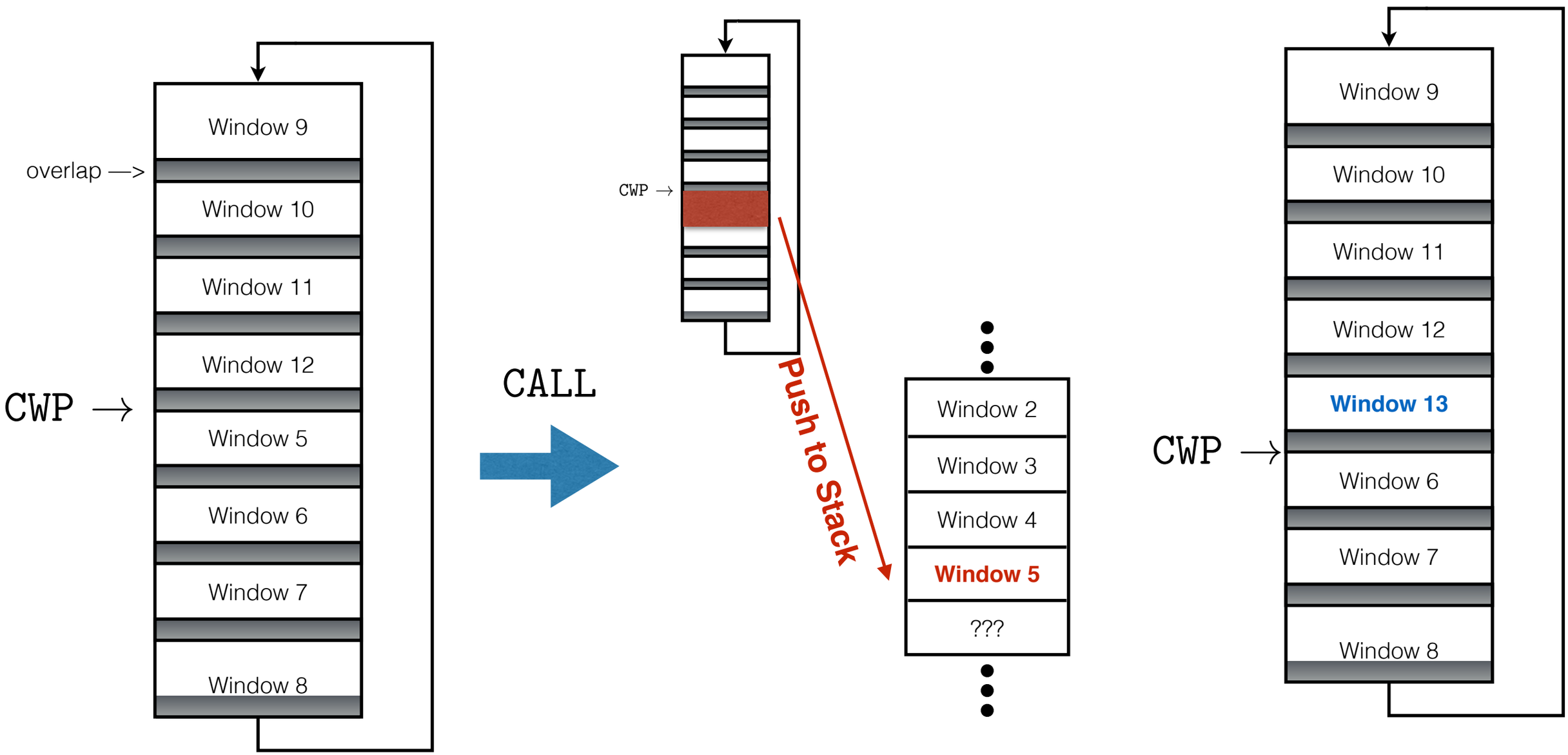
Register Overflow/Underflow

- Every time a new register bank needs to be allocated for a procedure, and there is no physical space for it (register overflow), the oldest allocated one gets stored into memory. Pointers used to keep track of locations.



