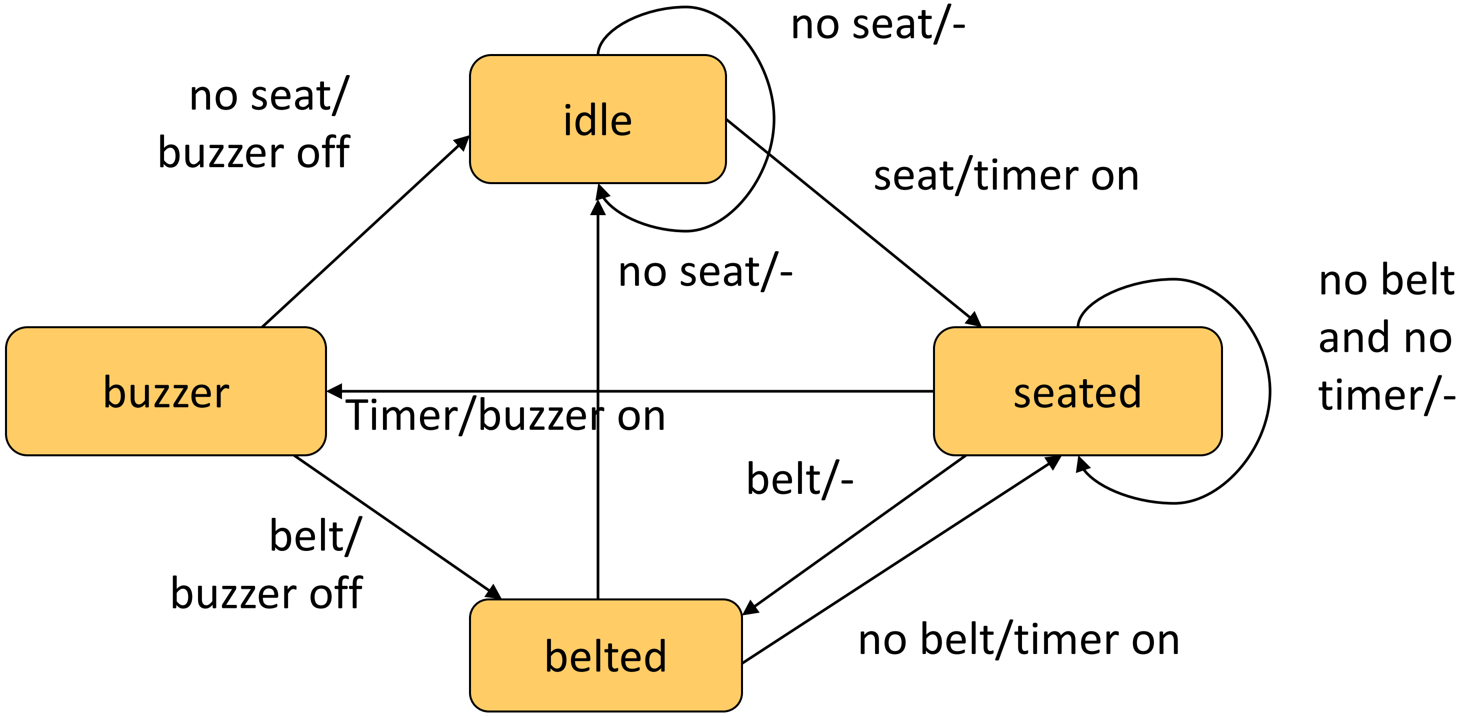
Program design and analysis

- Software components.
- Representations of programs.
- Assembly and linking.

