Reduced Instruction Set Computers (RISC)

Chapter 15

Joannah Nanjekye

August 01, 2024

Key Advances in Computers

- The family concept
  - IBM System/360 1964
  - DEC PDP-8
  - Separates architecture from implementation
- Microprogrammed control unit
  - Idea by Wilkes 1951
  - Produced by IBM S/360 1964
  - Simplifies design and implementation of control unit
- Cache memory
  - BM S/360 model 85 1969

Key Advances in Computers

- Solid State RAM
  - See lecture slides on memory
- Microprocessors
  - Intel 4004 1971
- Pipelining
  - Introduces parallelism into fetch execute cycle
- Vector processing
  - Explicit parallelism
- Multiple processors
- RISC design

RISC

- Reduced Instruction Set Computer
  - A dramatic departure from historical architectures
- 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

CISC

- Complex Instruction Set Computer
- Complexity led to:
  - Large instruction sets
  - More addressing modes
  - Hardware implementations of HLL statements

CISC and RISC Processors

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

Instruction Execution Characteristics

1. Operations performed
   - Determine the functions to be performed by the processor and its interaction with memory
2. Operands used
   - The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them
3. Execution sequencing
   - Determines the control and pipeline organization
4. Semantic gap
   - The difference between the operations provided in HLLs and those provided in computer architecture
5. High-level languages (HLLs)
   - Convenience and abstraction for programmers

Operations

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

Frequency of HLL Operations

	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%

Operands

- Mainly local scalar variables
- Optimization priotize 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 operations
- Depends on number of parameters passed
  - Great majority use few parameters
  - 90% use three or fewer
- As well as Procedure invocation depth
  - Fairly shallow for most programs
- Most variables are local and scalar
  - cache or registers

Implications

- HLLs better supported by optimizing performance of the most time-consuming features
- Not close instruction set architectures

Operands and Registers

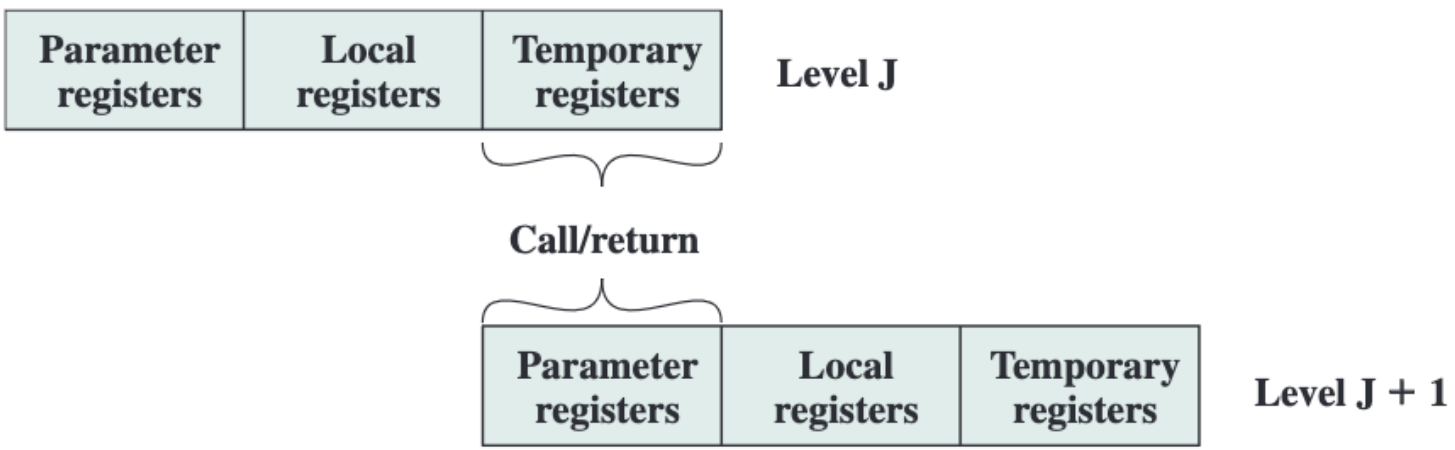
- Quick access to operands is desirable
  - Many assignment statements
  - Significant number of operand accesses per HLL statement
- Register storage is fastest available storage
- Addresses are much shorter than memory or cache
- Aim to keep operands in registers as much as possible

