

5. Program design and analysis



⌘ Software components.

☑ State machine, circular buffer, queue

⌘ Representations of programs.

☑ Data flow graph

☑ Control/data flow graph

⌘ Assembly, linking, and loading.

⌘ Basic compilation techniques

5. Program design and analysis



- ⌘ Program optimization
- ⌘ Program-level performance analysis
- ⌘ Analysis of program size
- ⌘ Program validation and testing
- ⌘ Software modem

Interactive system



- ⌘ Permanently communicate with its environment
- ⌘ At their own speed, making it wait
- ⌘ Concurrent processes in OS or DB management

Reactive system



⌘ React to the environment that cannot wait

⌘ Features

- ☑ Intended to be deterministic

- ☑ Involve concurrency

 - ☑ Run in parallel with its environment

 - ☑ Distributed architecture: physical concurrency

 - ☑ A set of concurrent processes: logical concurrency

⌘ Most critical systems are reactive

Real-time system



⌘ Real-time: receive interrupt or read sensors, then issue commands to it

- ☑ Timing constraints

- ☑ Safety

- ☑ Logical correctness

- ☑ Temporal correctness

Components for embedded programs



- ⌘ Three structural components

- ⌘ Reactive systems: user interfaces (?)

 - ☑ State machine

- ⌘ Digital signal processing

 - ☑ Circular buffer

 - ☑ Queue

Software state machine



⌘ State machine keeps internal state as a variable, changes state based on inputs.

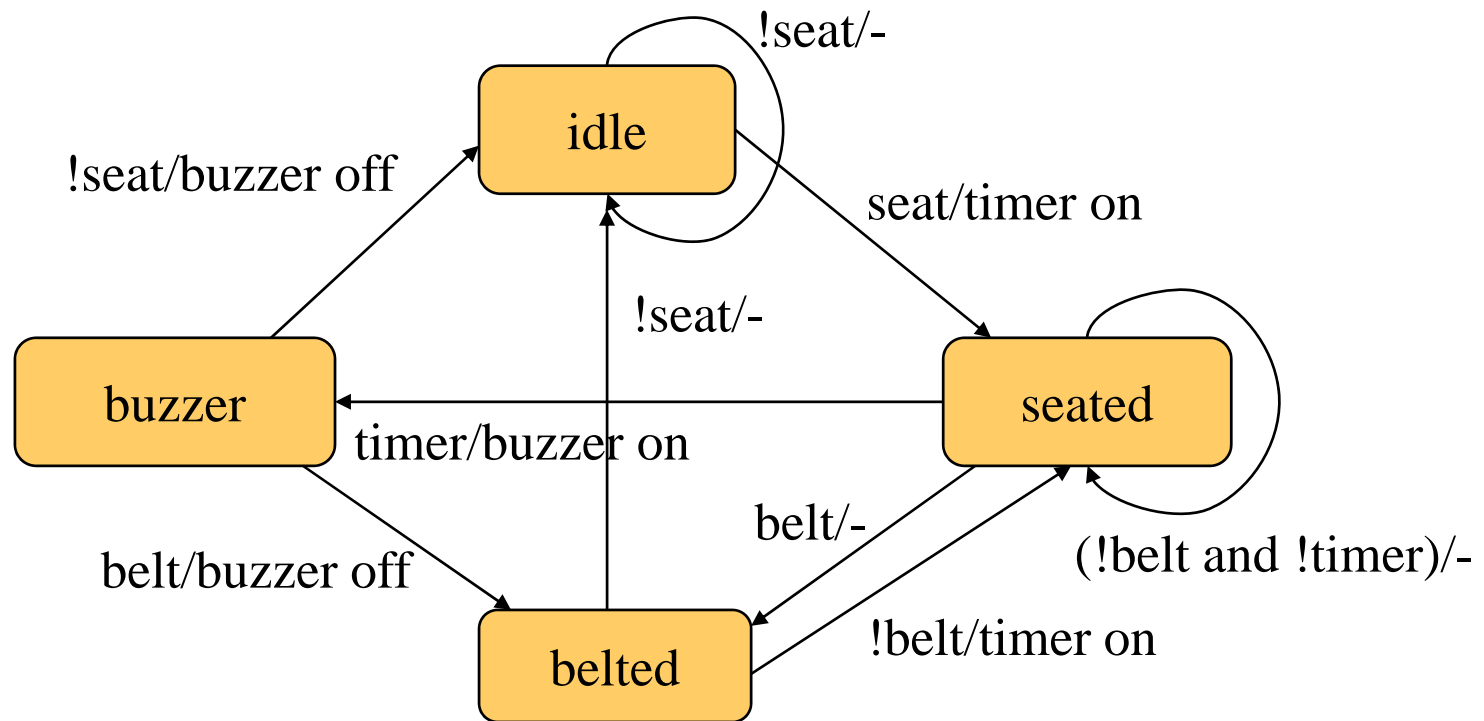
⌘ Uses:

- ☑ control-dominated code;

- ☑ reactive systems.

State machine example

A simple seat belt controller



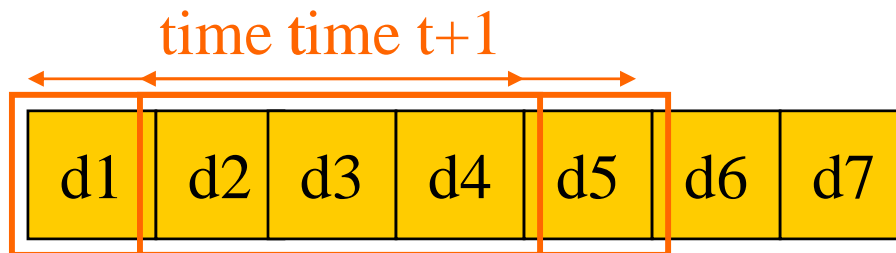
C implementation

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
    case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
               /* default is self-loop */
               break;
    case SEATED: if (belt) state = BELTED;
                 else if (timer) state = BUZZER;
                 /* default is self-loop */
                 break;
    case BELTED: if (!seat) state = IDLE;
                 break;
    case BUZZER: if (belt) state = BELTED;
                 else if (!seat) state = IDLE;
                 break;
}
```

Circular buffer

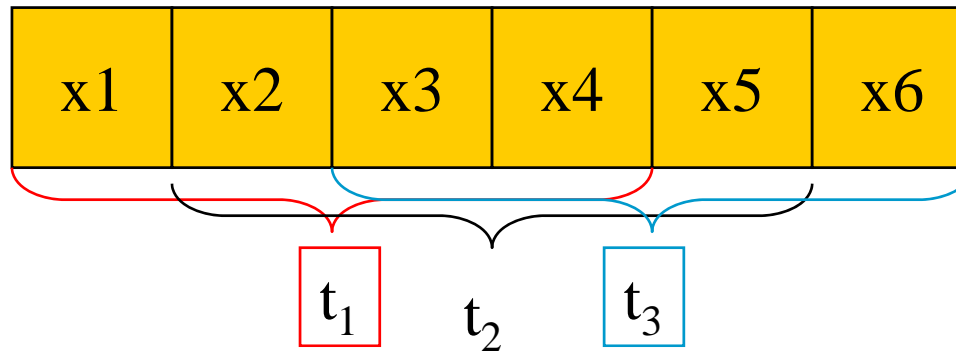
⌘ Commonly used in signal processing:

- ⊞ new data **regularly** arrives;
- ⊞ each datum has a limited lifetime.

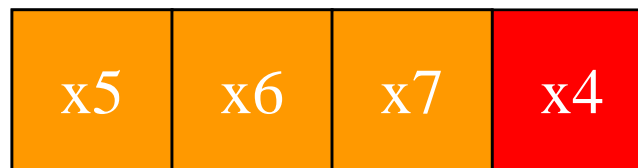


⌘ Use a circular buffer to hold the **data stream**.

Circular buffer



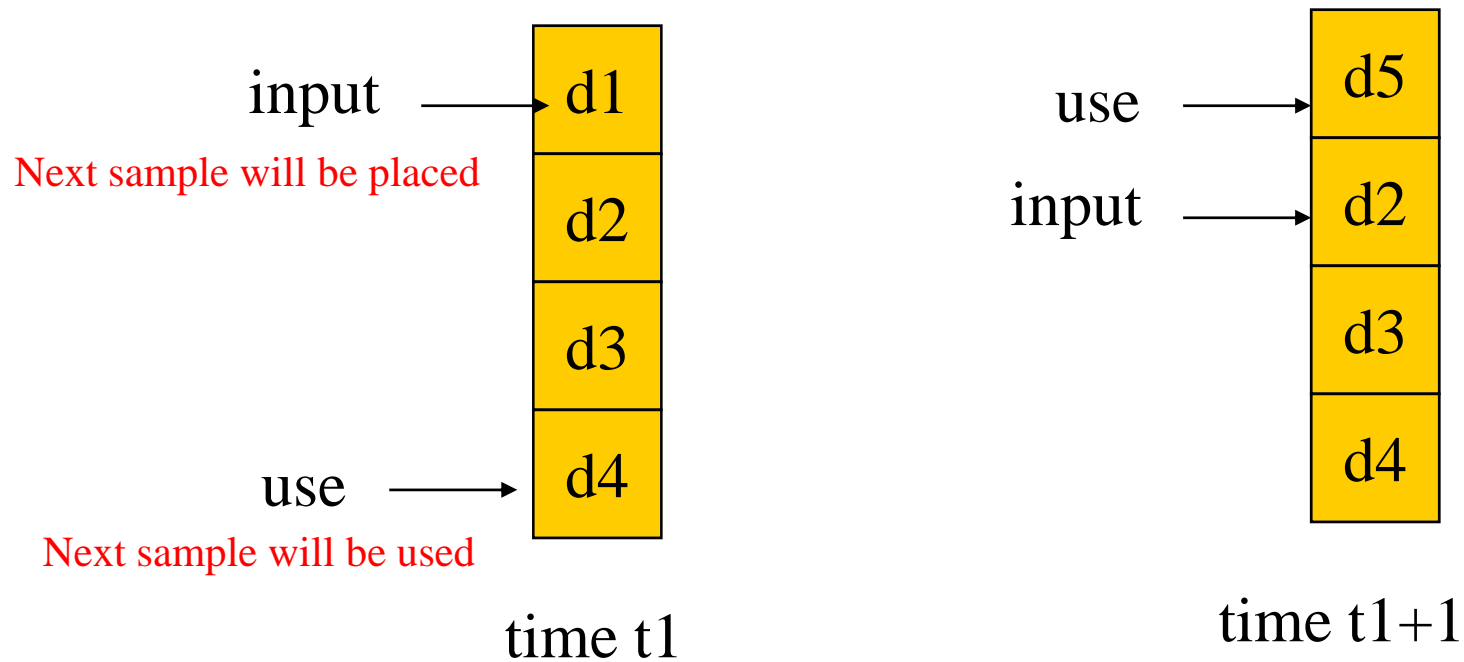
Data stream



Circular buffer

Circular buffers

⌘ Indexes locate currently used data, current input data:



Circular buffer for FIR filter

```
int circ_buffer[M];      /* circular buffer for data */
int circ_buffer_head = 0;
int c[N];                /* coefficients */
...
int f;                   /* loop counter */
int ibuf;                /* loop index for buffer buffer */
int ic;                  /* loop index for the coeff array */
for (f=0, ibuff=circ_buff_head, ic=0;    ic<N;
      ibuff=(ibuff==(M-1)?0:ibuff++), ic++)
    f = f + c[ic]*circ_buffer[ibuff];
```

Queues



- ⌘ Elastic buffer: holds data that arrives irregularly.
- ⌘ Can be implemented with a linked list
- ⌘ Allow it to grow to an arbitrary size
- ⌘ In many application we are unwilling to pay the price of dynamically allocating memory.
 - ☑ Use an array

Example



- ⌘ Example 3.5 (p. 99) :A circular buffer to manage interrupt-driven data
- ⌘ Example 5.3: an array for non-interrupt version
- ⌘ Errors
 - ⊞ initialize_queue
 - ⊞ enqueue

Buffer-based queues

```
#define Q_SIZE 32
#define Q_MAX (Q_SIZE-1)
int q[Q_MAX], head, tail;
void initialize_queue() { head =
    tail = 0; }
void enqueue(int val) {
    if (((tail+1)%Q_SIZE) ==
        head) error(EnQ_error);
    /* add to the full Q */
    q[tail]=val;
    if (tail == Q_MAX) tail = 0;
    else tail++;
}
```