Register Overflow/Underflow

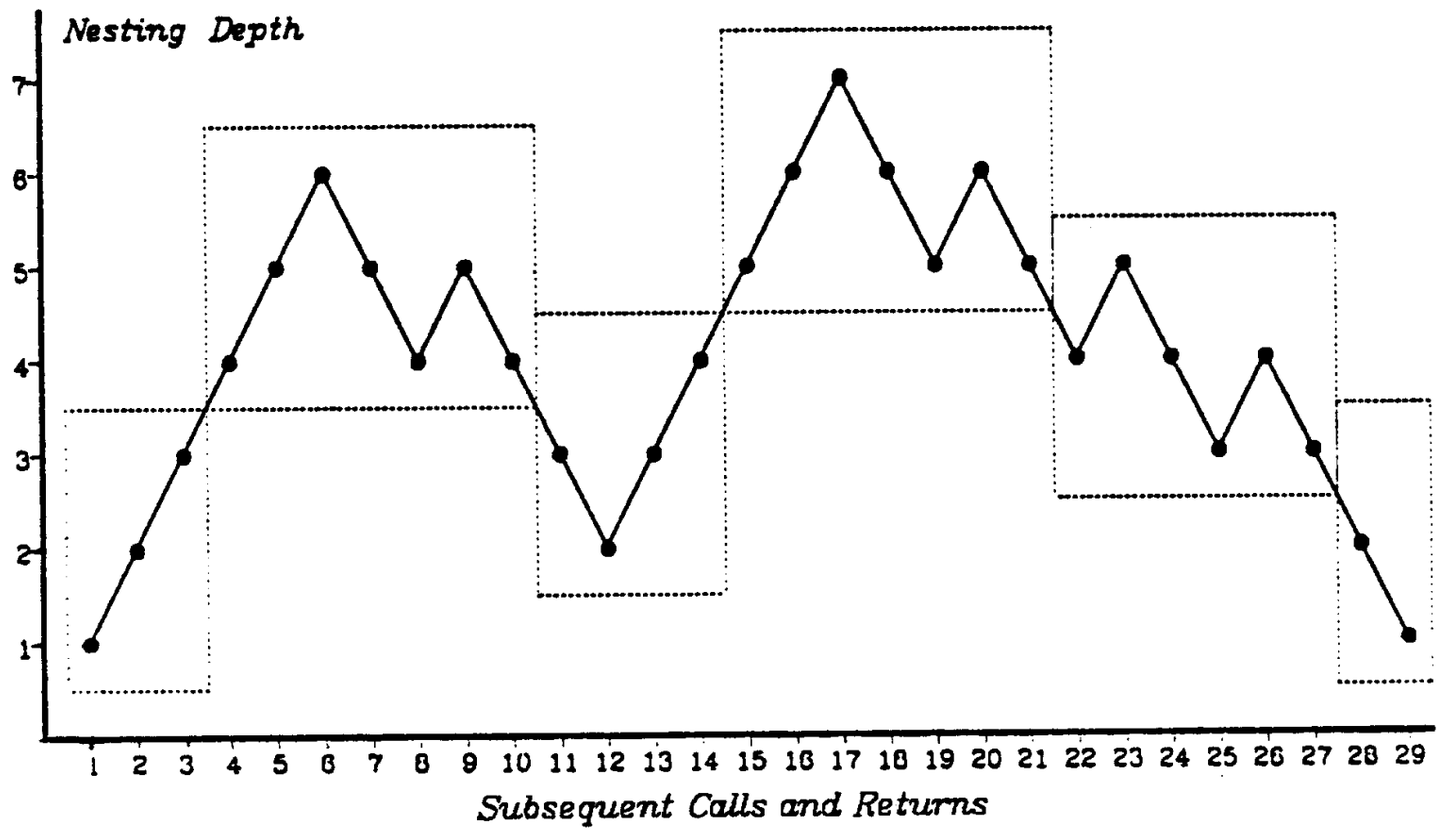
- Every time a new register bank needs to be allocated for a procedure, and there is no physical space for it (register overflow), the oldest allocated one gets stored into memory. Pointers used to keep track of locations.