Large Register File

1. Hardware approach
   - Have more registers
   - Thus more variables will be in registers
2. Software approach
   - Require compiler to allocate registers
   - Allocate based on most used variables in a given time
   - Requires sophisticated program analysis to allocate registers efficiently
   - Can be very difficult on architectures such x86 where registers have special purposes

Register Windows

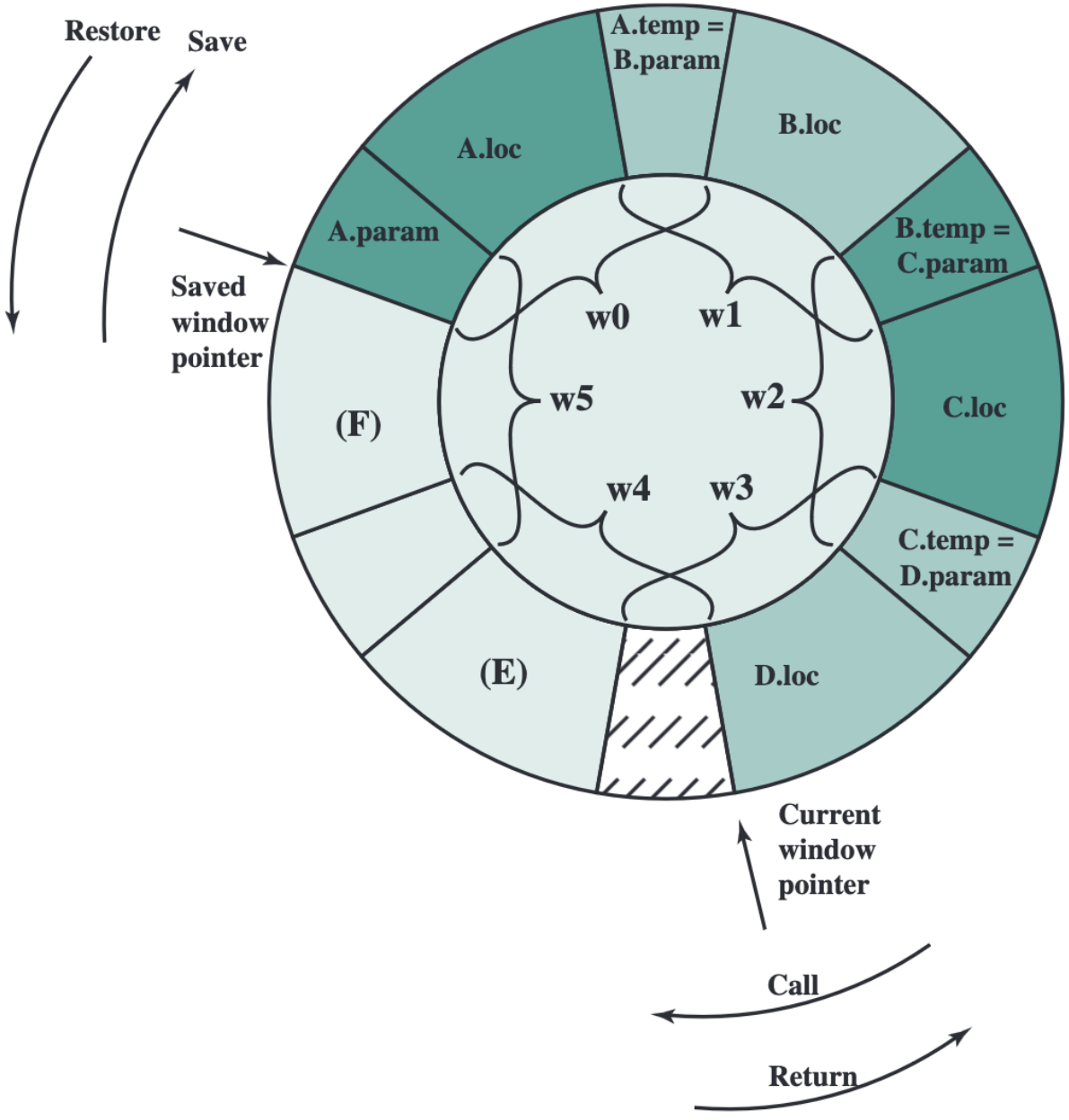
- Because most calls use only a few parameters and call depth is typically shallow
- One window of registers is visible and is addressable as if it were the only set of registers
- We can divide a register set into 3 areas:
  1. Parameter registers
  2. Local registers
  3. Temporary registers