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



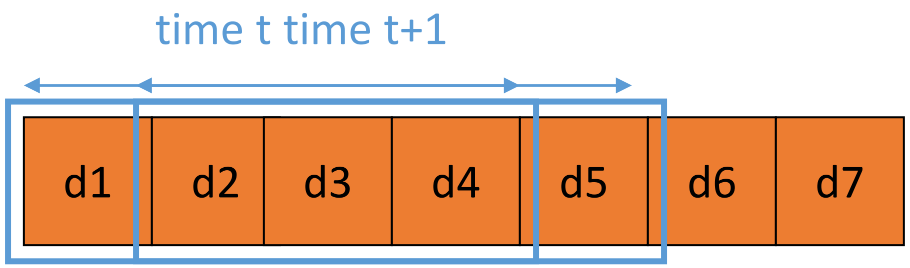
C implementation

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
   case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
            break;
   case SEATED: if (belt) state = BELTED;
                        else if (timer) state = BUZZER;
            break;
   …
}
```

```
switch(state) { /*check the current state */
        case IDLE:
        if (seat){ state = SEATED; timer_on = TRUE; }
        /*default case is self-loop */
        break;
        case SEATED:
        if (belt) state = BELTED; /*won't hear the buzzer */
        else if (timer) state = BUZZER; /*didn't put on
belt in time */
        /*default case is self-loop */
        break;
        case BELTED:
        if (!seat) state = IDLE; /* person left */
        else if (!belt) state = SEATED; /* person still
in seat */
        break;
        case BUZZER:
        if (belt) state = BELTED; /*belt is on---turn off
buzzer */