```
int dequeue() {
    int returnval;
    if (head == tail) error(DeQ_error);
    /* remove from empty Q */
    returnval = q[head];
    if (head == Q_MAX) head = 0;
        else head++;
    return returnval;
}
```


Models of programs



⌘ Source code is not a good representation for programs:

- ☑ clumsy;

- ☑ leaves much information implicit.

⌘ Compilers derive intermediate representations (IR) to manipulate and optimize the program.

Data flow graph (DFG)



- ⌘ A model for a code segment with no conditionals
- ⌘ Basic block: One entry and one exit
- ⌘ Describes the minimal ordering requirements on operations.

Single assignment form



$x = a + b;$

$y = c - d;$

$z = x * y;$

$y = b + d;$

$x = a + b;$

$y = c - d;$

$z = x * y;$

$y1 = b + d;$

original basic block \longrightarrow single assignment form

Data flow graph

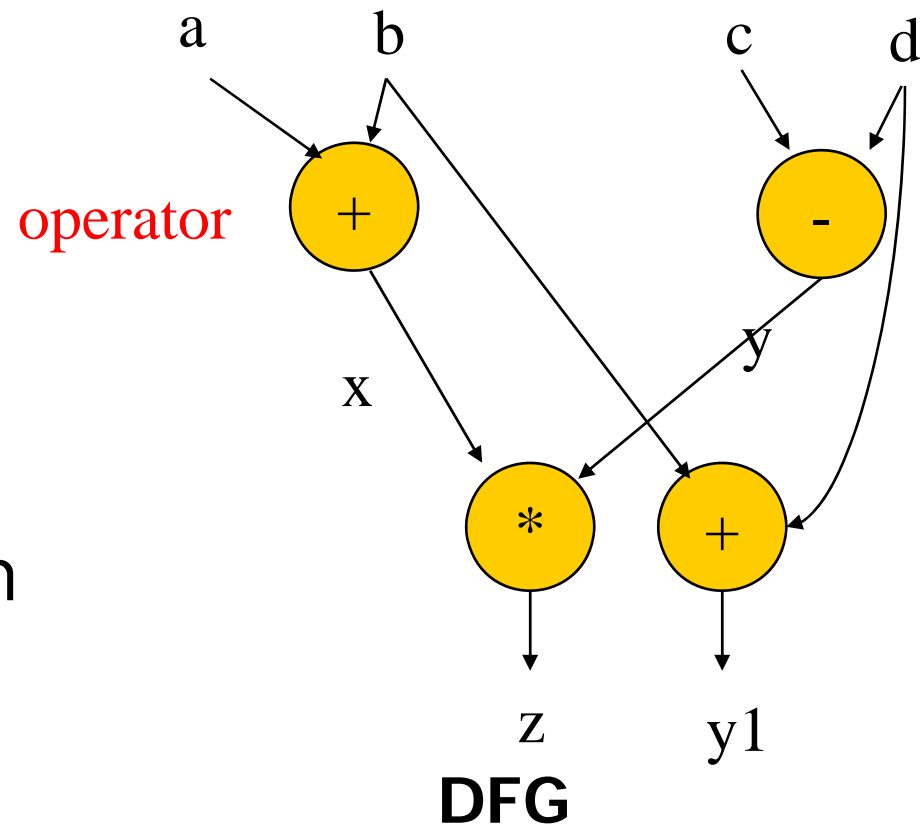
$x = a + b;$

$y = c - d;$

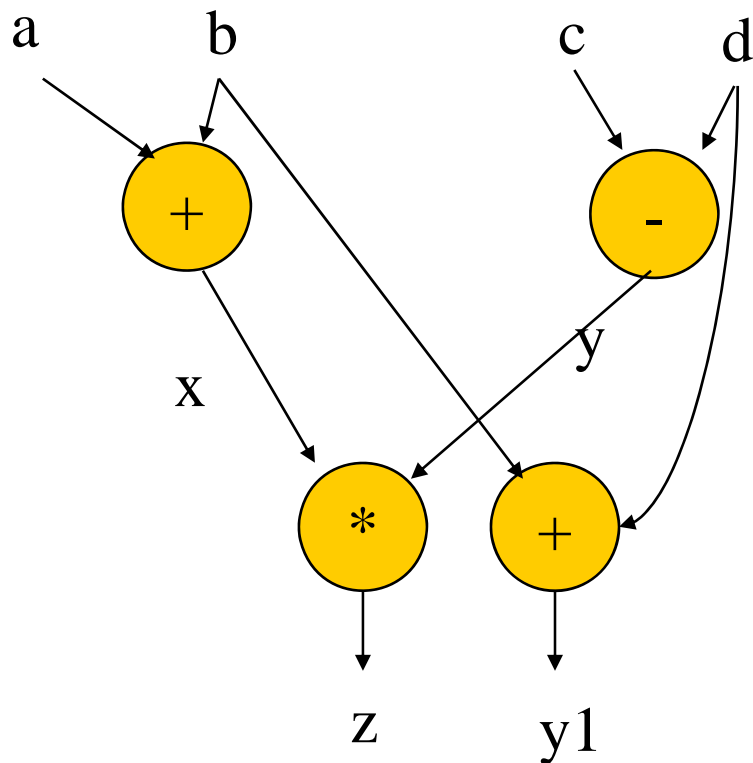
$z = x * y;$

$y1 = b + d;$

single assignment form



DFGs and partial orders



Partial order:

⌘ $a+b, c-d; b+d, x*y$

Can do pairs of operations
in any order.

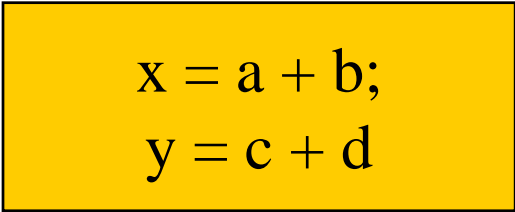
Control-data flow graph (CDFG)

- ⌘ represents both control and data.
- ⌘ Uses data flow graphs as components.
- ⌘ Two types of nodes:
 - ☒ decision;
 - ☒ data flow.

Data flow node



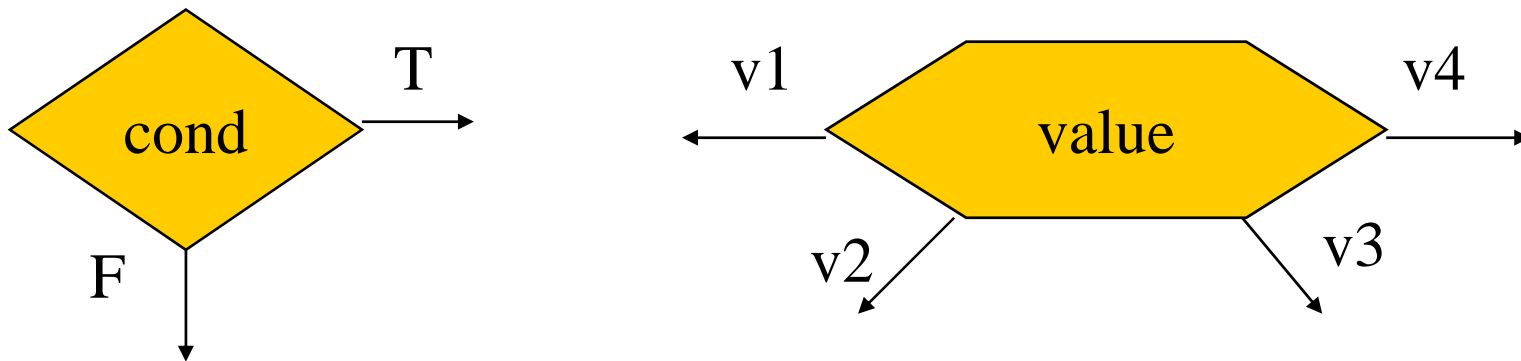
Encapsulates a data flow graph:



```
x = a + b;  
y = c + d
```

Write operations in basic block form for simplicity.

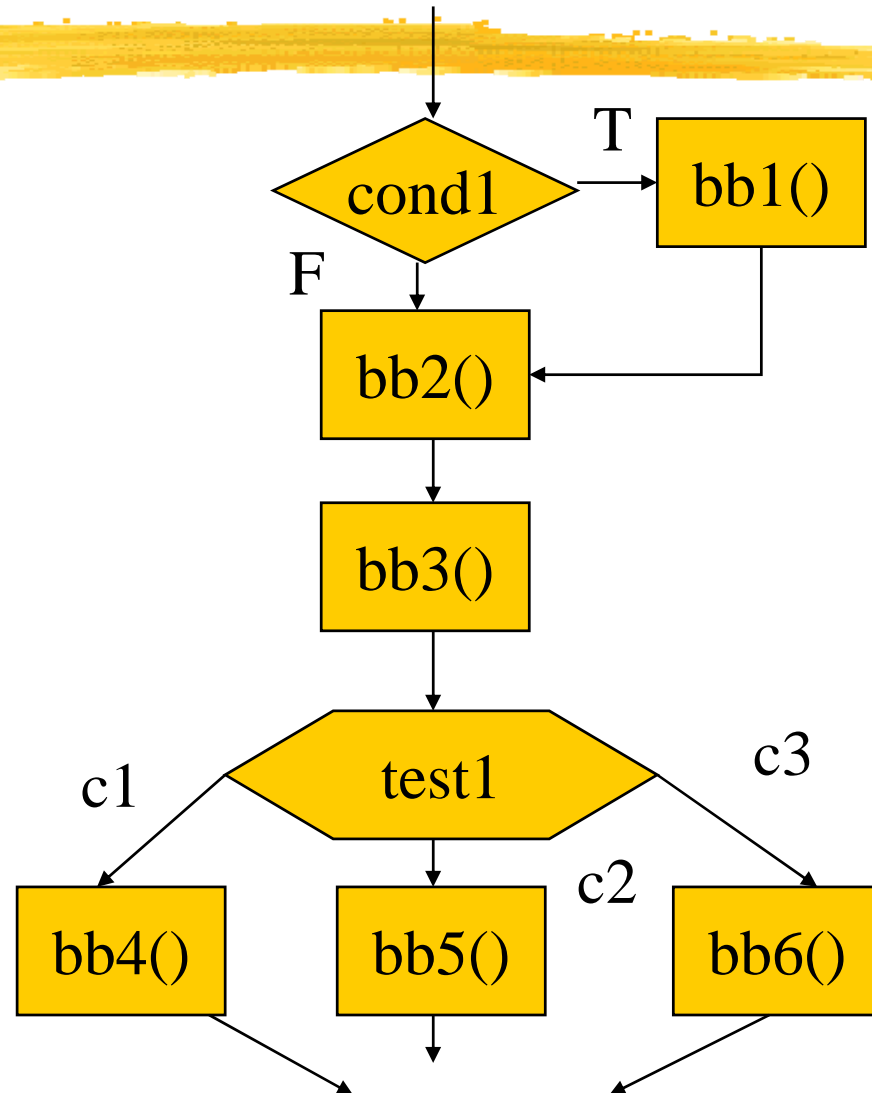
Control



Equivalent forms

CDFG example

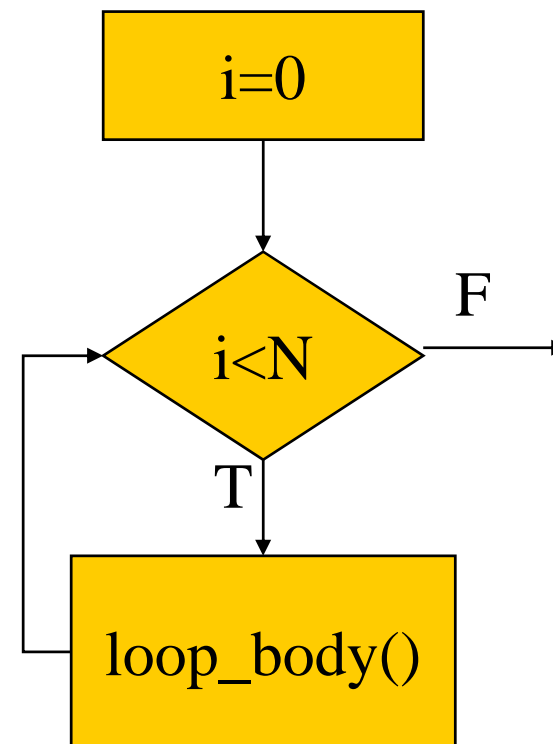
```
if (cond1) bb1();  
else bb2();  
bb3();  
switch (test1) {  
  case c1: bb4(); break;  
  case c2: bb5(); break;  
  case c3: bb6(); break;  
}
```



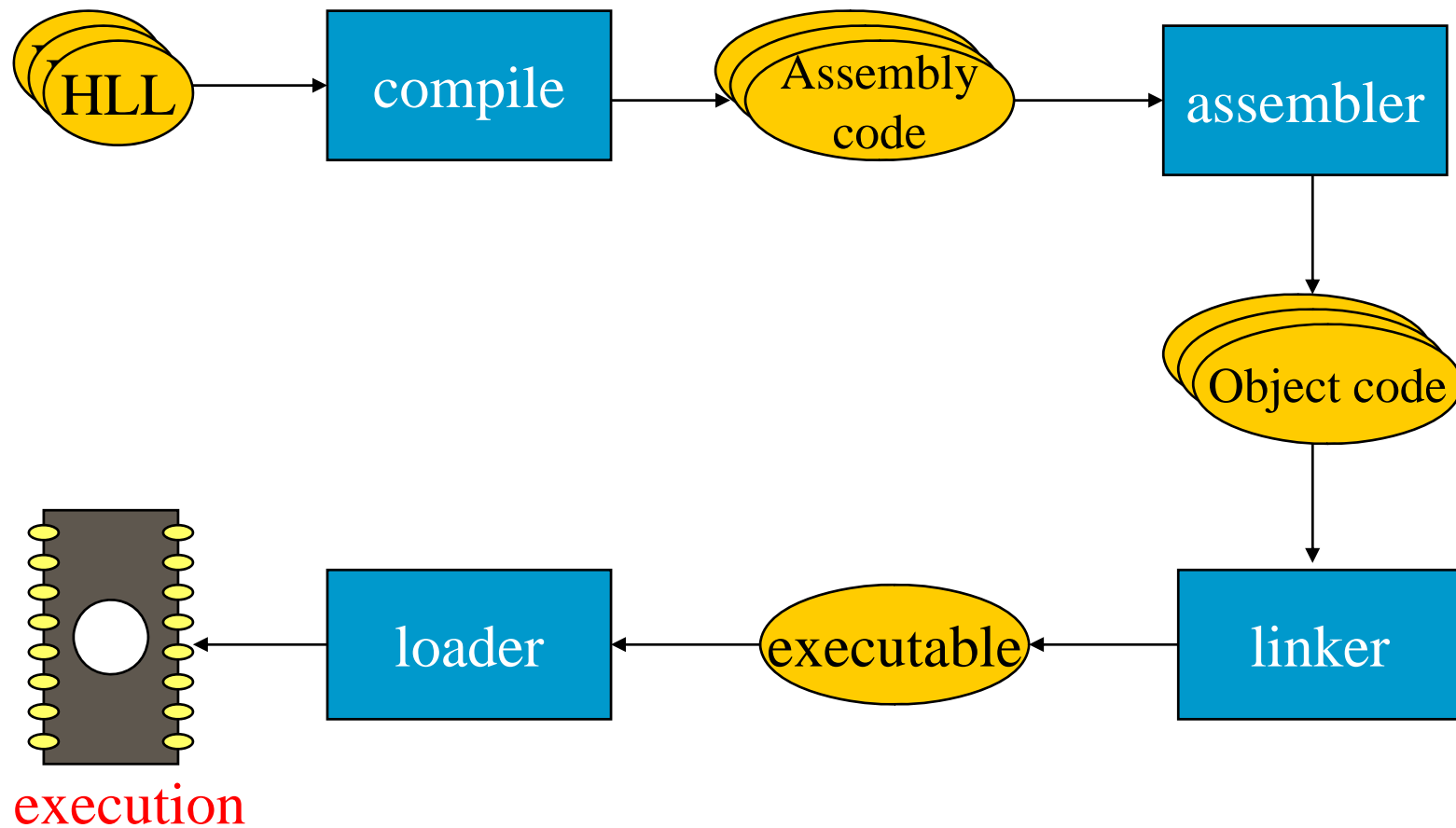
for loop

```
for (i=0; i<N; i++)  
    loop_body();  
for loop
```

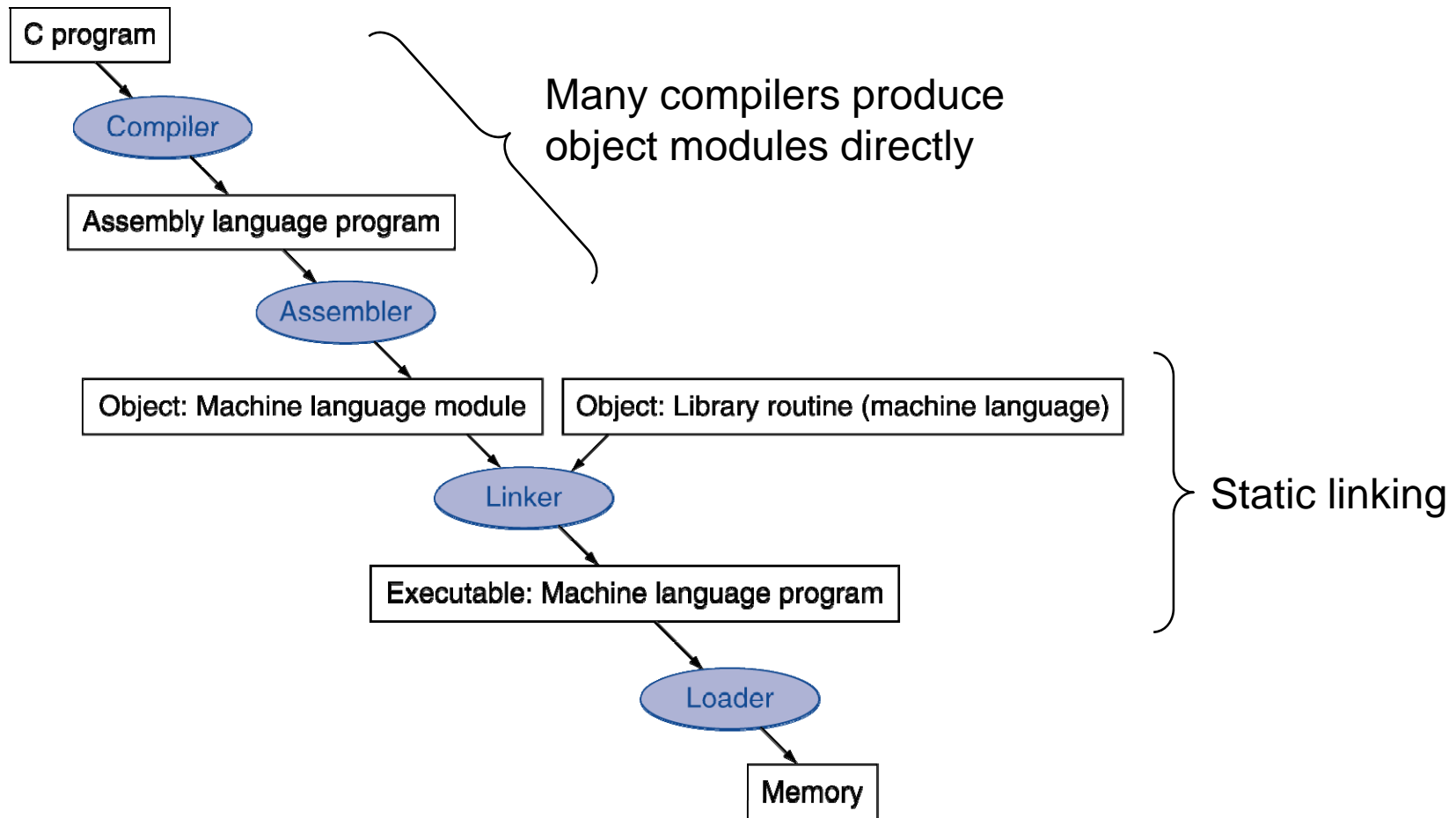
```
i=0;  
while (i<N) {  
    loop_body(); i++; }  
equivalent
```



Assembly, linking, and loading



Translation



Multiple-module programs



- ⌘ Programs may be composed from several files.
- ⌘ Addresses become more specific during processing:
 - ⊠ **relative addresses** are measured relative to the start of a module;
 - ⊠ **absolute addresses** are measured relative to the start of the CPU address space.

Assemblers



⌘ Major tasks:

- ☑ generate binary for symbolic instructions;
- ☑ translate labels into addresses;
- ☑ handle pseudo-ops (data, etc.).

⌘ Generally one-to-one translation.

⌘ Assembly labels:

```
ORG 100  
label1  ADR r4,c
```

Symbol table



	ADD r0,r1,r2	xx	0x8
xx	ADD r3,r4,r5	yy	0x10
	CMP r0,r3		
yy	SUB r5,r6,r7		

assembly code

symbol table

Symbol table generation



- ⌘ Use program location counter (**PLC**) to determine address of each location.
- ⌘ Scan program, keeping count of PLC.
- ⌘ Addresses are generated at assembly time, not execution time.

Symbol table example

PLC=0x7			
PLC=0x8	ADD r0,r1,r2	xx	0x8
PLC=0x9	ADD r3,r4,r5	yy	0x10
PLC=0x10	MIP r0,r3		
yy →	SUB r5,r6,r7		

Producing an Object Module



- ⌘ Assembler (or compiler) translates program into machine instructions
- ⌘ Provides information for building a complete program from the pieces
 - ☑ Header: described contents of object module
 - ☑ Text segment: translated instructions
 - ☑ Static data segment: data allocated for the life of the program
 - ☑ Relocation info: for contents that depend on absolute location of loaded program
 - ☑ Symbol table: global definitions and external refs
 - ☑ Debug info: for associating with source code

Two-pass assembly



⌘ Pass 1:

- ☑ generate symbol table

⌘ Pass 2:

- ☑ generate binary instructions

Relative address generation



- ⌘ Some label values may not be known at assembly time.
- ⌘ Labels within the module may be kept in relative form.
- ⌘ Must keep track of external labels---can't generate full binary for instructions that use external labels.

Pseudo-operations



⌘ Pseudo-ops do not generate instructions:

☑ **ORG** sets program location.

☑ **EQU** generates symbol table entry without advancing PLC.

☑ **Data statements** define data blocks.

Linking Object Modules



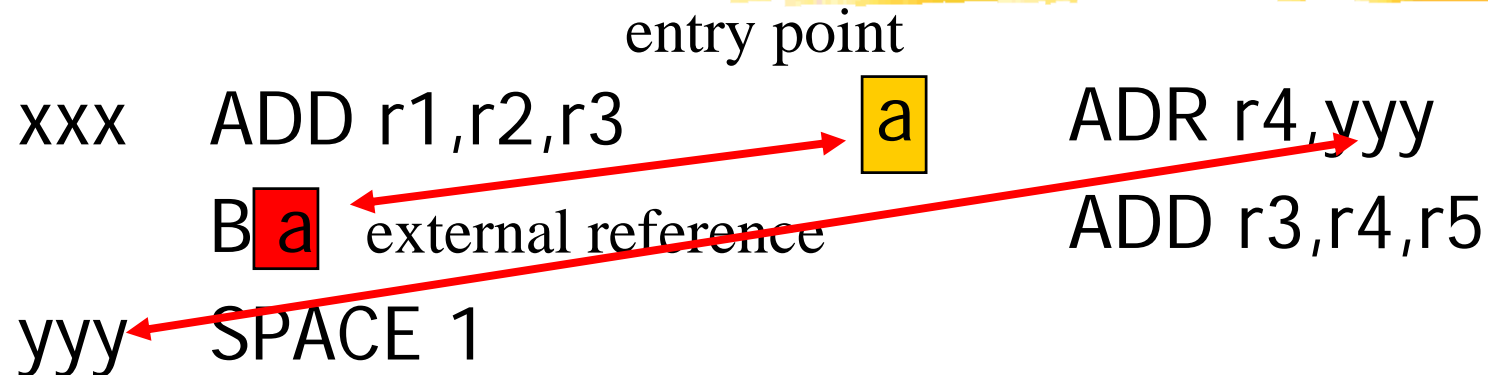
- ⌘ Produces an executable image
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs
- ⌘ Could leave location dependencies for fixing by a relocating loader
 - ☑ But with virtual memory, no need to do this
 - ☑ Program can be loaded into absolute location in virtual memory space

Linking



- ⌘ Combines several object modules into a single executable module.
- ⌘ Jobs: 2 passes
 - ☑ put modules in order; (load map)
 - ☑ resolve labels across modules after merging all symbol tables into a larger one.

Externals and entry points



Entry points: the place in a file where a label is defined

External references: the place in a file where an external label is used.

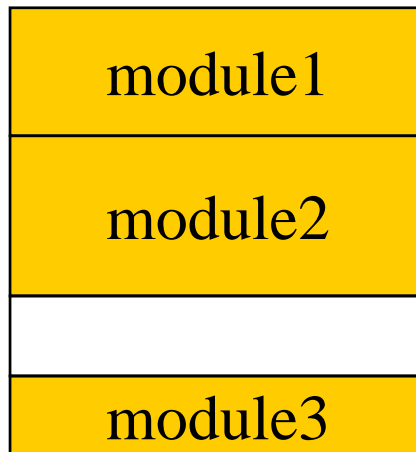
Refer: Figure 5.10 (p. 226)

SPACE : alias %

EQU: alias *

Module ordering

- ⌘ Code modules must be placed in absolute positions in the memory space.
- ⌘ **Load map** or linker flags control the order of modules.



Static shared library and DLL



- When different programs are running on a computer, those different programs usually turn out to share a lot of common code.
 - Nearly every C program uses routines such as fopen, and printf.
 - Programs running under a GUI such as X Windows, or MS Windows all use pieces of the GUI library.
 - Most systems now provide shared libraries for programs to use, so all the programs that use a library can share a single copy of it.
 - **Static shared library**
 - The linker binds program references to library routines to those specific addresses at link time.
 - **Dynamic linked library**
 - Library sections and symbols are not bound to actual addresses until the program that uses the library starts running.

Dynamic Linking



⌘ Only link/load library procedure when it is called

- ☑ Shares one copy of library among all executing programs;
- ☑ Requires procedure code to be relocatable
- ☑ Automatically picks up new library versions

Loading a Program



- ⌘ Load from image file on disk into memory
 1. Read header to determine segment sizes
 - ☒ Validation: permission, memory requirement
 2. Create virtual address space
 3. Copy text and initialized data into memory
 - ☒ Or set page table entries so they can be faulted in
 4. Copy command line arguments on stack
 5. Initialize registers (including \$sp, \$fp)
 6. Jump to startup routine

Program design and analysis



- ⌘ Compilation flow.
- ⌘ Basic statement translation.
- ⌘ Basic optimizations.
- ⌘ Interpreters and just-in-time compilers.

Compilation



⌘ Compilation strategy (Wirth):

- ☑ compilation = translation + optimization

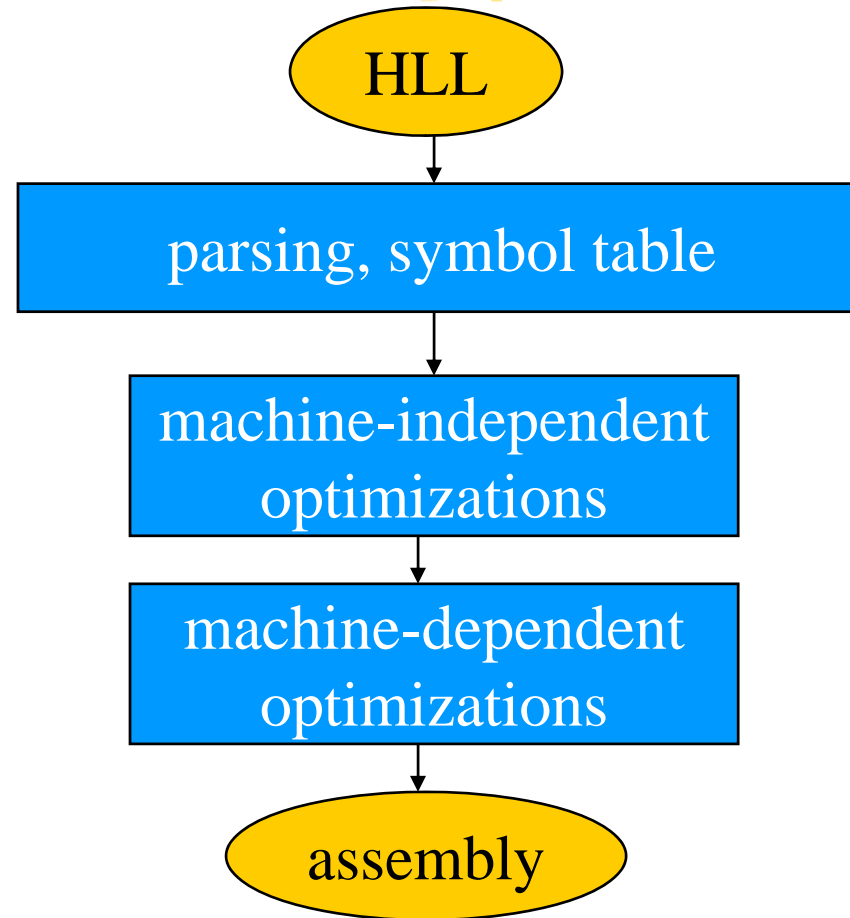
⌘ Compiler determines quality of code:

- ☑ use of CPU resources;

- ☑ memory access scheduling;

- ☑ code size.

Basic compilation phases



Statement translation and optimization

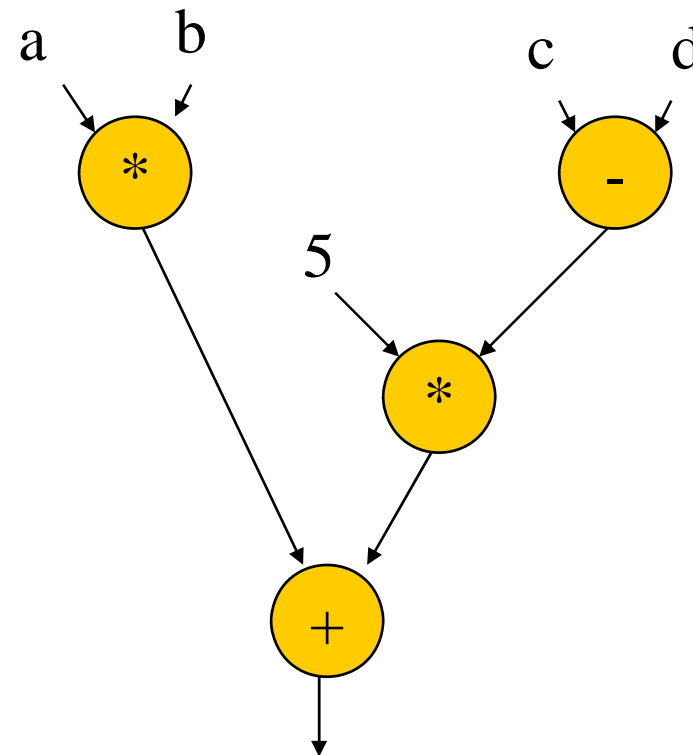


- ⌘ Source code is translated into intermediate form such as CDFG.
- ⌘ CDFG is transformed/optimized.
- ⌘ CDFG is translated into instructions with optimization decisions.
- ⌘ Instructions are further optimized.

Arithmetic expressions

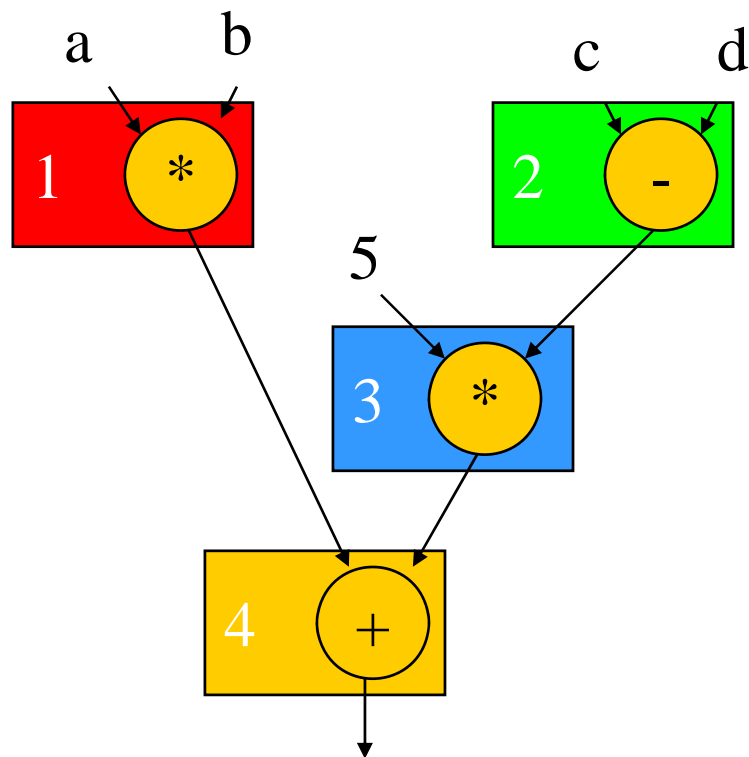
$a * b + 5 * (c - d)$

expression



DFG

Arithmetic expressions



DFG