- Upon return, when needed, the register bank is retrieved from memory and put into one of the physical register banks (register underflow).

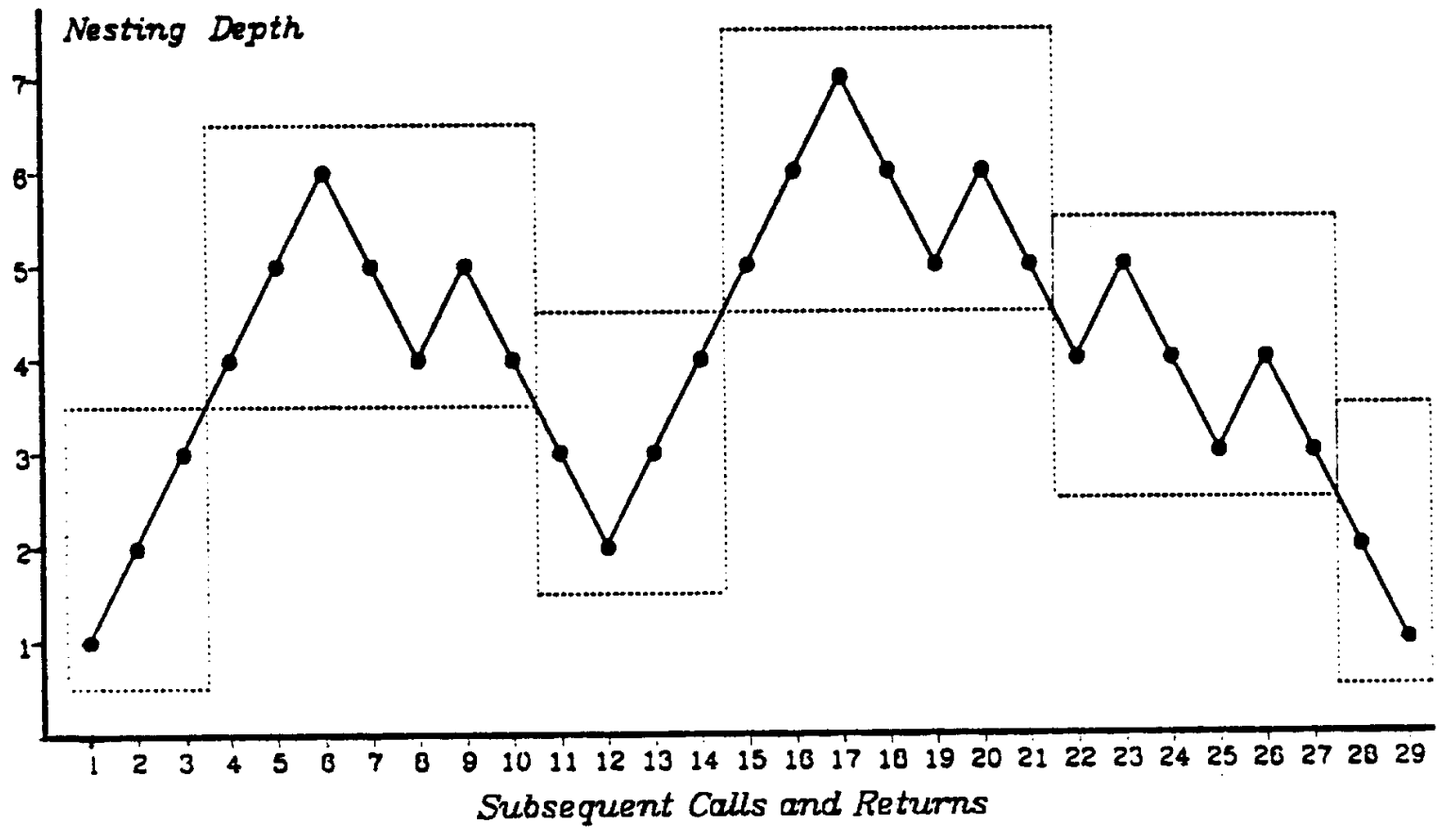
Register Overflow/Underflow and Performances

- Typical programmes have more than 8 nested function calls… Doesn't overflow/underflow defeat the whole purpose?