        else if (!seat) state = IDLE; /* no one in seat--
turn off buzzer */
        break;
}
```

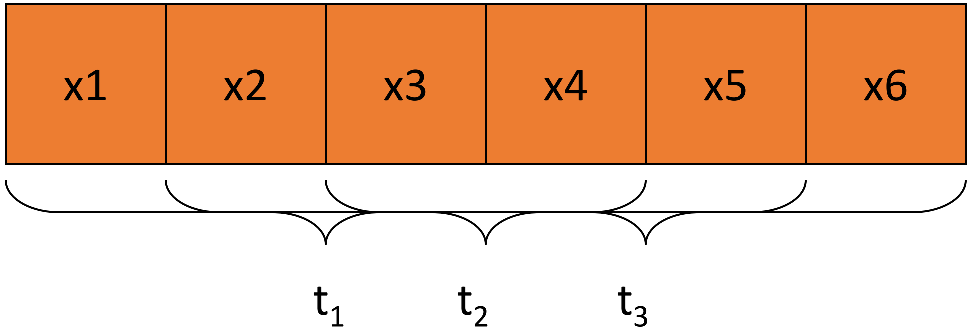
Signal processing and circular buffer

- Commonly used in signal processing:
  - new data constantly arrives;
  - each datum has a limited lifetime.

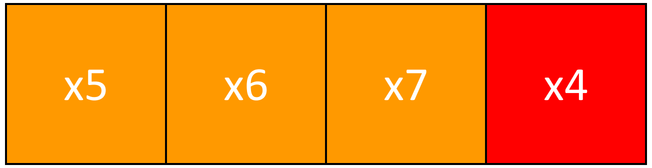


- Use a circular buffer to hold the data stream.

Circular buffer



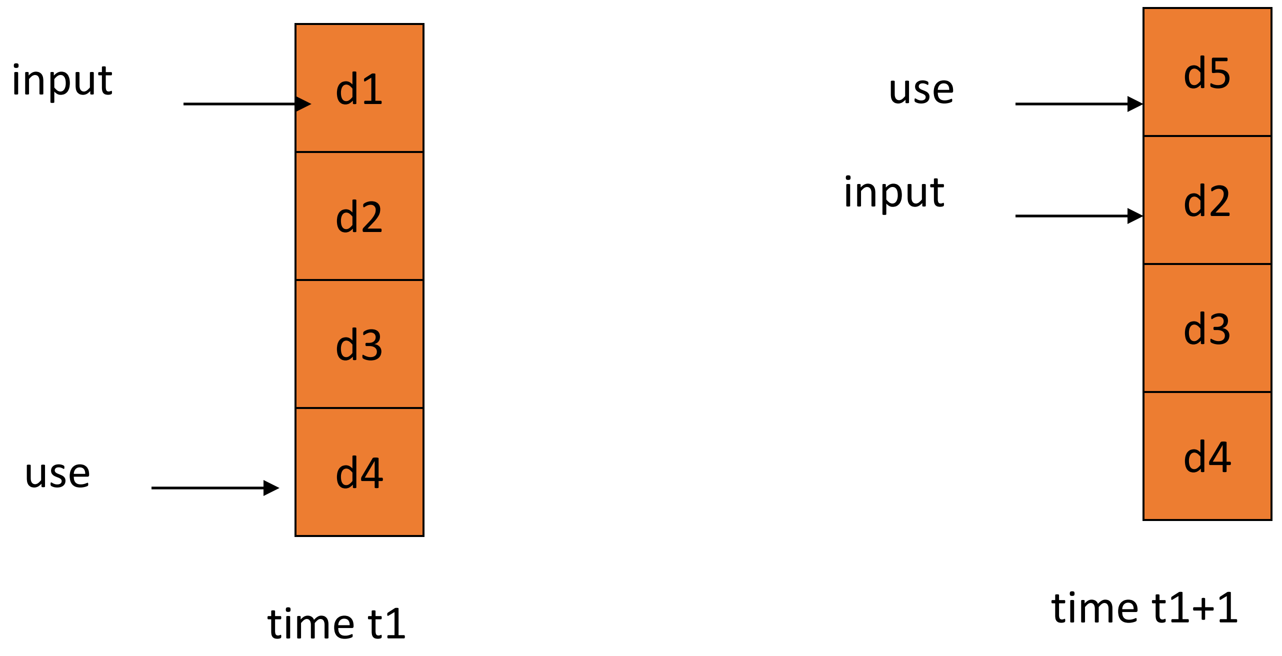
Data stream



Circular buffer

Circular buffers

- Indexes locate currently used data, current input data:



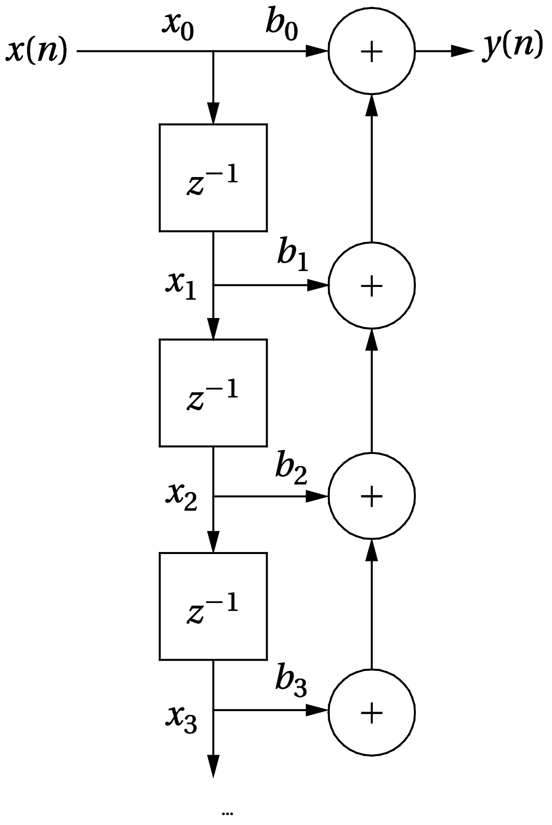
Circular buffer in C

```
#define CMAX 6 /* filter order */
int circ[CMAX]; /* circular buffer */
int pos; /* position of current sample */

void circ_update(int xnew) {
   /* compute the new head value with wraparound; the pos pointer moves from 0 to CMAX-1 */
   pos = ((pos == CMAX-1) ? 0 : (pos+1));
   /* insert the new value at the new head */
   circ[pos] = xnew;
   }
```

Circular buffer in C, cont'd.