```
ADR r4,a
MOV r1,[r4]
ADR r4,b
MOV r2,[r4]
ADD ,r1,r2
```

```
ADR r4,c
MOV r1,[r4]
ADR r4,d
MOV r5,[r4]
SUB r6,r4,r5
```

```
MUL r7,r6,#5
```

```
ADD r8,r7,r3
```

code

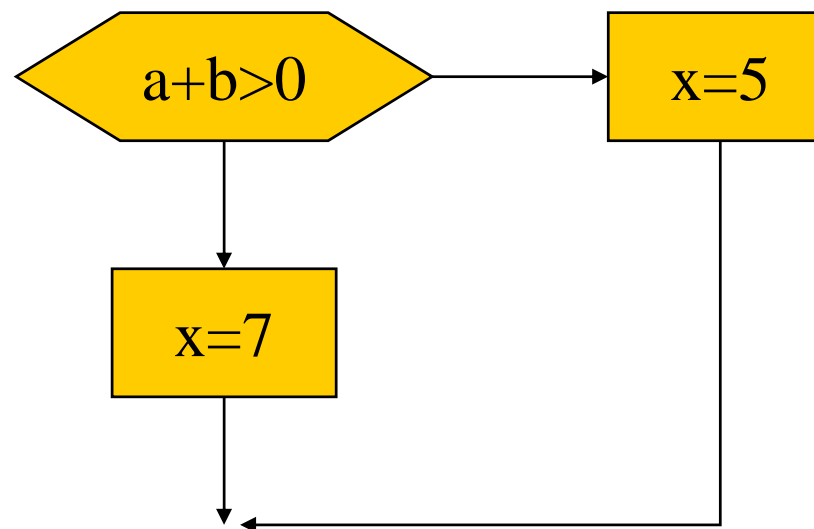
Control code generation

if (a+b > 0)

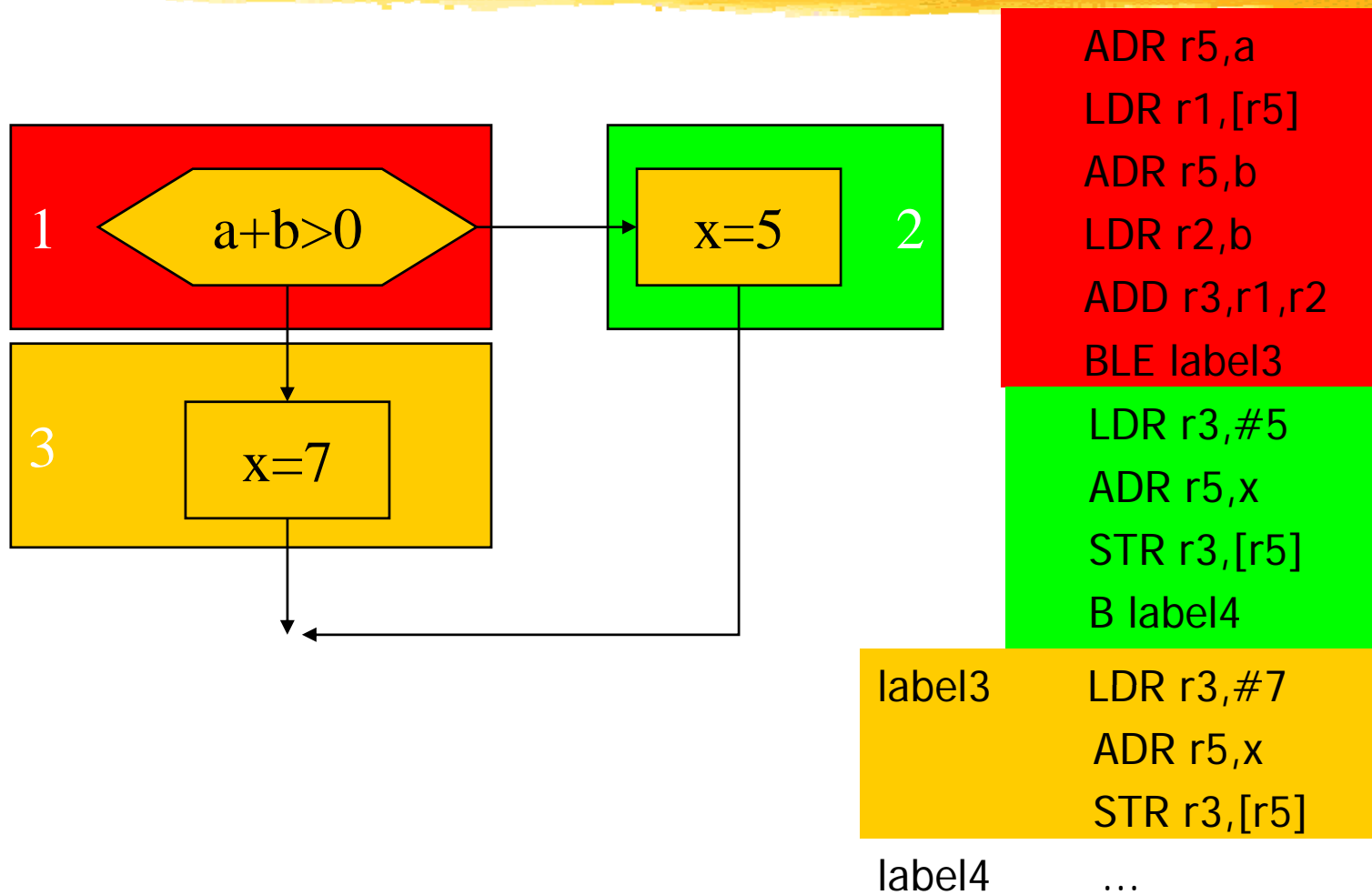
 x = 5;

else

 x = 7;



Control code generation



Procedure linkage



⌘ Need code to:

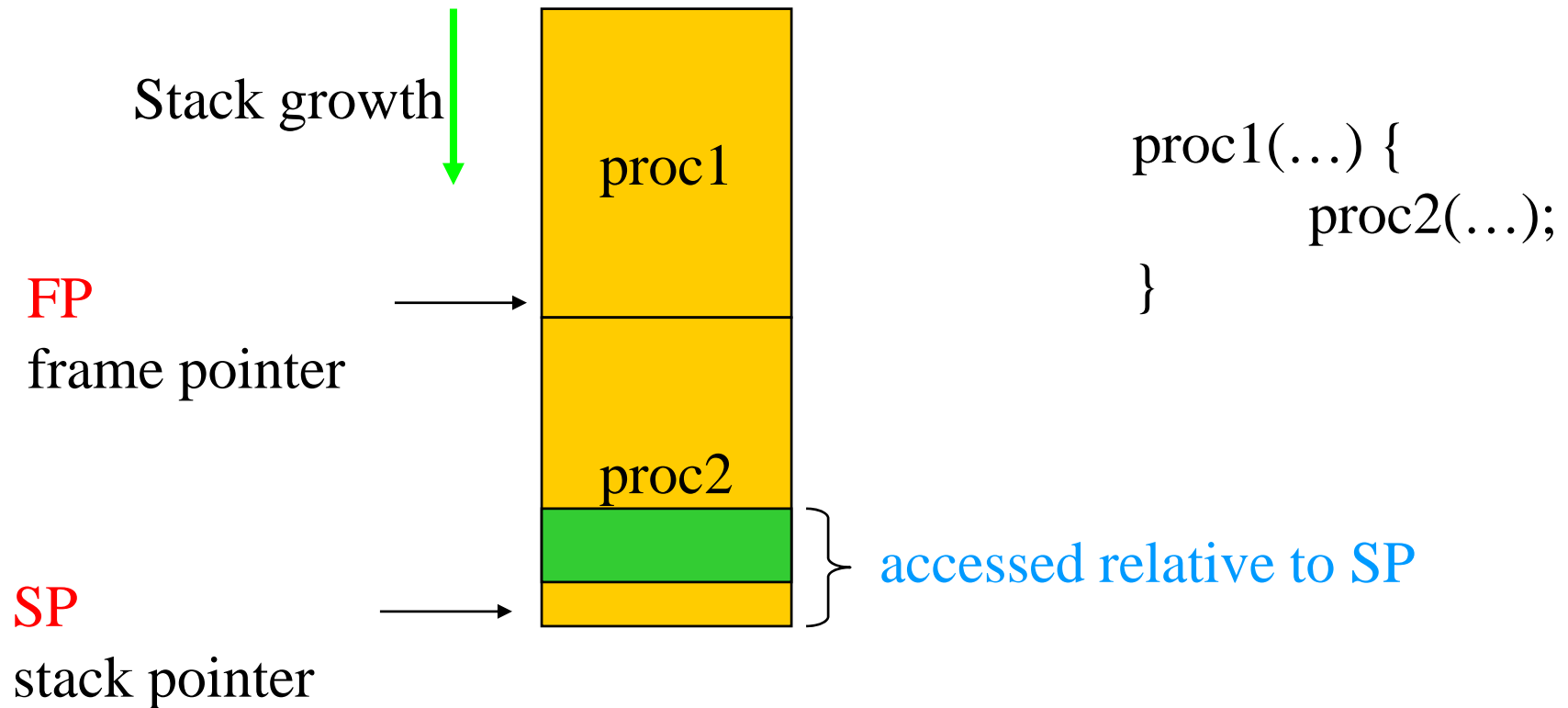
- ☑ call and return;

- ☑ pass parameters and results.

⌘ Parameters and returns are passed on stack.

- ☑ Procedures with few parameters may use registers.

Procedure stacks



ARM procedure linkage



⌘ APCS (ARM Procedure Call Standard):

- ☑ r0-r3 pass parameters into procedure. Extra parameters are put on stack frame.
- ☑ r0 holds return value.
- ☑ r4-r7 hold register values.
- ☑ r11 is frame pointer, r13 is stack pointer.
- ☑ r10 holds limiting address on stack size to check for stack overflows.

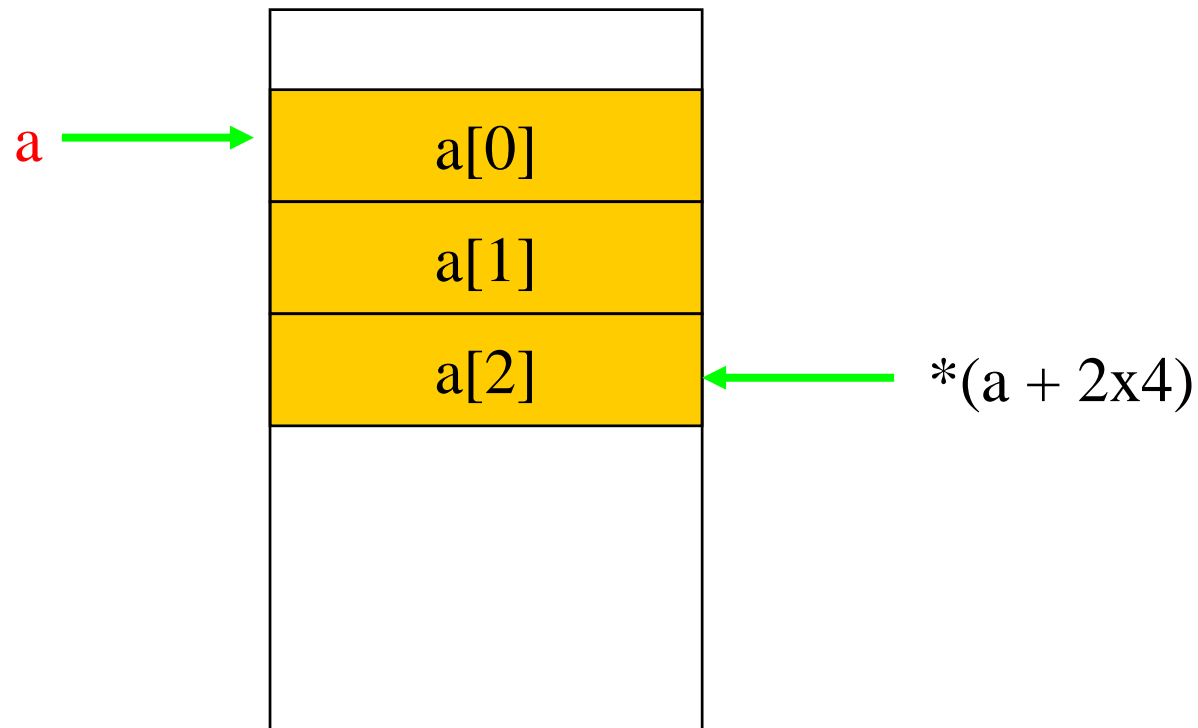
Data structures



- ⌘ Different types of data structures use different data layouts.
- ⌘ Some offsets into data structure can be computed at compile time, others must be computed at run time.

One-dimensional arrays

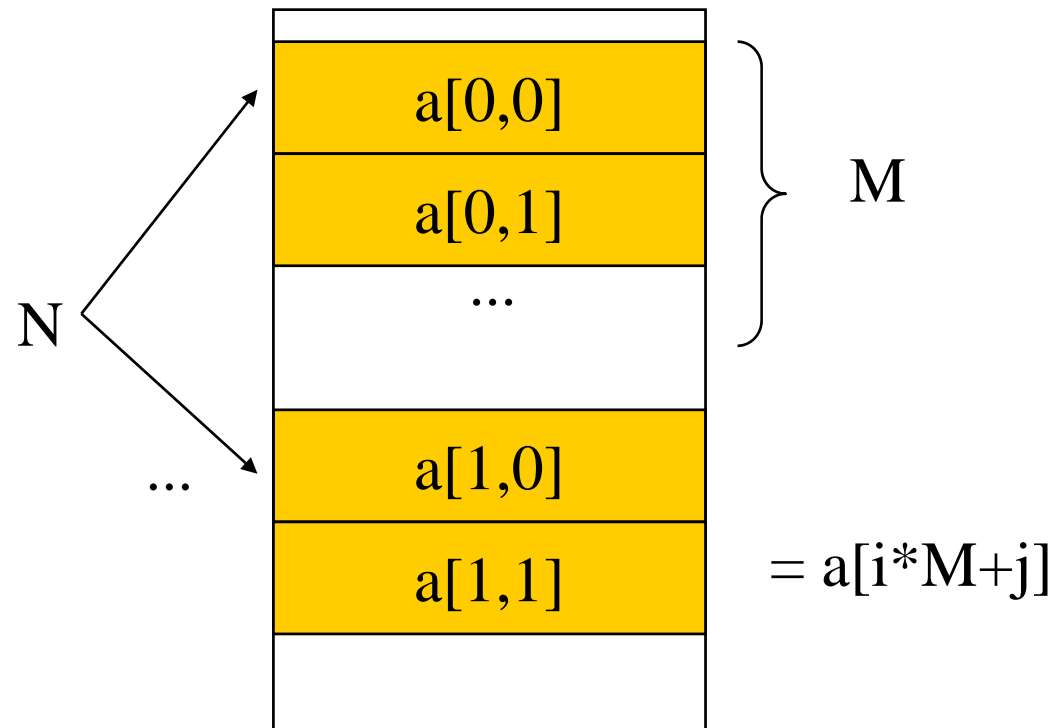
⌘ C array name points to 0th element:



Two-dimensional arrays

⌘ Row-major layout:

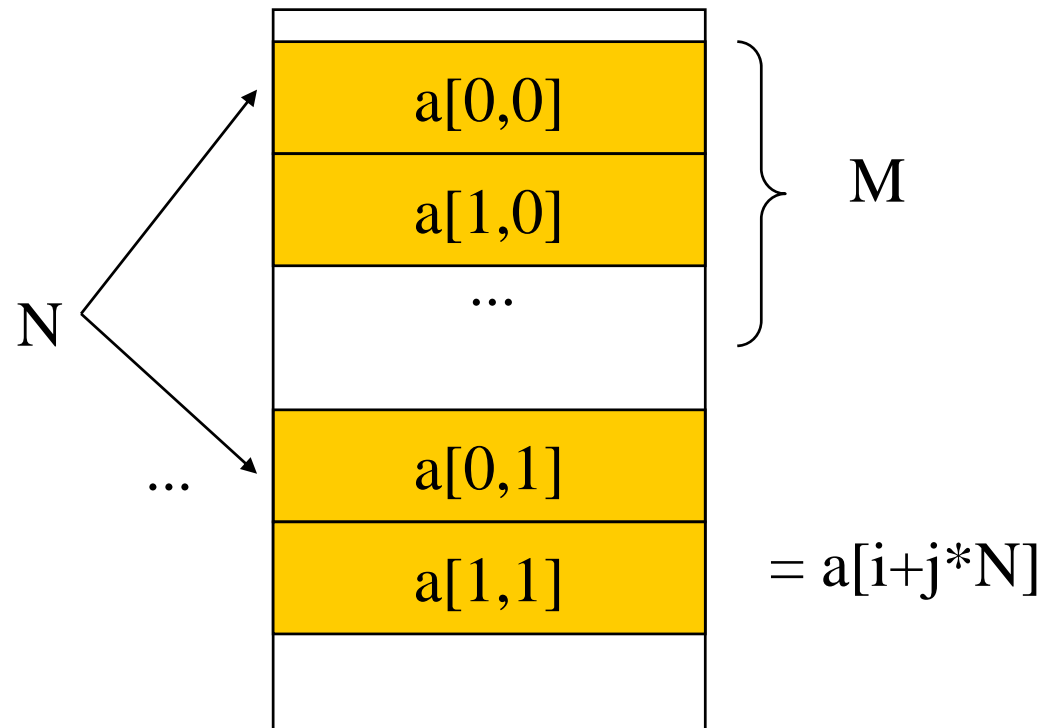
Array size: $a[M,N]$



Two-dimensional arrays

⌘ Column-major layout: FORTRAN

Array size: $a[M,N]$

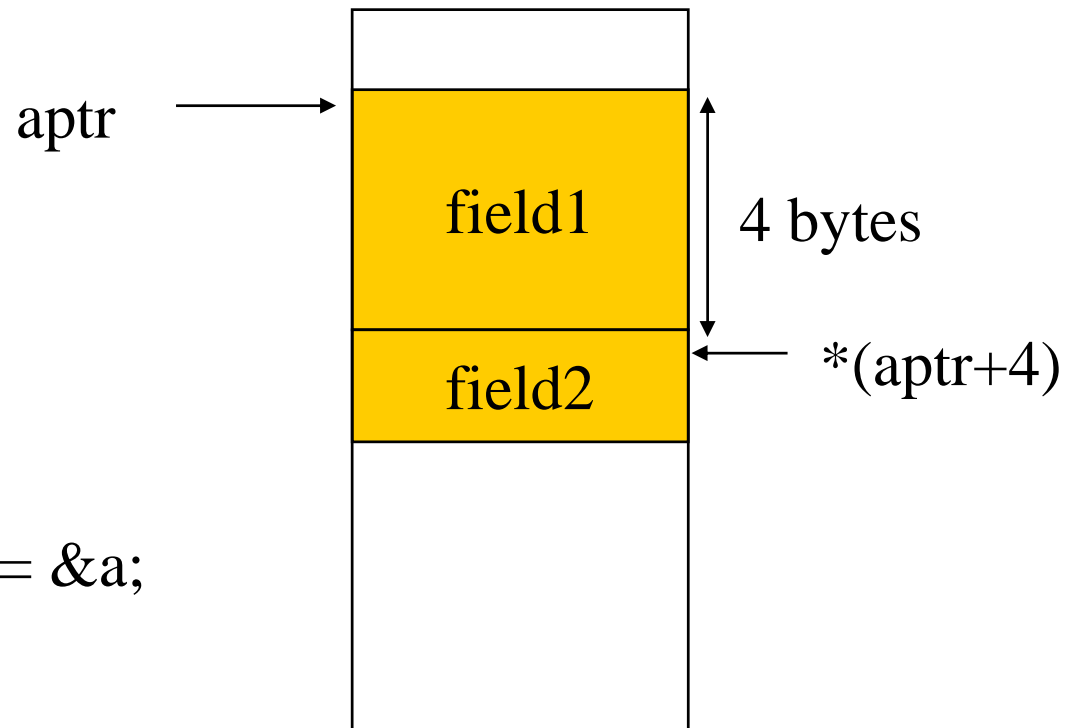


Structures

⌘ Fields within structures are static offsets:

```
struct {  
    int field1;  
    char field2;  
} mystruct;
```

```
struct mystruct a, *aptr = &a;
```



Expression simplification



⌘ Machine independent transformation

⌘ Constant folding:

☐ $8 + 1 = 9$

⌘ Algebraic:

☐ $a * b + a * c = a * (b + c)$

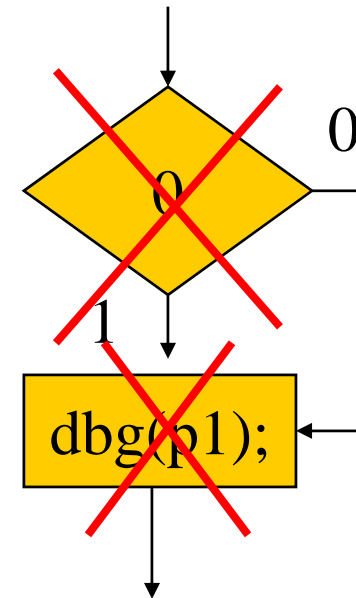
⌘ Strength reduction:

☐ $a * 2 = a \ll 1$

Dead code elimination

- ⌘ Dead code: difficult to identify in general
- ⌘ Can be eliminated by analysis of control flow.
- ⌘ a special case

```
#define DEBUG 0  
if (DEBUG) dbg(p1);
```



Procedure inlining



- ⌘ Eliminates procedure linkage overhead:
- ⌘ Increase code size

```
int foo(a,b,c) { return a + b - c; }  
z = foo(w,x,y);
```



```
z = w + x + y;
```

Loop transformations



⌘ Goals:

- ⏏ reduce loop overhead;
- ⏏ increase opportunities for pipelining;
 - ⏏ Reduce pipeline stalls
- ⏏ improve memory system performance.

Loop unrolling

⌘ Reduces loop overhead, enables some other optimizations.

⌘ Expose parallelism

```
for (i=0; i<4; i++)  
    a[i] = b[i] * c[i];
```



```
for (i=0; i<2; i++) {  
    a[i*2] = b[i*2] * c[i*2];  
    a[i*2+1] = b[i*2+1] * c[i*2+1];  
}
```

Loop fusion and distribution

⌘ Fusion combines two loops into 1:

```
for (i=0; i<N; i++) a[i] = b[i] * 5;  
for (j=0; j<N; j++) w[j] = c[j] * d[j];  
⇒ for (i=0; i<N; i++) {  
    a[i] = b[i] * 5;  
    w[i] = c[i] * d[i];  
}
```

⌘ Loop distribution breaks one loop into two.

⌘ Both changes optimizations within loop body.

Loop tiling

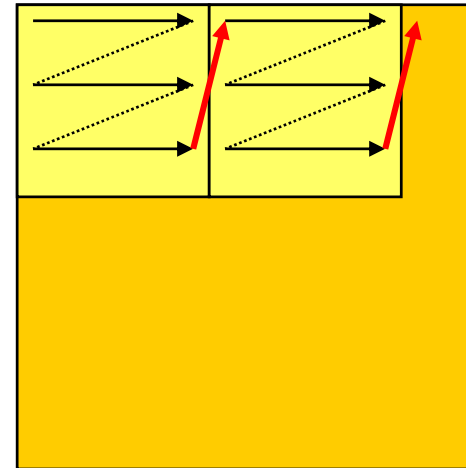
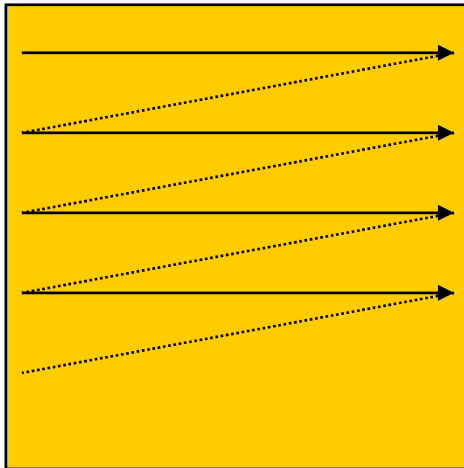


- ⌘ Breaks one loop into a nest of loops.
- ⌘ Changes order of accesses within array.
 - ☑ Changes cache behavior: *why?*

Loop tiling example

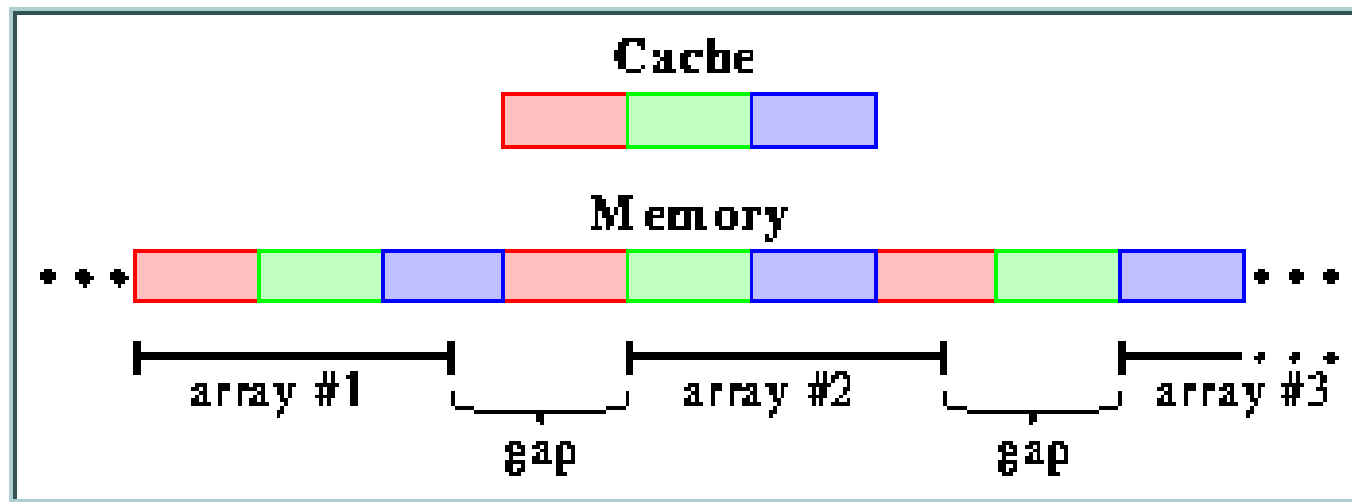
```
for (i=0; i<N; i++)  
  for (j=0; j<N; j++)  
    c[i] = a[i,j]*b[i];
```

```
for (i=0; i<N; i+=k)  
  for (j=0; j<N; j+=k)  
    for (ii=0; ii<min(i+k,n); ii++)  
      for (jj=0; jj<min(j+k,N); jj++)  
        c[ii] = a[ii,jj]*b[ii];
```



Array padding

- ⌘ Add array elements to change mapping into cache, which reduces cache conflict:



Code generation



⌘ Code selection

- ☑ Tree parsing

⌘ Instruction scheduling

- ☑ List scheduling

⌘ Register allocation

- ☑ graph coloring

Register allocation



⌘ Goals:

- ☑ choose register to hold each variable;
- ☑ determine lifespan of variable in the register.

⌘ Basic case: within basic block.

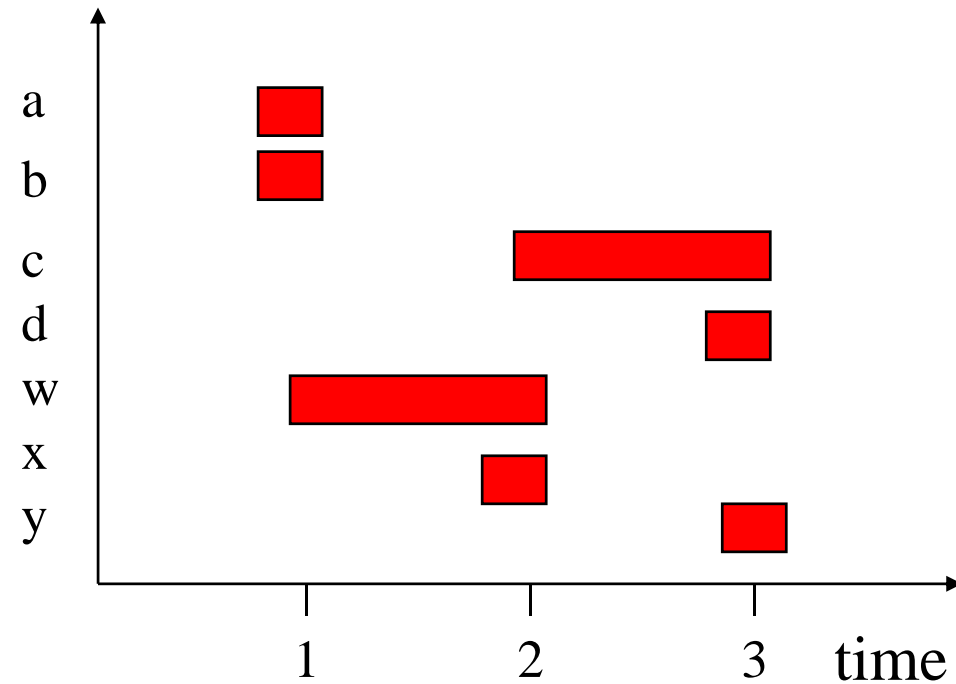
⌘ Spilling registers: problematic

Register lifetime graph

$w = a + b;$ $t=1$

$x = c + w;$ $t=2$

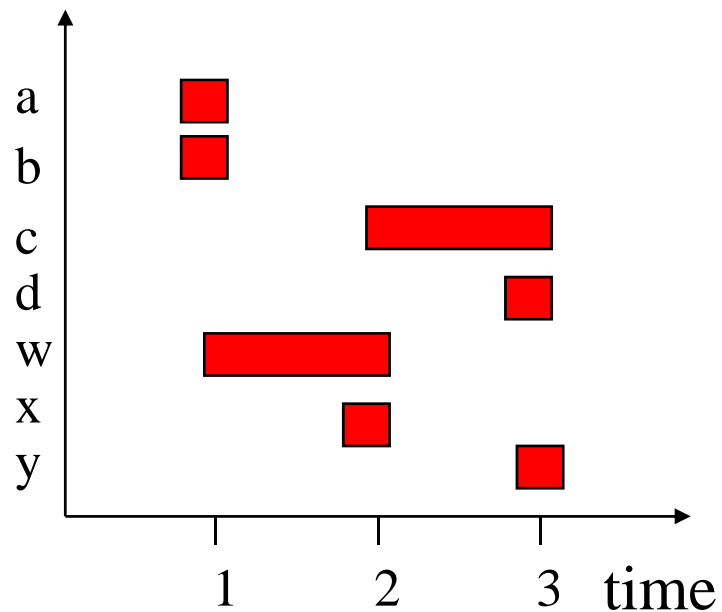
$y = c + d;$ $t=3$



Register assignment

a r0; b r1; c r2; d r0; w r3; x r0; y r3

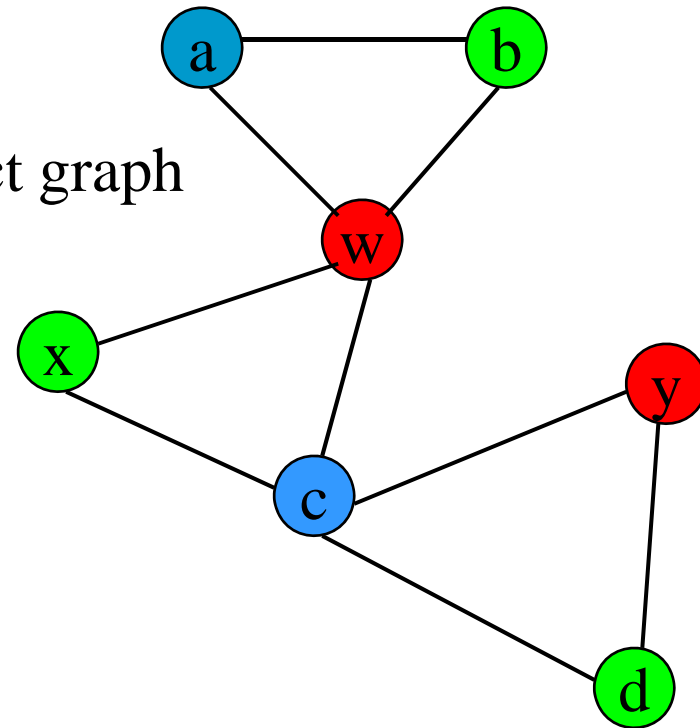
Conflict graph



Register assignment

a r0; b r1; c r2; d r0; w r3; x r0; y r3

Conflict graph



Minimum coring problem

Instruction scheduling



- ⌘ Non-pipelined machines do not need instruction scheduling: any order of instructions that satisfies data dependencies runs equally fast.
- ⌘ In pipelined machines, execution time of one instruction depends on the nearby instructions: **opcode, operands.**

Reservation table

⌘ A reservation table is used to relate instructions/time to CPU resources.

Time/instr	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

List scheduling for instruction scheduling

- ⌘ Greedy heuristic algorithm: most common in practice
- ⌘ Data-ready instructions stored in a **priority list**
- ⌘ Priorities assigned according to **heuristics**
 - ⊞ pick an instruction with the largest number of successors
 - ⊞ Pick instruction on the critical path or minimal slack
 - ⊞ Pick long latencies instructions

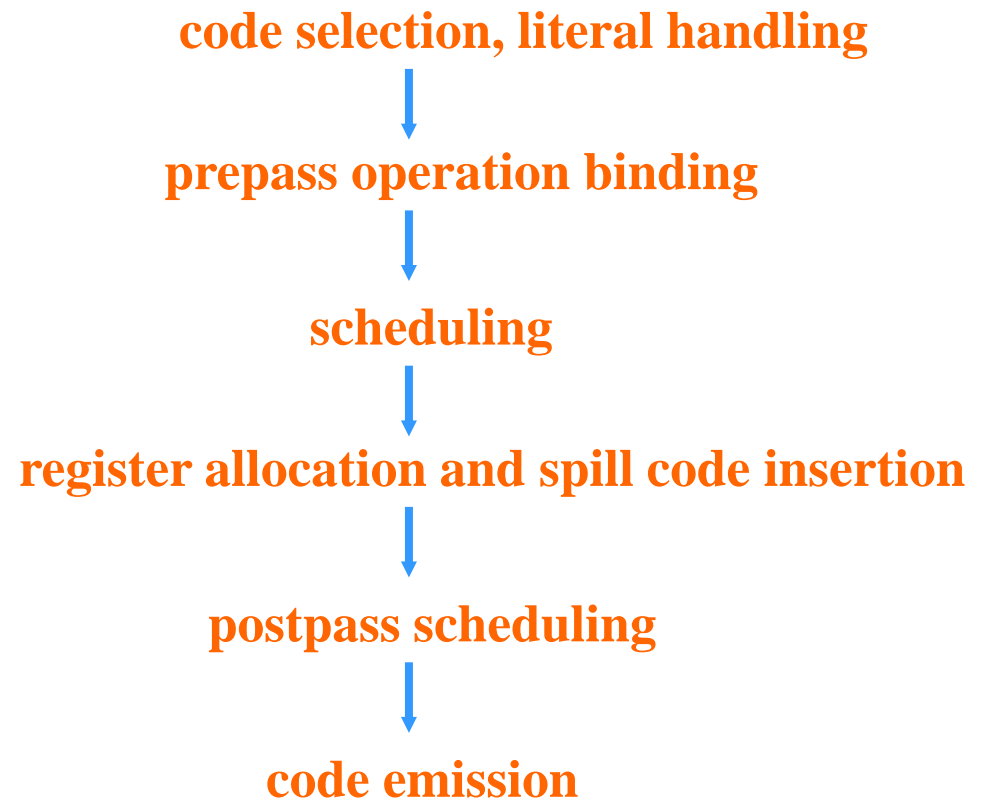
```
if ready list is not empty
    schedule top priority instruction
else
    schedule a stall;
advance to next issue slot
```

Compiler Code Generation

⌘ Schedule both before and after register allocation

☑ Initial scheduling is free of real processor register constraints

☑ 2nd phase required due to spill code



A Motivating Example

⌘ Machine model: **one** memory access (1-cycle), **one** arithmetic operation (2-cycle) in parallel

⌘ Source code: Do-All style loop