Depth of Call Stack

- To handle any possible pattern of call and return, the number of register windows would have to be unbounded
- Instead, when call depth exceeds the number of available register windows, older activations have to be saved in memory and restored later when call depth decreases
- A circular buffer organization can make this reasonably efficient

Circular Buffer



Circular Buffer Operation

- When a call is made, a current window pointer is moved to show the currently active register window
- A saved window pointer indicates where the next saved window should restore
- When a CALL causes all windows to be in use (CWP is incremented and becomes equal to SWP), an interrupt is generated and the oldest window (the one furthest back in the call nesting) is saved to memory
- When a Return decrements CWP and it becomes equal to SWP an interrupt is generated and registers are restored from memory

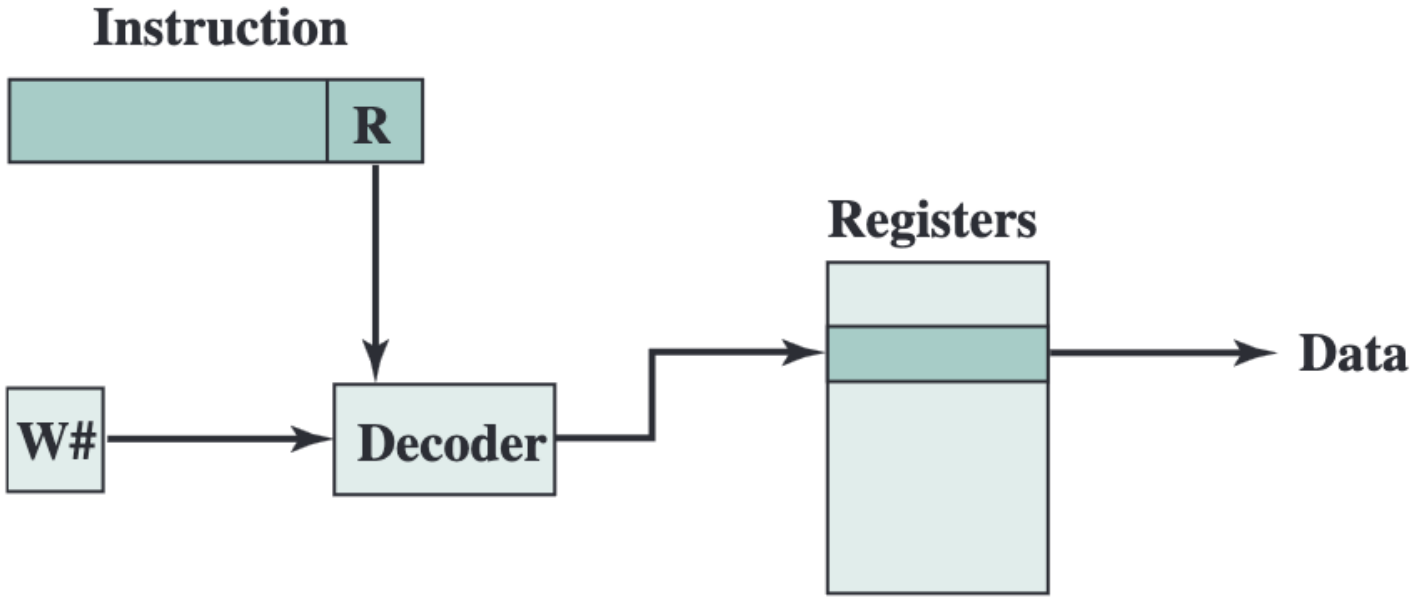
Global Variables

- Allocate all globals to memory and never use registers to store
  - Easy to implement
  - Inefficient for frequently accessed global variables
- Alternative is to incorporate a set of global registers in the processor
  - Increases hardware (handle register split) and compiler complexity (linker has to decide)
  - A few globals can be stored in registers