Register Overflow/Underflow and Performances

- Typical programme have more than 8 nested function calls… Doesn't overflow/underflow defeat the whole purpose?



- No! What matter is not the absolute depth, but the relative one! In typical programme overflow/underflow happen only 1% of the time.

Function Call

- Call indexed instruction:
  - Syntax: `call (r_src1)src2, r_dst`
  - What it does:

```
CWP    ← CWP+1          ;update to next reg. window
r_dst  ← PC             ;ret. addr. stored in r_dst
PC     ← r_src1+src_2   ;fun. start addr.
```

- Call relative instruction:
  - Syntax: `callr r_dst, src`
  - What it does:

```
CWP    ← CWP+1    ;update to next reg. window
r_dst  ← PC       ;ret. addr. stored in r_dst
PC     ← PC+src   ;rel jump to start addr. of fun.
```

Function Return

- Return instruction:
  - Syntax: `ret (r_dst)src2`
  - What it does:

```
CWP ← CWP-1          ;restore prev reg. window
PC  ← r_dst+src2     ;return addr. + const. offset
```

- The destination register must be the same across call/callr and ret.

Fetch and Execution Overlapping in RISC-1

- Simple instructions on RISC take one cycle. By overlapping fetch and execution throughput can be doubled:

Without overlapping:

```
cycle 1    |  cycle 2    |  cycle 3    |  cycle 4
fetch i1   |  execute i1 |             |
           |             |  fetch   i2 | execute i2
```

With overlapping:

```
cycle 1    |  cycle 2    |  cycle 3    |  cycle 4
fetch i1   |  execute i1 |             |
           |  fetch i2   |  execute i2 |
           |             |  fetch   i3 | execute i3
           |             |             | fetch i4
```

- Example of two-stage pipeline [more on general pipelining later on…]

RISC-1 Fetch-Execute: Memory Access

- However fetch-execution overlapping raises a few problems…
- Memory cannot be accessed twice in the same cycle! Hence you can't fetch and execute a load/store operation at the same time.

```
    |  cycle 1      |  cycle 2      |  cycle 3      |  cycle 4
i1  |  fetch load   |  comp. addr.  |  mem. access  |
i2  |               |  fetch i2     |  stall        |  execute i2
i3  |               |               |               |  fetch i3
i4  |               |               |               |
```

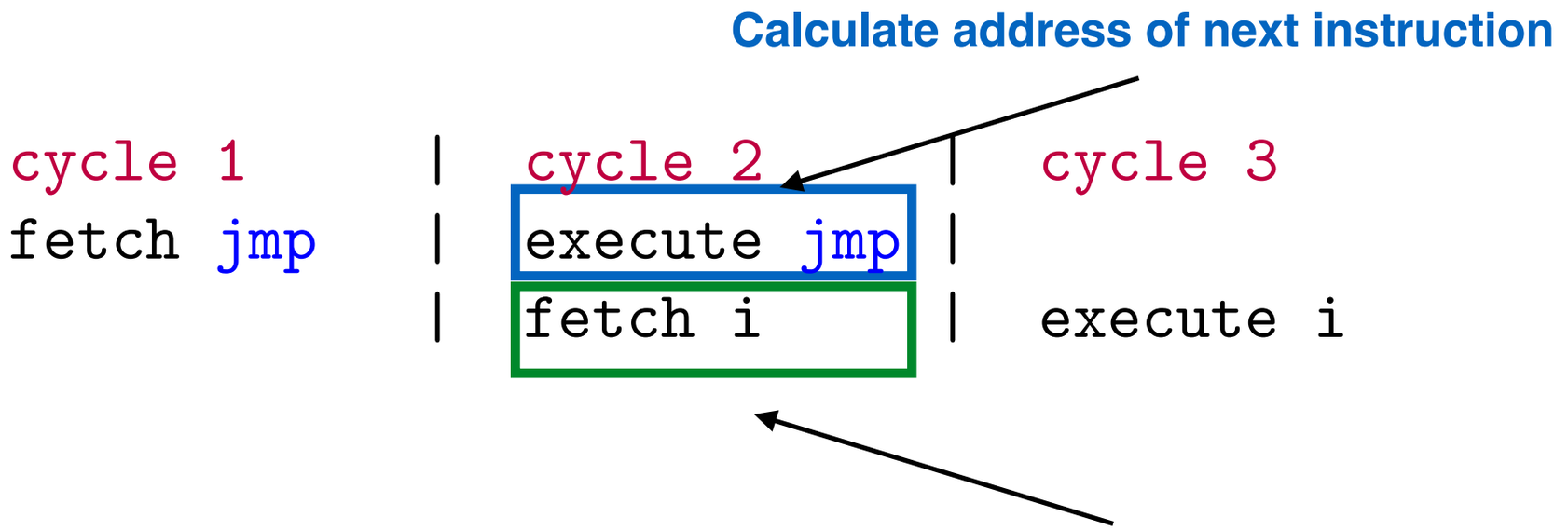
Risc-1 Pipelining – Jumps?