```
for (i=0; i < n; i++)
```

```
    A[i] = A[i] * b + c
```

⌘ Code for one iteration: 6 cycles/iteration

cycle 1: Read

cycle 2: Mul ti pl y

cycle 3:

cycle 4: Add

cycle 5:

cycle 6: Wri te

Loop unrolling

⌘ **Unrolling** replaces the body of the loop by several copies of the body and adjusts loop-control code

☒ Degree of unrolling = number of loop bodies

⌘ **Unrolling once and schedule**: 7 cycles/2 iterations

1:	Read	
2:	Mul	Read
3:		Mul
4:	Add	
5:		Add
6:	Write	
7:		Write

⌘ **Unrolling twice and schedule**: 10 cycles/3 iterations

Impact of Unrolling

⌘ What would be the **optimal performance** of this loop?

☒ 2 cycles/iteration (why? Consider resource constraints only)

⌘ **Impact of unrolling**: Let u be the degree of unrolling

☒ Execution Time of unrolled loop = $6 + 2(u - 1) = 4 + 2u$

☒ Optimal execution time = $2u$

☒ Efficiency = $\frac{2u}{4 + 2u}$

☒ Efficiency = 90 % $\Rightarrow u = 18$

⌘ More you unroll, it become better, but the code size increases substantially

Software Pipelining (SP)

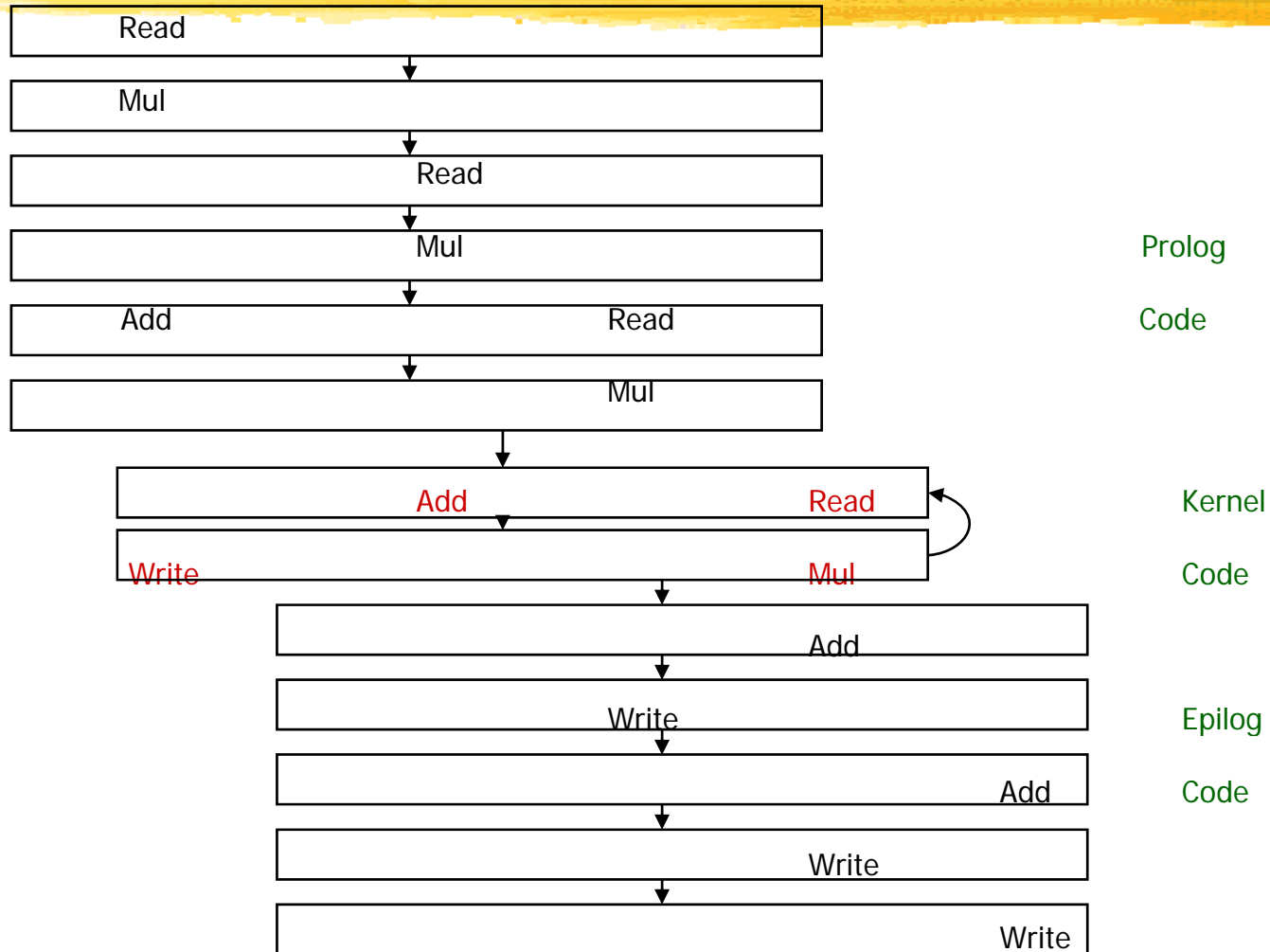
- ⌘ An optimization technique that can schedule instructions **beyond loop iteration boundaries**
 - ☑ By overlapping iterations in a pipelined fashion
 - ☑ Multiple iterations can be executed in parallel
 - ☑ Future iterations can initiate before current ones finish

- ⌘ Generating a pipelined schedule in overlapped iterations
 - ☑ Must find a pattern of code composed of multiple iterations that can be executed repeatedly, which is called a **kernel**

Finding a Kernel in Overlapped Code

	1 st iteration	2 nd iteration	3 rd iteration	4 th iteration	5 th iteration
1:	Read				
2:	Mul				
3:		Read			
4:		Mul			
5:	Add		Read		
6:			Mul		
7:		Add		Read	: repeated
8:	Write			Mul	: pattern
9:			Add		Read repeated
10:		Write			Mul pattern
11:				Add	
12:			Write		
13:					Add
14:				Write	
15:					
16:					Write

Generating a Pipelined Schedule



Software Pipelined Loop

⌘ A software pipelined loop is composed of:

- ⊞ **Prolog**: pipeline startup code
- ⊞ **Kernel**: repeated pattern that is executed repetitively
- ⊞ **Epilog**: pipeline drain code

⌘ **Initiation interval (II)**

- ⊞ Interval with which the next iteration initiates start after the current iteration initiates
- ⊞ Equals to the **cycle length of the kernel**
- ⊞ In our example schedule, $II = 2$ cycles

Benefit of Software Pipelining

- ⌘ Unlike unrolling, software pipelining can give you an **optimal** result
- ⌘ Code size is much **smaller** than unrolling

- ⌘ Schedule of each iteration
 - ☒ Schedule of each iteration is **identical**
 - ☒ For finding a pattern easily and quickly
 - ☒ Locally compacted code might not be globally optimal

SP Across Loops

⌘ Source Code

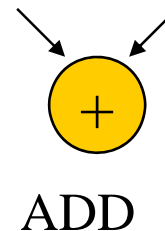
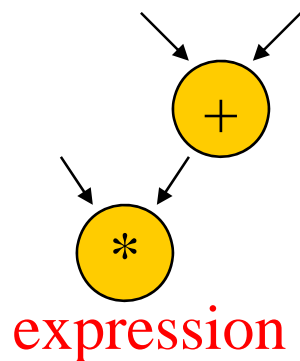
for (i=0; i < n; i++)	1: read
Sum = Sum + A[i]	2: Mult
A[i] = A[i] * b	3: Add
	4: Write

⌘ Software pipelined Code

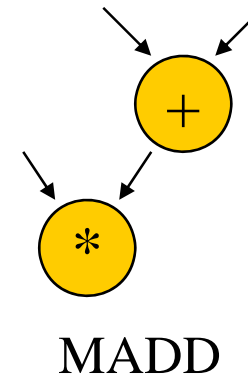
1:	Read		
2:	Mul		
3:	Add	Read	kernel
4:	Write	Mul	
5:		Add	
6:		Write	

Instruction selection

- ⌘ May be several ways to implement an operation or sequence of operations.
- ⌘ Represent operations as graphs, match possible instruction sequences onto graph.



templates



Using your compiler?



- ⌘ Understand various optimization levels (-O1, -O2, etc.)
- ⌘ Look at mixed compiler/assembler output.
- ⌘ Modifying compiler output requires care:
 - ☒ correctness;
 - ☒ loss of hand-tweaked code.

Interpreters and JIT compilers



- ⌘ **Interpreter**: translates and executes program statements on-the-fly.
- ⌘ **JIT compiler**: compiles small sections of code into instructions during program execution.
 - ☒ Eliminates some translation overhead.
 - ☒ Often requires more memory.

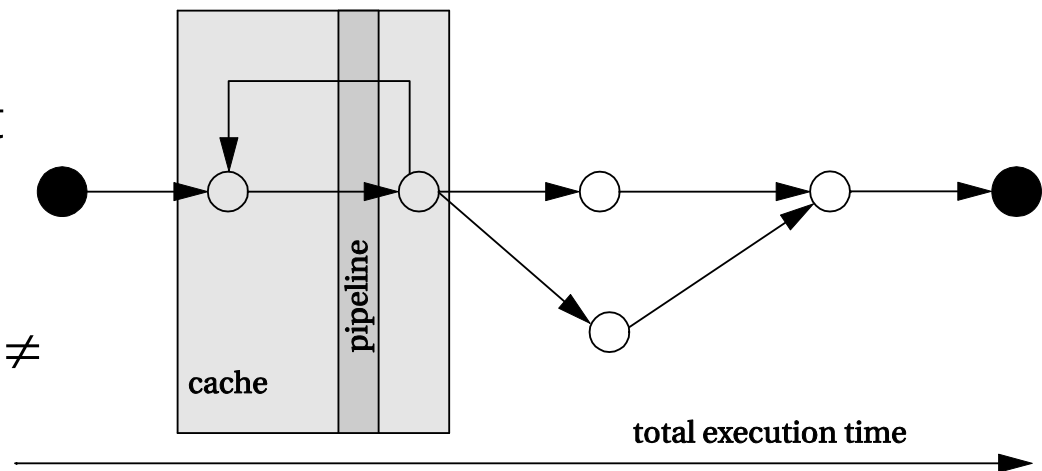
Program design and analysis



- ⌘ Program-level performance analysis.
- ⌘ Optimizing for:
 - ☑ Execution time.
 - ☑ Energy/power.
 - ☑ Program size.
- ⌘ Program validation and testing.

Program performance analysis

- ⌘ Need to understand performance in detail:
 - ☑ Real-time behavior, not just typical.
 - ☑ On complex platforms.
- ⌘ Program performance \neq CPU performance:
 - ☑ Pipeline, cache are windows into program.
 - ☑ We must analyze the entire program.




Complexities of program performance



- ⌘ Varies with input data:
 - ☑ Different-length paths.
- ⌘ Cache effects.
- ⌘ Instruction-level performance variations:
 - ☑ Pipeline interlocks.
 - ☑ Fetch times.

How to measure program performance



- ⌘ Simulate execution of the CPU.
 - ☑ Makes CPU state visible.
- ⌘ Measure on real CPU using timer.
 - ☑ Requires modifying the program to control the timer.
- ⌘ Measure on real CPU using logic analyzer.
 - ☑ Requires events visible on the pins.

Program performance metrics



- ⌘ Average-case execution time.
 - ☑ Typically used in application programming.
- ⌘ Worst-case execution time.
 - ☑ A component in deadline satisfaction.
- ⌘ Best-case execution time.
 - ☑ Task-level interactions can cause best-case program behavior to result in worst-case system behavior.

Elements of program performance

⌘ Basic program execution time formula:

☑ execution time = program path + instruction timing

⌘ Solving these problems independently helps simplify analysis.

☑ Easier to separate on simpler CPUs.

⌘ Accurate performance analysis requires:

☑ Assembly/binary code.

☑ Execution platform.

Data-dependent paths in an if statement

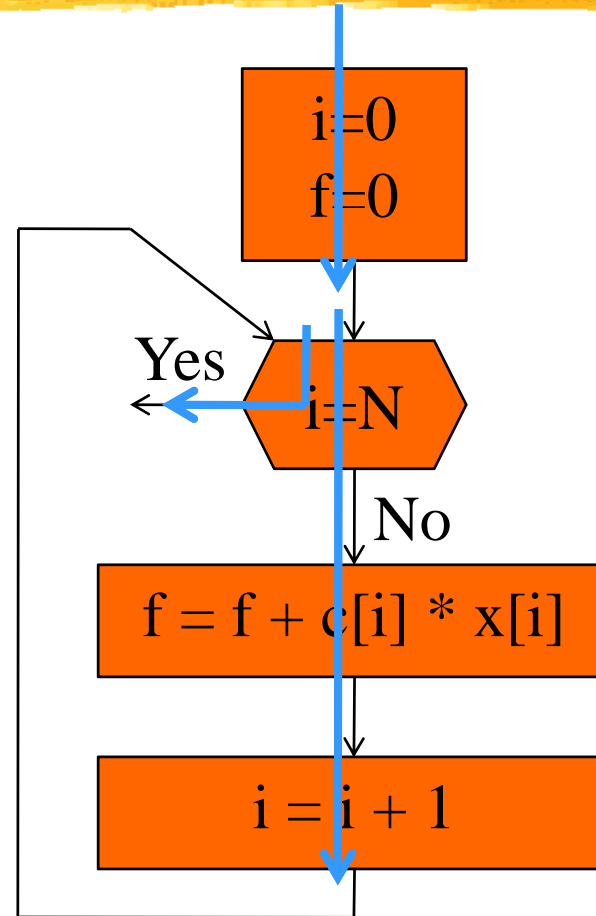
```
if (a || b) { /* T1 */
    if ( c ) /* T2 */
        x = r*s+t; /* A1 */
    else y=r+s; /* A2 */
    z = r+s+u; /* A3 */
}
else {
    if ( c ) /* T3 */
        y = r-t; /* A4 */
}
```

a	b	c	path
0	0	0	T1=F, T3=F: no assignments
0	0	1	T1=F, T3=T: A4
0	1	0	T1=T, T2=F: A2, A3
0	1	1	T1=T, T2=T: A1, A3
1	0	0	T1=T, T2=F: A2, A3
1	0	1	T1=T, T2=T: A1, A3
1	1	0	T1=T, T2=F: A2, A3
1	1	1	T1=T, T2=T: A1, A3

Paths in a loop