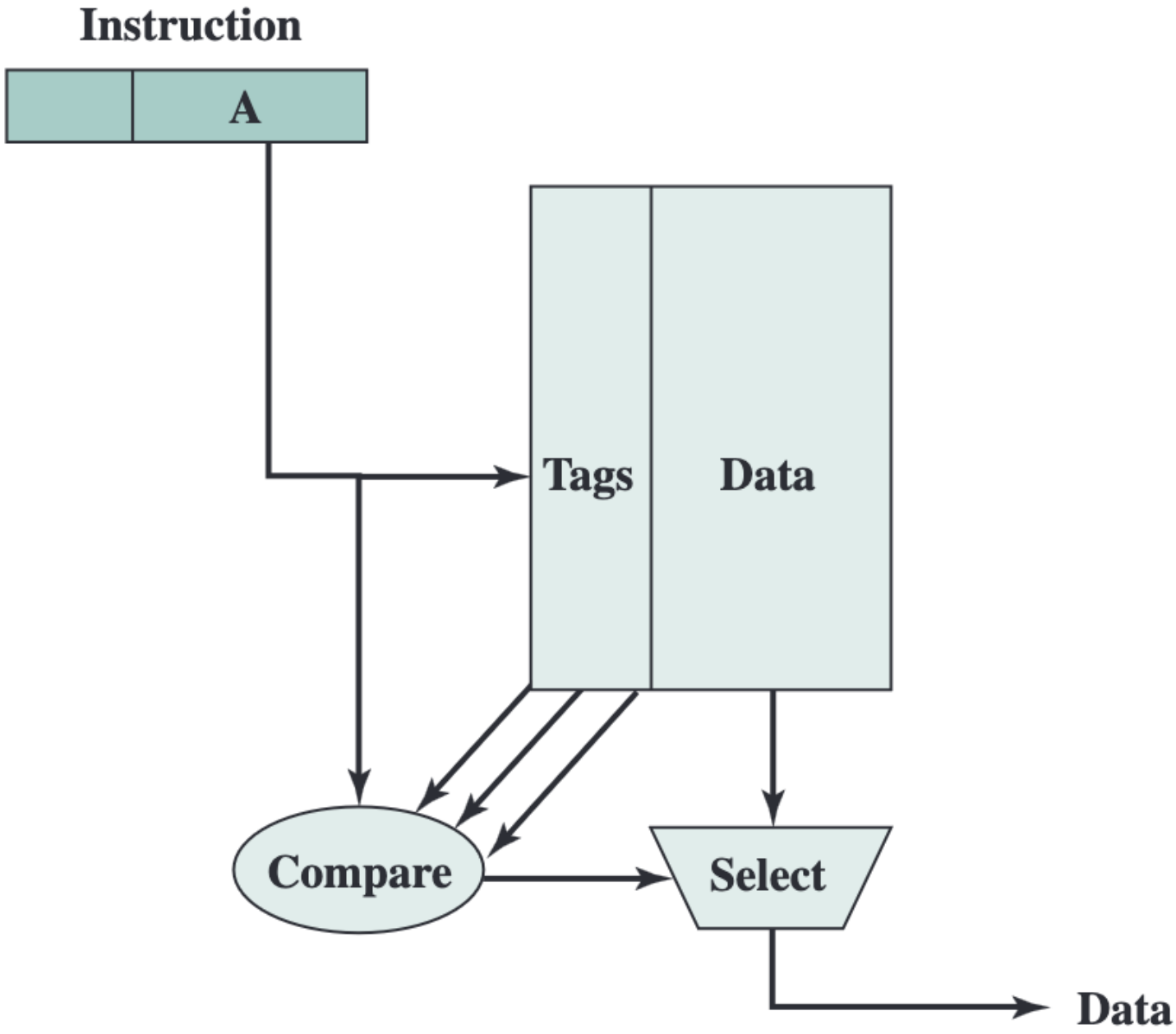
Registers vs. Cache

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
Multiple operands addressed and accessed in one cycle	One operand addressed and accessed per cycle

Referencing a Scalar - Window Based Register File



Referencing a Scalar - Cache



Registers vs. Cache

It should be clear that even if the cache is as fast as the register file, the access time will be considerably longer[1]

- Not obvious
- But register files have simpler and therefore faster addressing
- When L1 (and possibly L2) cache are on-board, cache memory access is almost as fast as register access