```
void circ_init() {
   int i;
   for (i=0; i<CMAX; i++) /* set values to 0 */
   circ[i] = 0; pos=CMAX-1; /* start at tail so first element will be at 0 */
   }
int circ_get(int i) {
   int ii;
   /* compute the buffer position */
   ii = (pos - i) % CMAX;
   return circ[ii]; /* return the value */
   }
```

FIR filter



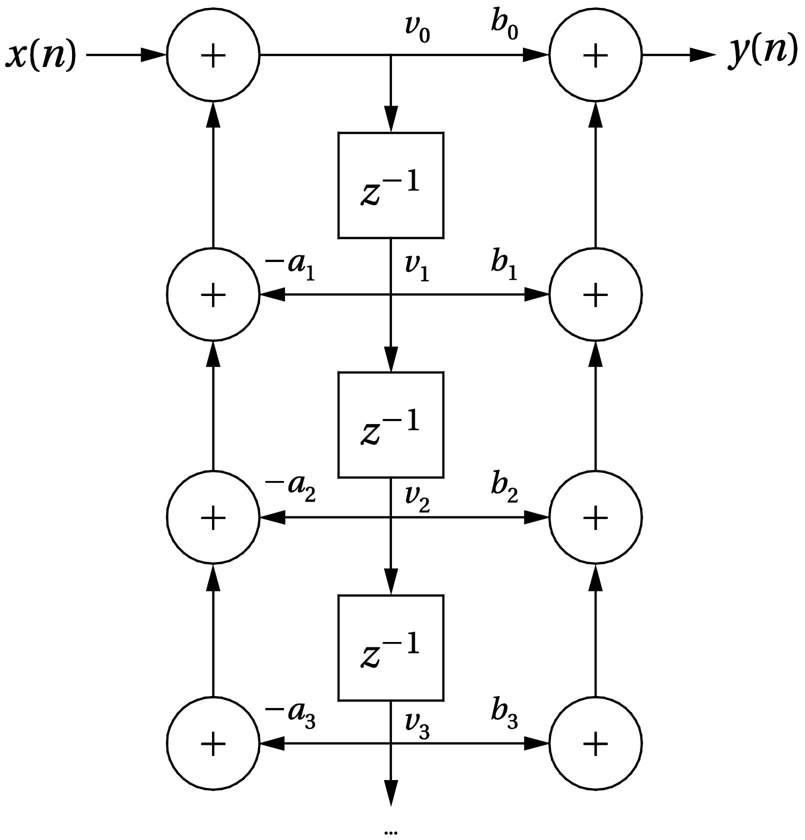
FIR filter update function

```
void circ_update(int xnew) {
  /* add the new sample and push off the oldest one */
  /* compute the new head value with wraparound; the      pos pointer moves from
  CMAX-1 down to 0 */
  pos = ((pos == 0) ? CMAX-1 : (pos-1));
  /* insert the new value at the new head */
  circ[pos] = xnew;
  }
```

FIR filter using circular buffer

```
int fir(int xnew) {
  /* given a new sample value, update the queue and         compute the filter
  output */
  int i;
  int result; /* holds the filter output */
  circ_update(xnew); /* put the new value in */
  for (i=0, result=0; i<CMAX; i++)
          result += b[i] * circ_get(i);
  return result;
  }
```

IIR direct form type II filter



IIR filter in C

```
int iir2(int xnew) {
  int i, aside, bside, result;
  for (i=0, aside=0; i<ZMAX; i++)
        aside += -a[i+1] * circ_get(i);
  for (i=0, bside=0; i<ZMAX; i++)
        bside += b[i+1] * circ_get(i);
  result = b[0] * (xnew + aside) + bside;
  circ_update(xnew); /* put the new value in */
  return result;
  }
```

Array-based queue in C