```
for (i=0, f=0; i<N; i++)  
    f = f + c[i] * x[i];
```



Performance estimation



- ⌘ Once we know the execution path the simplest estimate is
 - ☑ Assume that every instruction takes the same number of clock cycles
 - ☑ Multiply the count of instructions with the per-instruction execution time

Instruction timing



- ⌘ Not all instructions take the same amount of time.
 - ⊞ Multi-cycle instructions
 - ⊞ Multiple load or store instructions
 - ⊞ Floating point instructions
- ⌘ Execution times of instructions are not independent.
 - ⊞ Register bypassing
- ⌘ Execution times may vary with operand value.
 - ⊞ Floating-point operations.
 - ⊞ Some multi-cycle integer operations.

Measurement-driven performance analysis



- ⌘ More direct way

- ⌘ Not so easy as it sounds:

 - ☑ Must actually have access to the CPU.

 - ☑ Must know data inputs that give worst/best case performance.

 - ☑ Must make state visible.

- ⌘ Still an important method for performance analysis.

Trace-driven measurement



⌘ Trace-driven:

- ☑ Instrument the program.

- ☑ Save information about the path.

⌘ Requires modifying the program.

⌘ Trace files are large.

⌘ Widely used for cache analysis.

Feeding the program



- ⌘ The biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program
 - ☑ Need to know the desired input values.
 - ☑ May need to write **software scaffolding** to feed data into the program and get data out.
- ⌘ Software scaffolding may also need to examine outputs to generate feedback-driven inputs.

Performance measurement



- ⌘ Directly on hardware
- ⌘ By using a simulator

Physical measurement



- ⌘ In-circuit emulator allows tracing.
 - ☑ Affects execution timing.
- ⌘ Logic analyzer can measure behavior at pins.
 - ☑ Address bus can be analyzed to look for events.
 - ☑ Code can be modified to make events visible.
- ⌘ Particularly important for real-world input streams.

CPU simulation



- ⌘ Some simulators are less accurate.
- ⌘ Cycle-accurate simulator provides accurate clock-cycle timing.
 - ☑ Simulator models CPU internals.
 - ☑ Simulator writer must know how CPU works.
- ⌘ SimpleScalar (<http://www.simplescalar.com>): a framework for building cycle-accurate CPU models.

SimpleScalar FIR filter simulation

```
int x[N] = {8, 17, ... };
int c[N] = {1, 2, ... };
main() {
    int i, k, f;
    for (k=0; k<COUNT; k++)
        for (i=0; i<N; i++)
            f += c[i]*x[i];
}
```

N	total sim cycles	sim cycles per filter execution
100	25854	259
1,000	155759	156
1,0000	1451840	145

Performance optimization motivation



- ⌘ Embedded systems must often meet deadlines.
 - ☑ Faster may not be fast enough.
- ⌘ Need to be able to analyze execution time.
 - ☑ Worst-case, not typical.
- ⌘ Need techniques for reliably improving execution time.

Programs and performance analysis



- ⌘ Best results come from analyzing optimized instructions, not high-level language code:
 - ☑ non-obvious translations of HLL statements into instructions;
 - ☑ code may move;
 - ☑ cache effects are hard to predict.

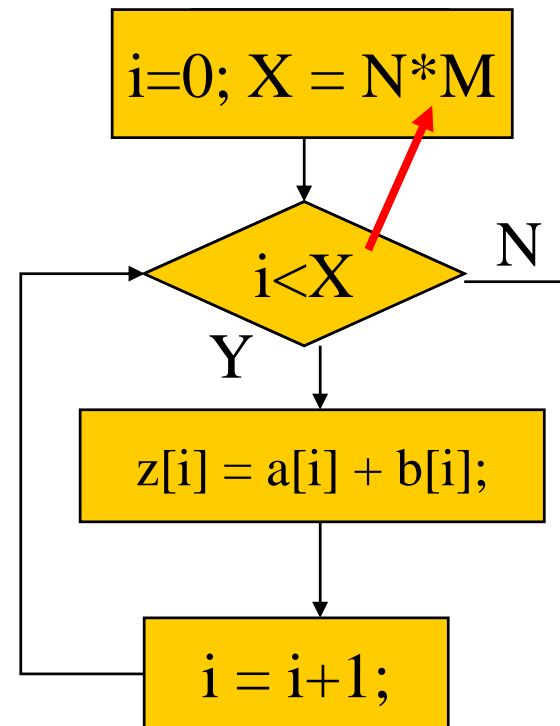
Loop optimizations



- ⌘ Loops are good targets for optimization.
- ⌘ Basic loop optimizations:
 - ☑ loop invariant code motion
 - ☑ induction-variable elimination;
 - ☑ strength reduction ($x*2 \rightarrow x \ll 1$).

Code motion

```
for (i=0; i<N*M; i++)  
    z[i] = a[i] + b[i];
```



Induction variable



- ⌘ a variable that gets increased or decreased by a fixed amount on every iteration of a loop,
- ⌘ or is a linear function of another induction variable.
- ⌘ The compiler can eliminate some induction variables and apply strength reduction to others

Induction variable elimination

⌘ Consider loop:

```
for (i=0; i<N; i++)  
  for (j=0; j<M; j++)  
    z[i,j] = b[i,j];
```

⌘ Introduce an induction variable

```
for (i=0; i<N; i++)  
  for (j=0; j<M; j++) {  
    k = i*M + j; // induction variables  
    z[k] = b[k];  
  }
```


Induction variable elimination

- ⌘ Rather than recompute $i * M + j$ for each array in each iteration, share induction variable between arrays, increment at end of loop body.

```
k=0;
for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
        z[k] = b[k];
        k++; // a strength reduction
    }
}
```

Cache analysis



- ⌘ **Loop nest**: set of loops, one inside other.
- ⌘ **Perfect loop nest**: no conditionals in nest.
- ⌘ Because loops use large quantities of data, cache conflicts are common.

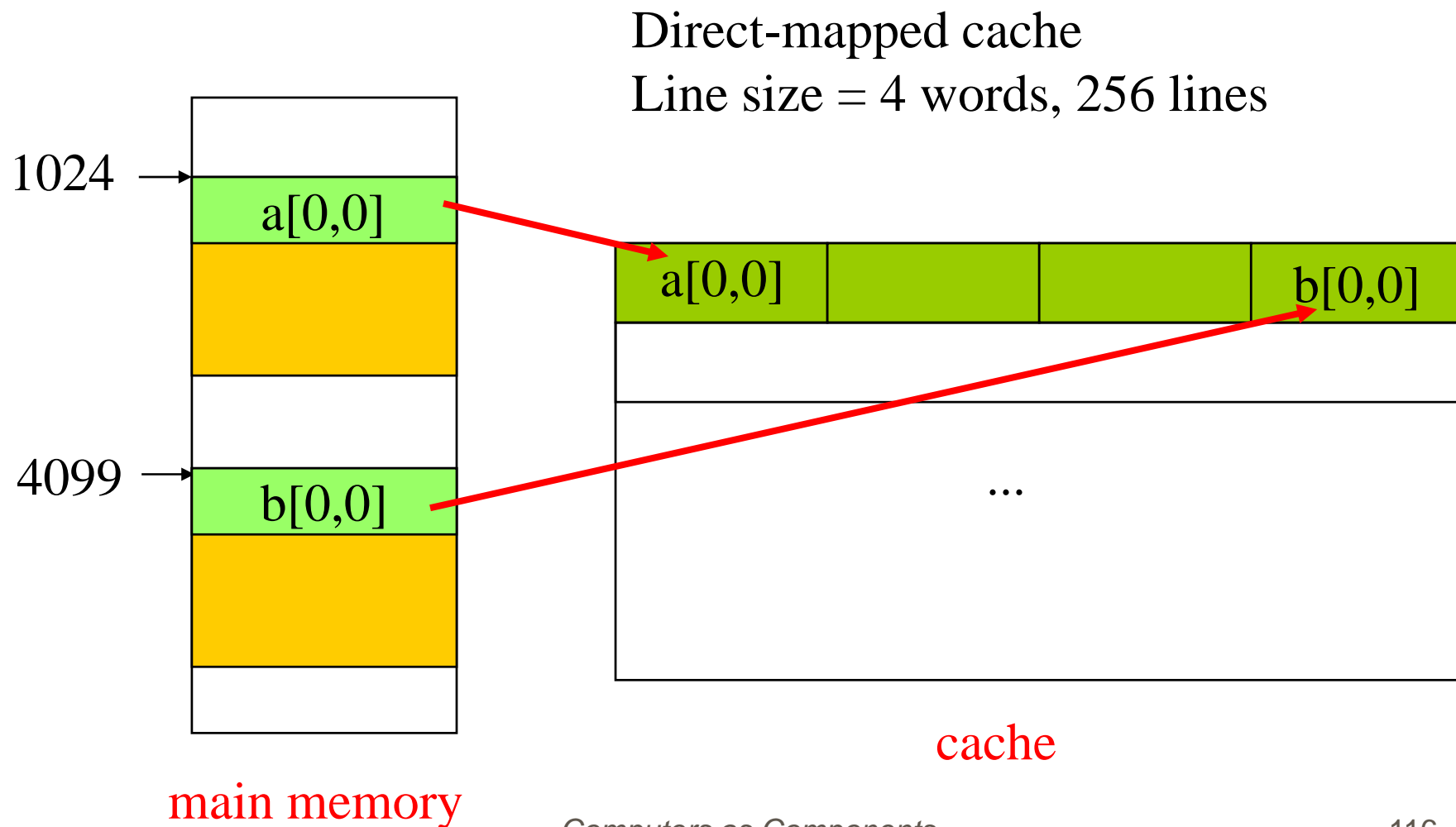
Array conflicts in cache

```
for (i=0; i<N; i++) {                               // N=256
    for (j=0; j<M; j++) {                             // M = 4
        a[i][j] = b[i][j] * c;
    }
}
```

Four-way set-associative cache

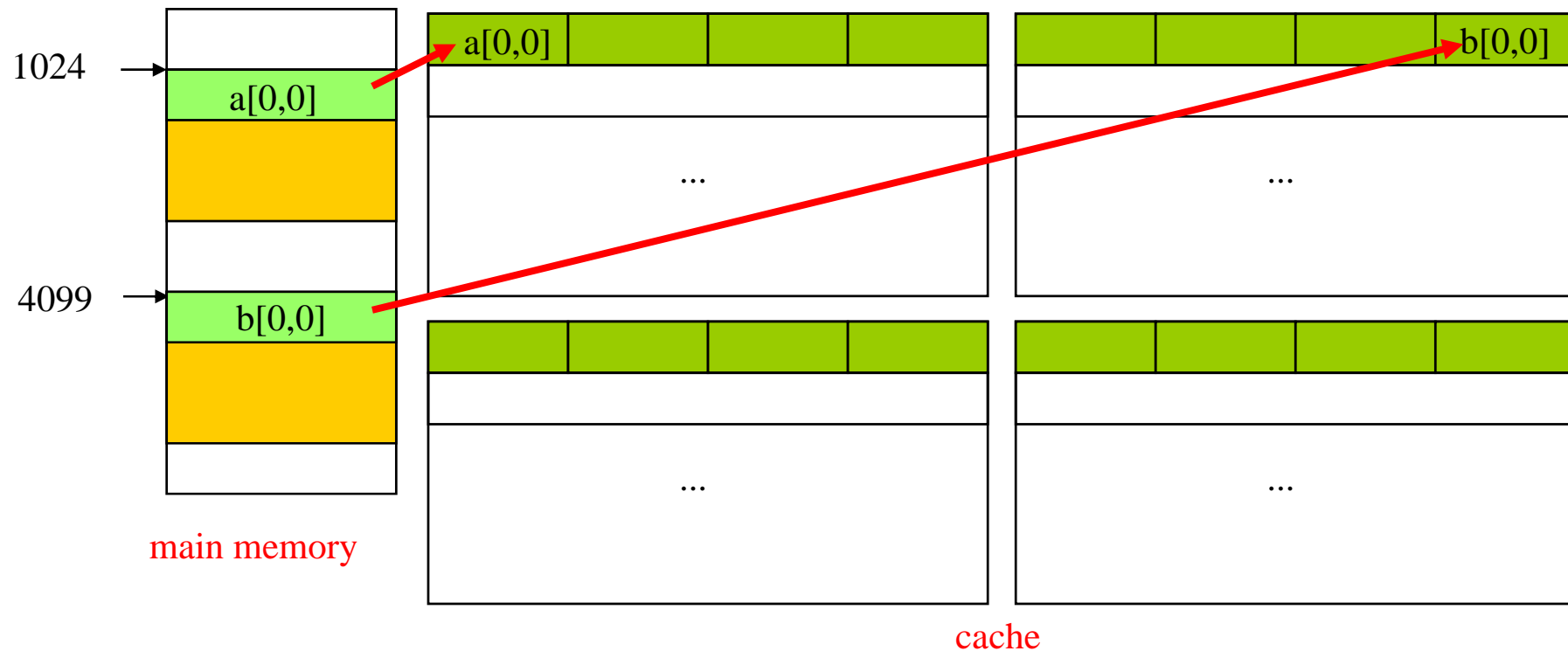
Line size = 4 words, 256 lines

Array conflicts in cache



Array conflicts in cache

Four-way set-associative cache
Line size = 4 words, 256 lines



Array conflicts



- ⌘ Array elements conflict because they are mapped into the same line, even if not mapped to the same location.
- ⌘ Solutions:
 - ☑ move one array;
 - ☑ pad array.

Performance optimization hints



- ⌘ Profiling the program to find hot spots
- ⌘ A profiler does not measure execution time
- ⌘ Two major ways to profile a program
 - ☒ add an counting instruction at a location, which increments every time the program passes that point
 - ☒ or sample the pc during execution and keep track of the distribution of the pc values.
- ⌘ Profiling add relatively little overhead to the program

Performance optimization hints



⌘ Use registers efficiently.

- ☑ group accesses to a value together

⌘ Use page mode memory accesses.

- ☑ to reduce the latency of the memory accesses

- ☑ rearrange variables so that they can be referenced contiguously

Performance optimization hints



⌘ Analyze cache behavior:

- ☑ instruction conflicts can be handled by rewriting a small code to make it smaller,
- ☑ move the instructions or pad with NOP instructions
- ☑ conflicting scalar data can easily be moved;
- ☑ conflicting array data can be moved, padded.

Cache behavior is important



- ⌘ Energy consumption has a **sweet spot** as cache size changes:
 - ☒ **cache too small**: program thrashes, burning energy on external memory accesses;
 - ☒ **cache too large**: cache itself burns too much power.

Optimizing for energy



⌘ First-order optimization:

☑ high performance = low energy.

⌘ Making the program run faster also reduces energy consumption

⌘ Memory access patterns: can be controlled by the programmers

Optimizing for energy



- ⌘ Use registers efficiently.
- ⌘ Identify and eliminate cache conflicts.
- ⌘ Moderate loop unrolling eliminates some loop overhead instructions.
- ⌘ Eliminate pipeline stalls.
- ⌘ Inlining procedures may help: reduces linkage, but may increase cache thrashing.

Optimizing for program size



⌘ Goal:

- ☑ reduce hardware cost of memory;
- ☑ reduce power consumption of memory units.

⌘ Two opportunities:

- ☑ data;
- ☑ instructions.

Data size minimization



- ⌘ Reuse constants, variables, data buffers in different parts of code.
 - ☑ Requires careful verification of correctness.
- ⌘ Generate data using instructions.

Reducing code size



- ⌘ Avoid function inlining.
- ⌘ Choose CPU with compact instructions.
- ⌘ Use specialized instructions where possible.

Program validation



⌘ The goal of validating the requirement and specification is to ensure that they satisfy the following criteria (ref. 446 page)

- ☑ correctness
- ☑ unambiguousness
- ☑ completeness
- ☑ verifiability
- ☑ consistency
- ☑ modifiability
- ☑ traceability

Program testing



- ⌘ does it work?
- ⌘ Concentrate here on functional verification.
- ⌘ Create a good set of tests for a given program
- ⌘ How much testing is enough?
- ⌘ Major testing strategies:
 - ☑ Black box doesn't look at the source code.
 - ☑ Clear box (white box) does look at the source code.

White-box testing



- ⌘ CDFG is an important tool
- ⌘ Examine the source code to determine whether it works:
 - ☑ Can you actually exercise a path?
 - ☑ Do you get the value you expect along a path?
- ⌘ Testing procedure:
 - ☑ **Controllability**: provide program with inputs.
 - ☑ Execute.
 - ☑ **Observability**: examine outputs.

Example