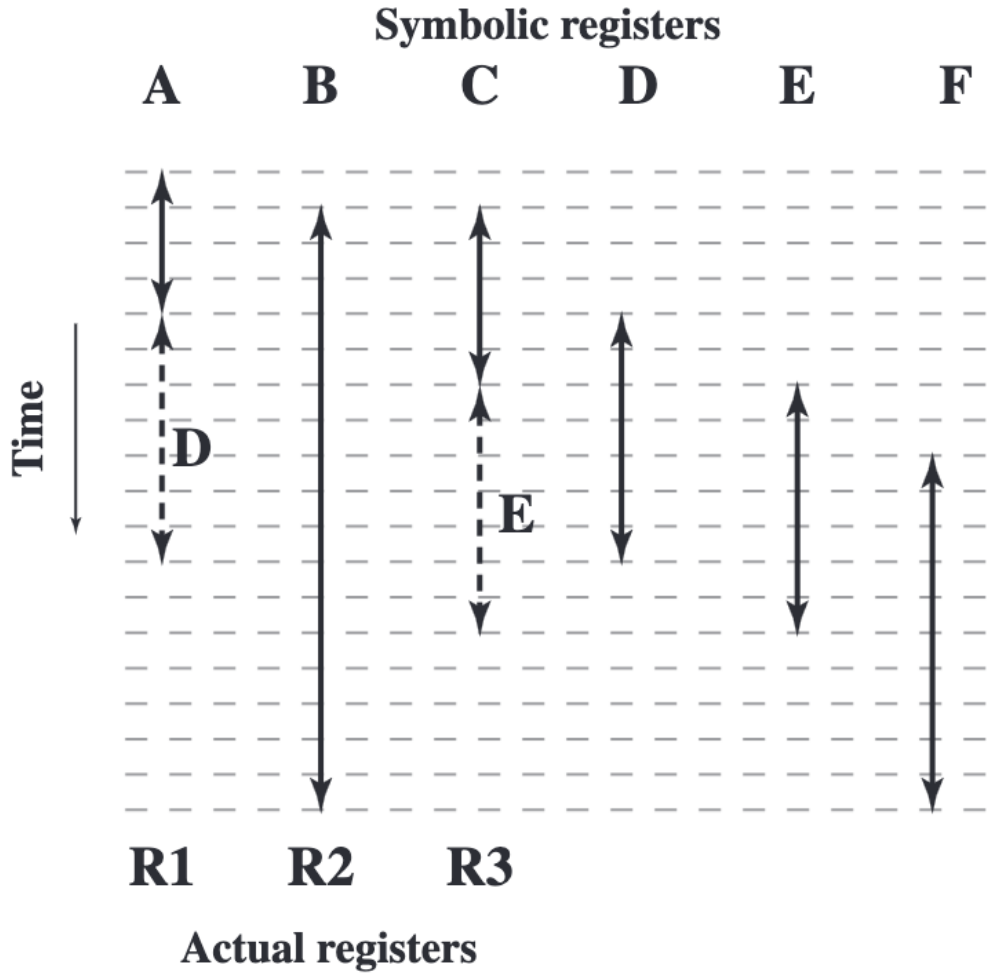
[1] Phrasing a bit weird here

Compiler Based Register Optimization

- Goal of optimizing compiler is to maximize register usage and minimize memory accesses
- Approach:
  1. Assign a symbolic or virtual register to each candidate variable
  2. Map (unlimited) symbolic registers to real registers
  3. Symbolic registers with usage that does not overlap in time can share real registers
  4. If you run out of real registers some variables use memory
  5. One commonly used algorithm is the graph coloring algorithm