```
#define Q_SIZE 5 /* your queue size may vary */
#define Q_MAX (Q_SIZE-1) /* this is the maximum index value into the array */
int q[Q_SIZE]; /* the array for our queue */
int head, tail; /* indexes for the current queue head and tail */
void queue_init() {
/* initialize the queue data structure */
head = 0;
tail = 0;
}
```

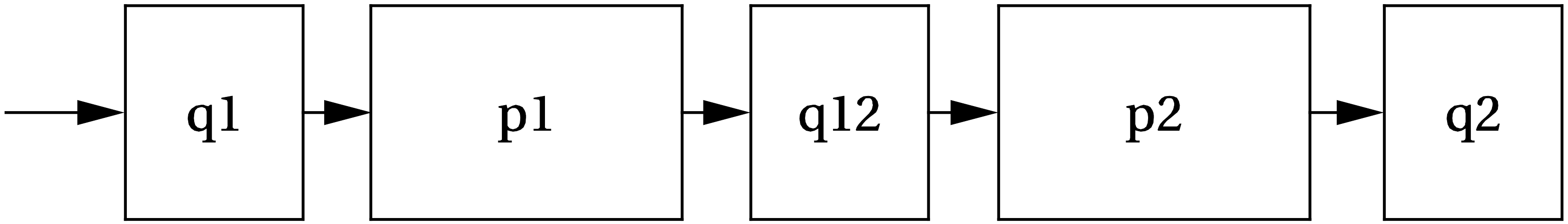
Array based queue, cont'd.

```
void enqueue(int val) {
   if (((tail+1) % Q_SIZE) == head) error("enqueue onto full queue",tail);
   q[tail] = val;
   /* update the tail */
   if (tail == Q_MAX)
         tail = 0;
   else
   tail++;
   }
```

Array based queue, cont'd.

```
int dequeue() {
   int returnval;
   if (head == tail) error("dequeue from empty queue",head);
   returnval = q[head];
   if (head == Q_MAX)
            head = 0;
   else
            head++;
   return  returnval;
}
```

Producer-consumer system



- Queues allow varying input and output rates.

Models of programs

- Source code is not a good representation for programs:
  - clumsy;
  - leaves much information implicit.
- Compilers derive intermediate representations to manipulate and optiize the program.

Data flow graph

- DFG: data flow graph.
- Does not represent control.
- Models basic block: code with no entry or exit.
- Describes the minimal ordering requirements on operations.

Single assignment form

```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

original basic block

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

single assignment form

Data flow graph

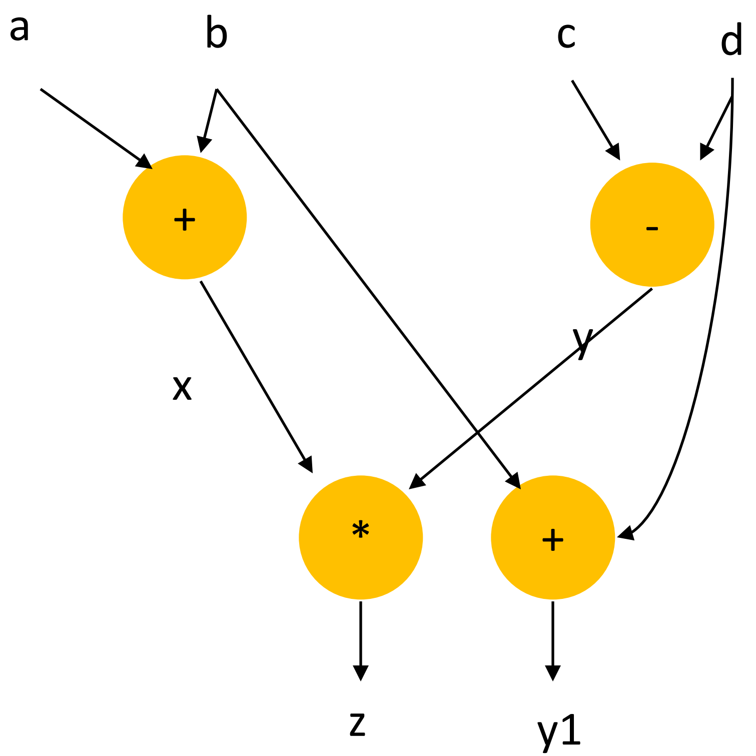
x = a + b;

y = c - d;

z = x * y;