- What happens with jumps (conditional jumps, function calls/ returns)?

```
cycle 1      |  cycle 2      |  cycle 3
fetch jmp    |  execute jmp  |
             |  fetch i      |  execute i
```

Risc-1 Pipelining – Jumps?

- What happens with jumps (conditional jumps, function calls/ returns)?



- Can't execute jmp/ret/call (i.e. calculate address of next instruction), and fetch next instruction at the same time in a single cycle.

Delayed Jumps

- RISC-1 circumvent the issue by implementing delayed jumps [we will study more elaborate solution later on…].

- In RISC-1, jumps effectively take place only after the following instruction is executed.

- This is done by placing a NOP instruction after the jump.

- To avoid the wasted computation, the compiler implemented in the original work, first tries to rearrange instruction to do something useful after the jump!

Easy Example of Delayed Jump

- jmp, ret, call effectively take place after the following instruction.

```
1       sub r15, #1, r0, {C}   ;comparing r15 with 1
2       jne L                  ;conditional jump
3       add r0, r0, r15
4       add r1, r2, r1
...
9   L:  add r0, r1, r15
```

- If conditional jump taken, then effective execution order will be: 1,3,2,9…
- If conditional jump not taken, then: 1,3,2,4…

Delayed Jumps: Naive Example

```
//j input parameters to the function, i and k local variables
i = 0;
while(i < 10){
    i = i + j;
}
k = 0;
```

Just adding a NOP after the jump instruction.