Register Interference

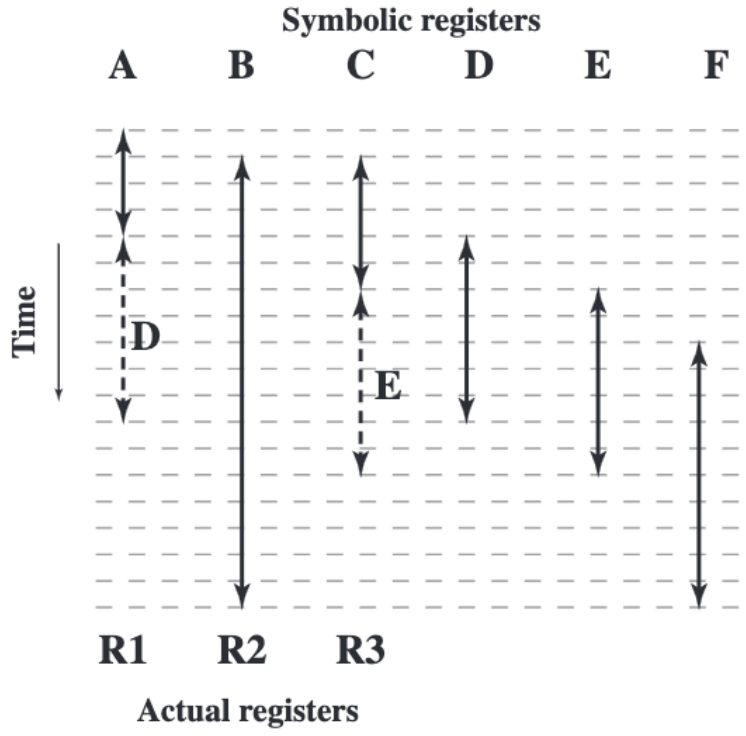
- We have six variables (symbolic registers) but only three actual registers available
- Analyze variable references over time to build a register interference graph

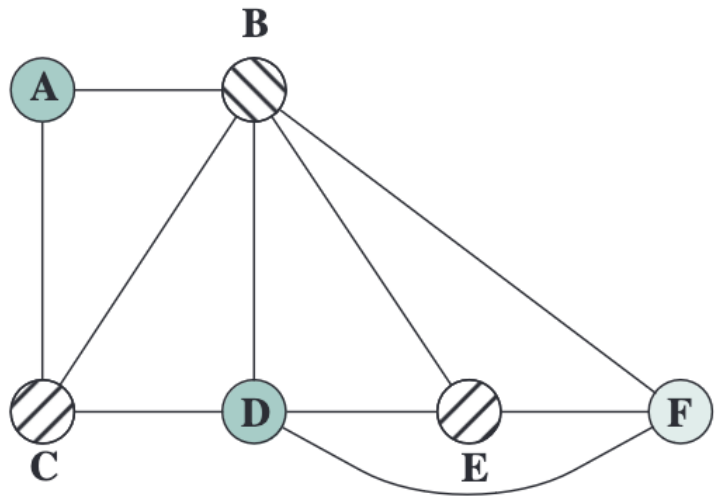


Graph Coloring

- 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 live in the same program fragment are joined by an edge
- Color the graph with n colors, where n is the number of real registers be colored are allocated in memory

Graph Coloring Approach





Why CISC

- Compiler simplification
  - But complex machine instructions are harder to exploit well
  - Machine code optimization is more difficult
- Smaller programs?
  - Program takes up less memory
  - May not occupy less bits, just look shorter in symbolic form
- Faster programs?
  - CISC machines need a more complex control unit and/or larger microprogram control store so simple instructions may take longer to execute

RISC Characteristics

1. One instruction per machine cycle
   - Simple hardwired instructions need little or no microcode
2. Register to register operations
   - Reduces variations in instruction set
   - Memory access Load and Store only
3. Few, simple addressing modes
   - Complex addressing modes can by synthesized in software from simpler ones
4. Few, simple instruction formats
   - Fixed length instruction format
   - Aligned on word boundaries
   - Fixed field locations especially the opcode
   - Simpler decode circuitry

RISC vs CISC

1. Not clear cut distinction
   - Many studies fail to distinguish the effect of a large register file from the effect of RISC instruction set
2. Many designs borrow from both philosophies
   - E.g. PowerPC and Pentium
   - RISC and CISC appear to be converging

Detailed RISC Characteristics

1. A single instruction size, typically 4 bytes
2. Small number of addressing modes
3. No memory-indirect addressing
4. No operations combine load/store with arithmetic
5. No more than one memory addressed operand per instruction
6. Does not support arbitrary (byte) alignment of data for load/store
7. Max number of MMU uses for a data address is 1
8. At least 5 bits for integer register specifier (32 registers)
9. At least 4 bits for FP register specifier (16 registers)