```
firout = 0.0;
for (j=curr, k=0; j<N; j++, k++)
    firout += buff[j] * c[k];
for (j=0; j<curr; j++, k++)
    firout += buff[j] * c[k];
if (firout > 100.0) firout = 100.0;
if (firout < -100.0) firout = -100.0;
```

⌘ Controllability:

- ☒ Must fill circular buffer with **proper** N values.

⌘ Observability:

- ☒ Want to examine firout before limit testing.

How to determine test sets



- ⌘ Can we test every path in an arbitrary program? No
- ⌘ Does it make sense to exercise every path? No
- ⌘ The choice of an appropriate subset of paths to be tested requires some thought.

Execution paths and testing



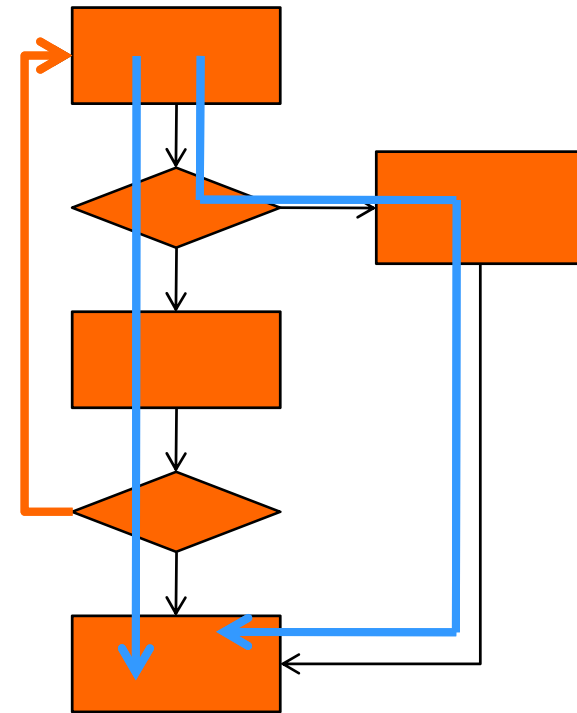
- ⌘ Paths are important in functional testing as well as performance analysis.
- ⌘ In general, an exponential number of paths through the program.
 - ☒ Show that some paths dominate others.
 - ☒ Heuristically limit paths.

Choosing the paths to test

⌘ Two reasonable choices:

- ☑ Execute every statement at least once.
- ☑ Execute every branch direction at least once.

not covered

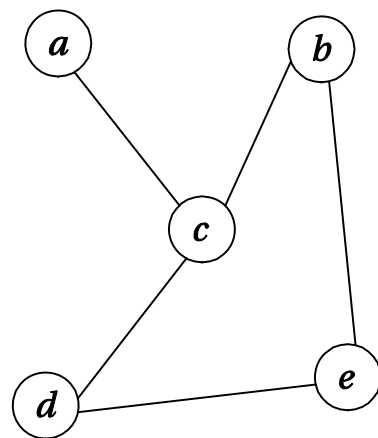


How to choose a set of paths

- ⌘ Intuition tells us that a relatively small number of paths should be able to cover most practical programs
- ⌘ Graph theory helps us get a quantitative handle
 - ☑ Cyclomatic complexity

Basis paths

- ⌘ Approximate CFG with undirected graph.
- ⌘ Undirected graphs have basis paths:
 - ☑ All paths are linear combinations of basis paths.



Graph

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	0	1	0	0
<i>b</i>	0	0	1	0	1
<i>c</i>	1	1	0	1	0
<i>d</i>	0	0	1	0	1
<i>e</i>	0	1	0	1	0

Incidence matrix

<i>a</i>	1	0	0	0	0
<i>b</i>	0	1	0	0	0
<i>c</i>	0	0	1	0	0
<i>d</i>	0	0	0	1	0
<i>e</i>	0	0	0	0	1

Basis set

Cyclomatic complexity

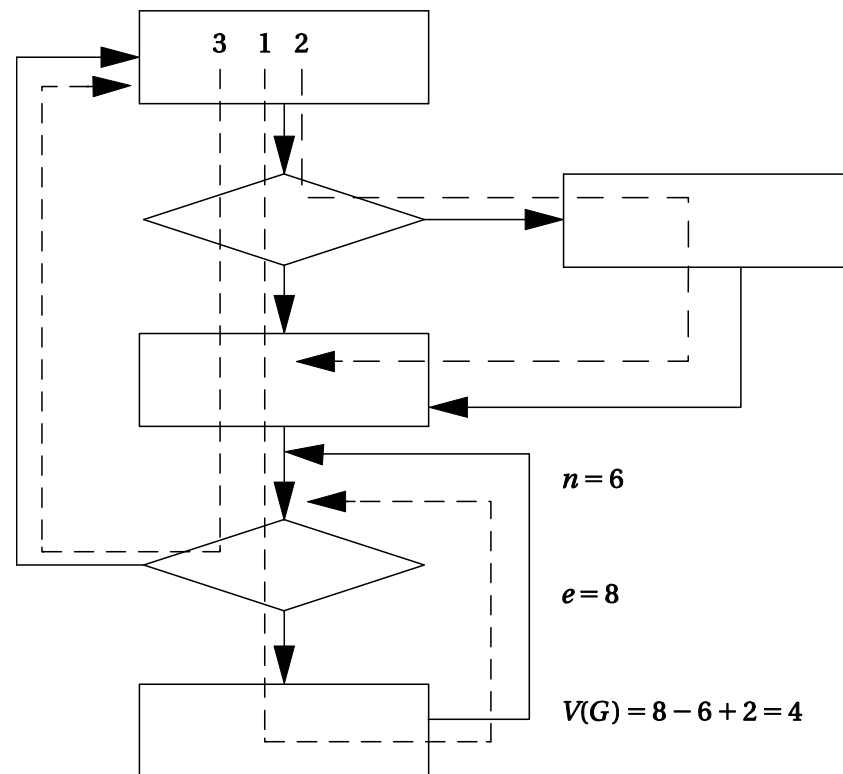
⌘ Cyclomatic complexity is a bound on the size of basis sets:

☒ $e = \# \text{ edges}$

☒ $n = \# \text{ nodes}$

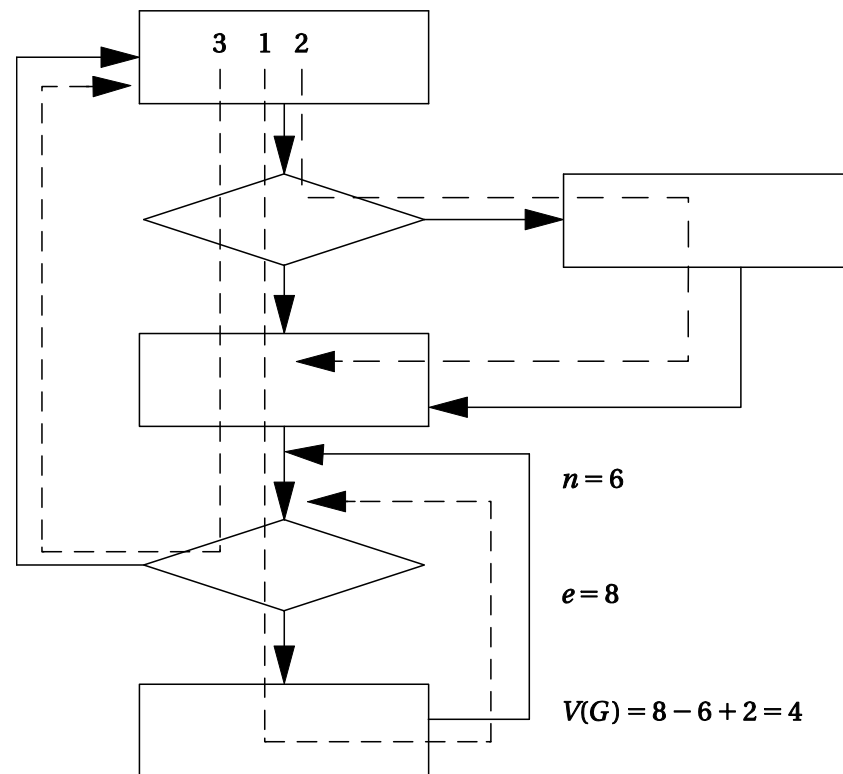
☒ $p = \text{number of graph components}$

☒ $M = e - n + 2p.$



Picking Basis Paths

- ⌘ Pick path through the code that covers the most edges (1)
- ⌘ Pick a new path that covers at least one new edge
- ⌘ Continue until the number of paths equals the cyclomatic complexity (i.e., number of basis tests)
- ⌘ Note: The basis paths through the code are not unique
- ⌘ Because there are actually three distinct paths in the graph, cyclomatic complexity in this case is an overly conservative bound



Branch testing



- ⌘ Heuristic for testing branches.
 - ☑ Exercise both true and false branches of the conditional.
 - ☑ Exercise every simple condition in the conditional's expression at least once.
- ⌘ One of the reasons to use many different types of test is to maximize the chance that supposedly unrelated elements will cooperate to reveal the error in a particular situation

Branch testing example

⌘ Correct:

```
if (a || (b >= c)) {  
    printf("OK\n");  
}
```

⌘ Incorrect:

```
if (a && (b >= c)) {  
    printf("OK\n");  
}
```

⌘ Branch Test input:

```
a = F
```

```
(b >= c) = T
```

⌘ Example:

```
Correct: [0 || (3 >= 2)] = T
```

```
Incorrect: [0 && (3 >= 2)] = F
```

so this test pick up the error

Another example

⌘ Correct:

```
⊞ if ((x == good_pointer) &&
    x->field1 == 3) {
    printf("got the value\n");
}
```

⌘ Incorrect:

```
⌘ if ((x = good_pointer) &&
    x->field1 == 3) {
    printf("got the value\n");
}
```

⌘ Incorrect code changes pointer.

⊞ Assignment returns new LHS in C.

⌘ A test we want to use

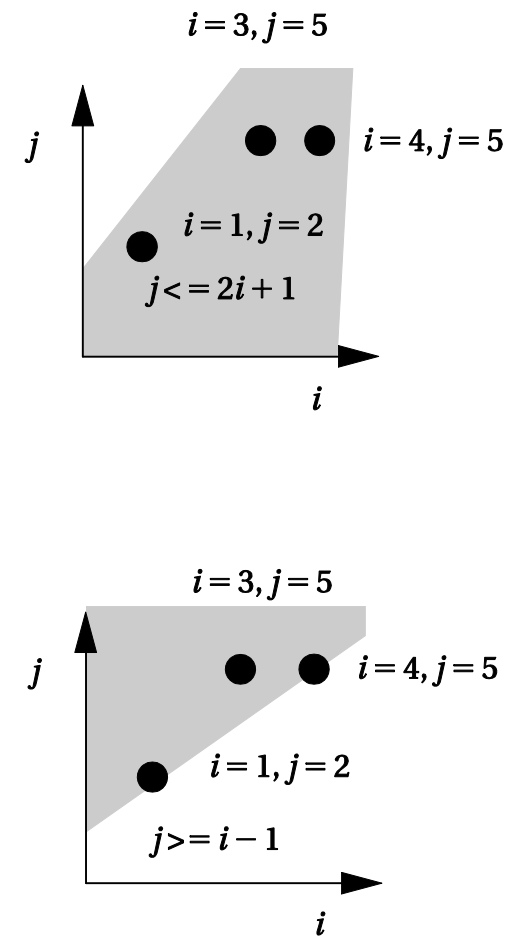
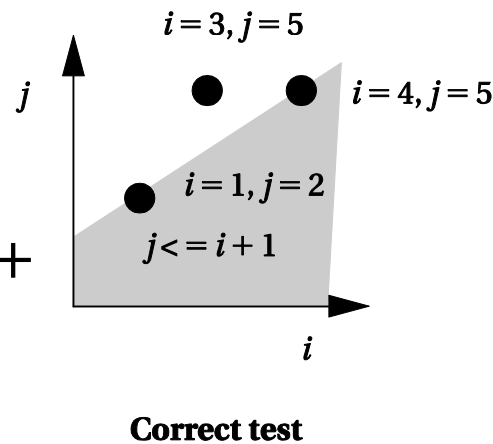
⊞ (x != good_pointer) && x->field1 = 3)

⊞ Not guaranteed to uncover the error

⊞ But reasonable chance of success

Domain testing

- Another sophisticated strategy for testing conditionals
- Heuristic test for linear inequalities.
- Test on each side + boundary of inequality.



A potential problem with path coverage



- ⌘ The paths chosen to cover the CFG may not have any important relationship with the program's function.
- ⌘ Data flow testing using def-use analysis selects paths with some relationship to the program's function

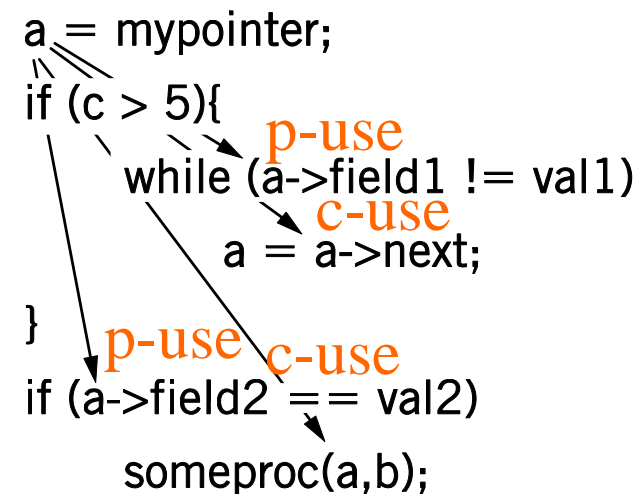
Def-use pairs

⌘ Variable def-use:

- ☑ Def when value is assigned (defined).
- ☑ Use when used on right-hand side.

⌘ Exercise each def-use pair.

- ☑ Requires testing correct path.



four def-use pairs

Loop testing



- ⌘ Loops need specialized tests to be tested efficiently.
- ⌘ Heuristic testing strategy:
 - ☑ Skip loop entirely if possible.
 - ☑ One loop iteration.
 - ☑ Two loop iterations.
 - ☑ # iterations much below max.
 - ☑ $n-1$, n , $n+1$ iterations where n is max.

Black-box testing



- ⌘ Complements clear-box testing.
 - ☑ May require a large number of tests.
- ⌘ Tests software in different ways.

Black-box test vectors



⌘ Random tests.

- ☑ May weight distribution based on software specification.

⌘ Regression tests.

- ☑ Tests of previous versions, bugs, etc.
- ☑ May be clear-box tests of previous versions.

How much testing is enough?



- ⌘ Exhaustive testing is impractical.
- ⌘ One important measure of test quality---bugs escaping into field.
- ⌘ Good organizations can test software to give very low field bug report rates.
- ⌘ Error injection measures test quality:
 - ☑ Add known bugs.
 - ☑ Run your tests.
 - ☑ Determine % injected bugs that are caught.