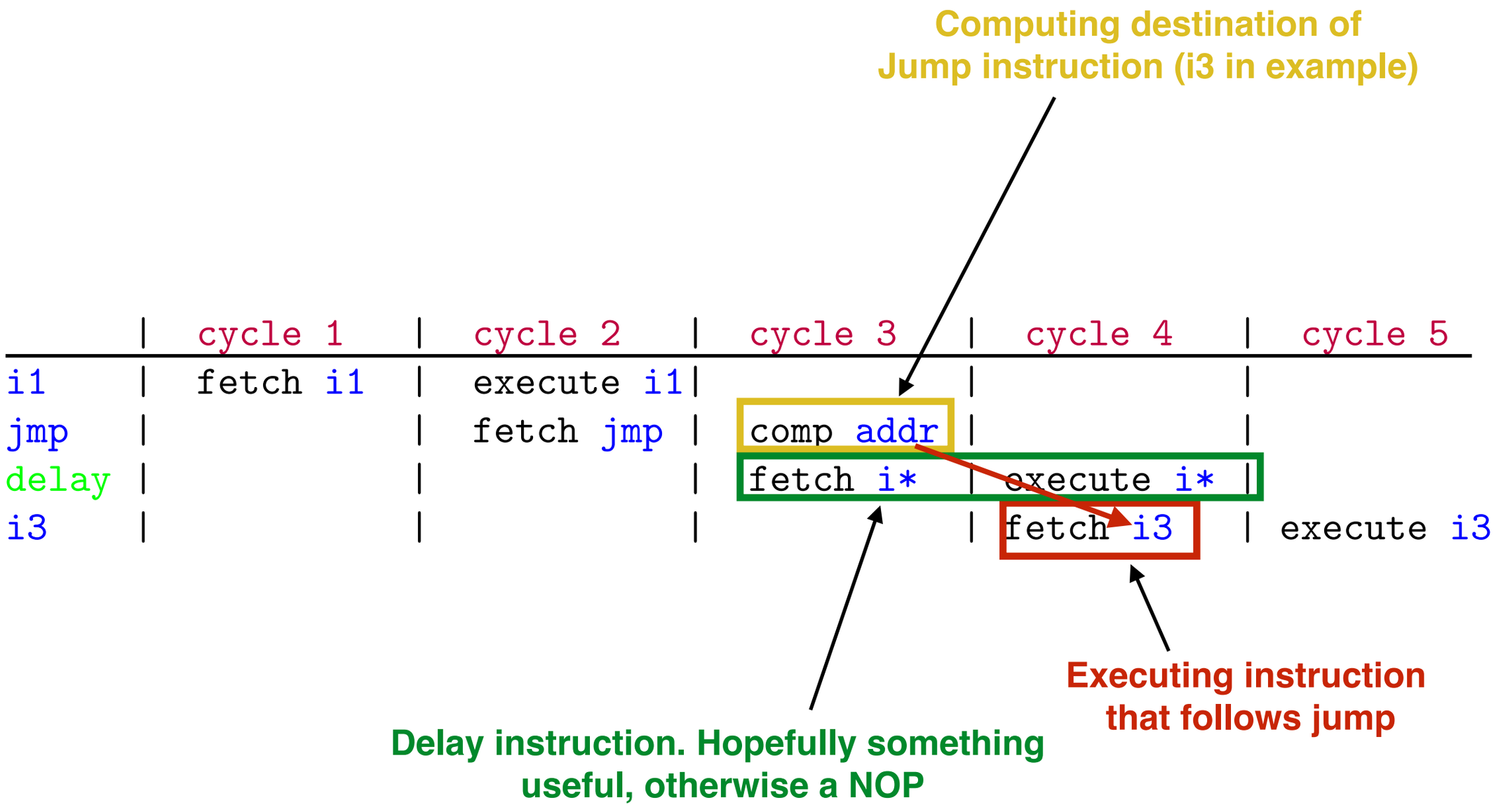
```
//The input parameter j is passed through high register r31
//Local variables i and k are saved in local registers r25 and r24
    add r0,  r0,  r25
loop:
    sub r25, #10, r0, {C} //checking i<10
    jge exit_loop
    add r0,  r0,  r0      //NOP
    add r25, r31, r25
    jmp loop
    add r0,  r0,  r0      //NOP
exit_loop:
    add r0,  r0,  r24    //Put k in the local regs.
```

Delayed Jumps: Optimised Example

```
//The input parameter j is passed through high register r31
//Local variables i and k are saved in local registers r25 and r24
    add r0,  r0,  r25
loop:
    sub r25, #10, r0, {C} //checking i<10
    jge exit_loop
    add r0,  r0,  r24     //Can do k=0 here already!
    jmp loop
    add r25, r31, r25     //Can perform i update while next address is being calculated
exit_loop:
    ...
```

- Got rid of all the NOP in this toy example.
- In practice useful instructions can be found in 60% of the cases occurring in real world code!

Delayed jump execution



- Trying to do something useful that doesn't arise conflicts in I*. Worst case scenario nothing useful is done, but you would have to remain idle anyway in there.

RISC Summary

- We have briefly discussed the history of RISC architectures.
- We have talked about the historical motivations behind RISC and its comparison with CISC.
- The design philosophy behind RISC.
- We have also seen in more details RISC-1, a practical implementation of a RISC architecture.
- Finally we have seen a first example of pipelining in RISC