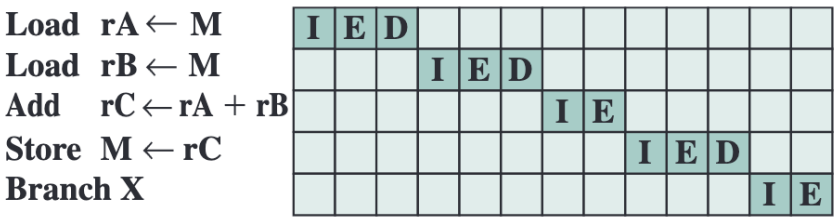
RISC Pipelining

- 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)
- Execute can be further subdivided
  - E1: Register file read
  - E2: ALU operation and register write

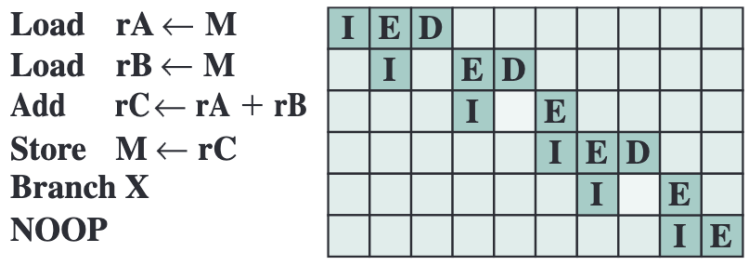
Sequential Execution 2-Stage Pipeline

In the 2 stage pipeline I and E (Fetch and Execute) can be performed in parallel but not D (reg/mem operation)

- Single port memory allows only one access per stage
- Insert a WAIT stage where D operations occur
- Use a No-op (NOOP or NOP) to keep the pipeline full when branch executes (minimizes circuitry needed to handle pipeline stall)



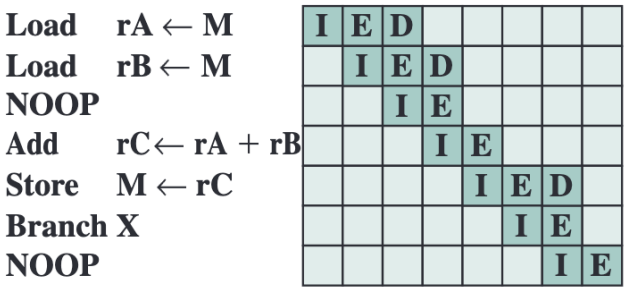
(a) Sequential execution



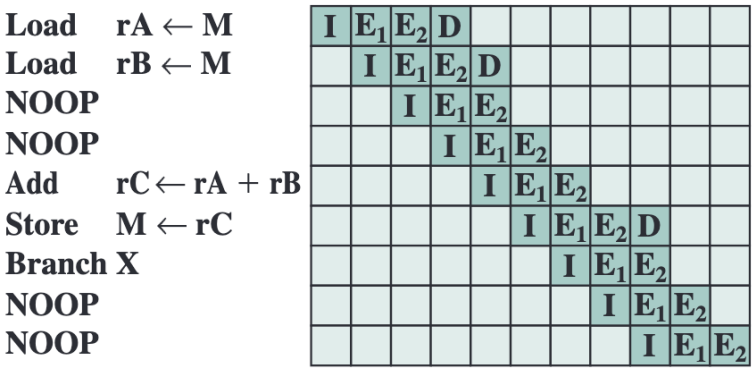
(b) Two-stage pipelined timing

3-Stage and 4-Stage Pipeline

- Permitting 2 memory accesses per stage allows 3-stage pipeline with almost 3x speedup
- Divide E phase into two smaller phases for more even timing in 4 stage pipeline
- Use NOPs for pipeline delays (e.g., data dependencies)



(c) Three-stage pipelined timing



(d) Four-stage pipelined timing

Pipelining Optimizations

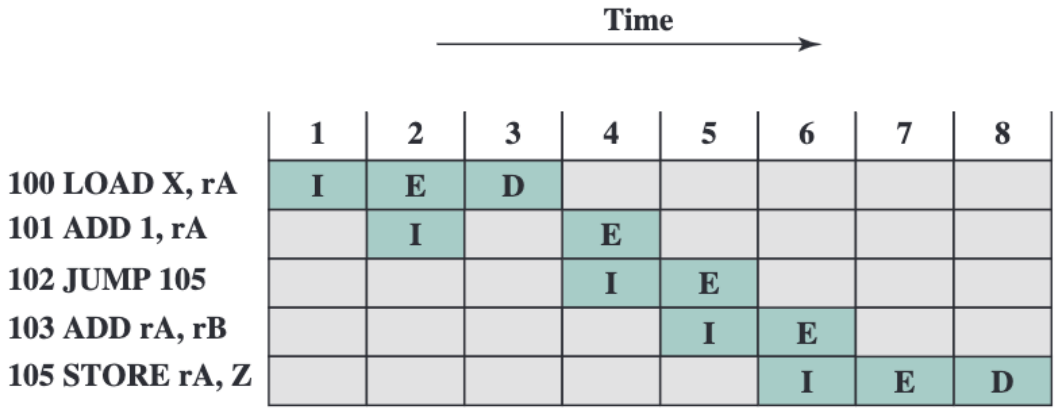
1. Delayed branch
2. Delayed Load
3. Loop Unrolling

Delayed Branch

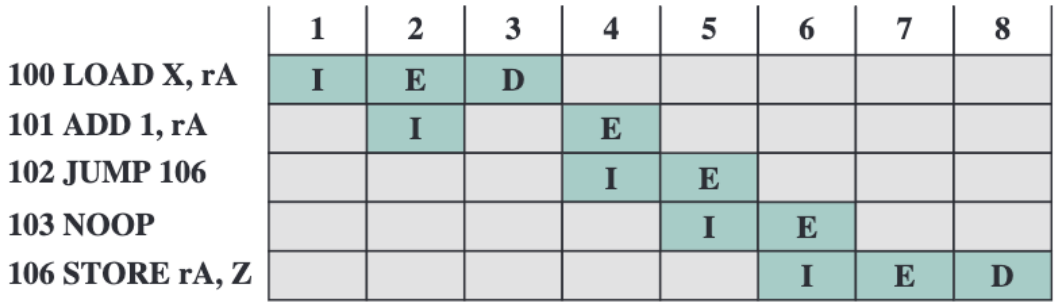
- Does not take effect until after execution of following instruction
- This following instruction is the delay slot

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	ADD	1, rA
103	ADD	rA, rB	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		

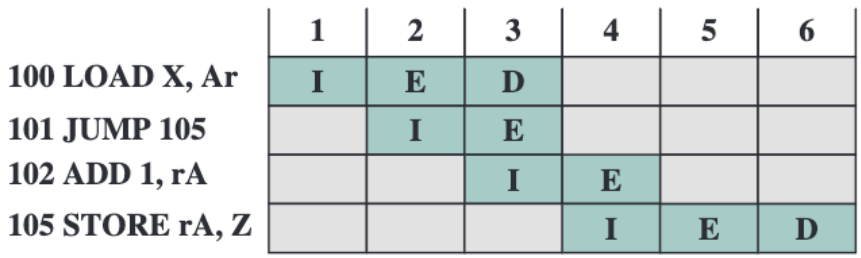
Use of Delayed Branch



(a) Traditional pipeline



(b) RISC pipeline with inserted NOOP



(c) Reversed instructions

Delayed Load

- Register to be target is locked by processor
- Continue execution of instruction stream until register required
- Idle until load is complete
- Re-arranging instructions can allow useful work while loading

Loop Unrolling

- Replicate body of loop a number of times
- Iterate loop fewer times
- Reduces loop overhead
- Increases instruction parallelism
- Improved register, data cache, or TLB locality

Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

After loop unrolling

```
int x;
for (x = 0; x < 100; x += 5 )
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

Loop Unrolling

- Second assignment performed while first is being stored and loop variable updated
- If array elements are in registers then locality of reference will improve because a[I] and a[I+1] are used twice, reducing loads per iteration from 3 to 2

```
do i=2, n-1
        a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) Original loop

```
do i=2, n-2, 2
        a[i] = a[i] + a[i-1] * a[i+1]
        a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = i) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) Loop unrolled twice

Sources Acknowledgement

- http://aturing.umcs.maine.edu/~meadow/courses/cos335/COA13.pdf
- https://en.wikipedia.org/wiki/Loop_unrolling