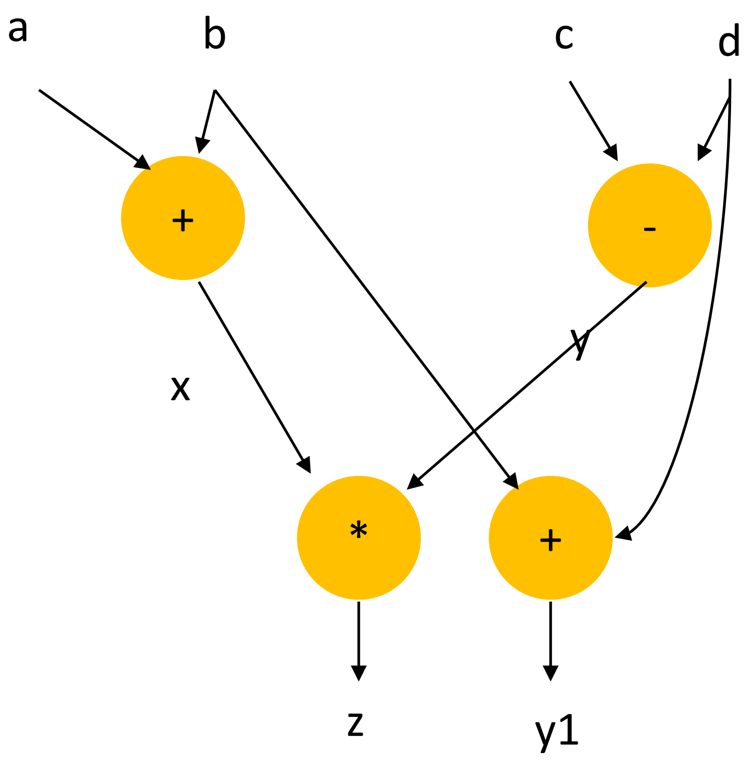
y1 = b + d;

single assignment form



DFG

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

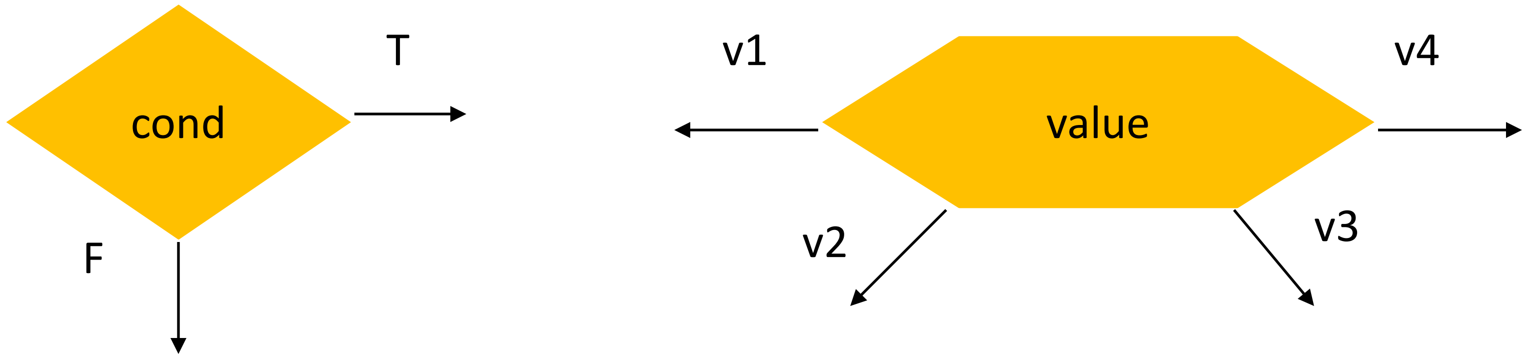
Data flow node

Encapsulates a data flow graph:

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

Write operations in basic block form for simplicity.

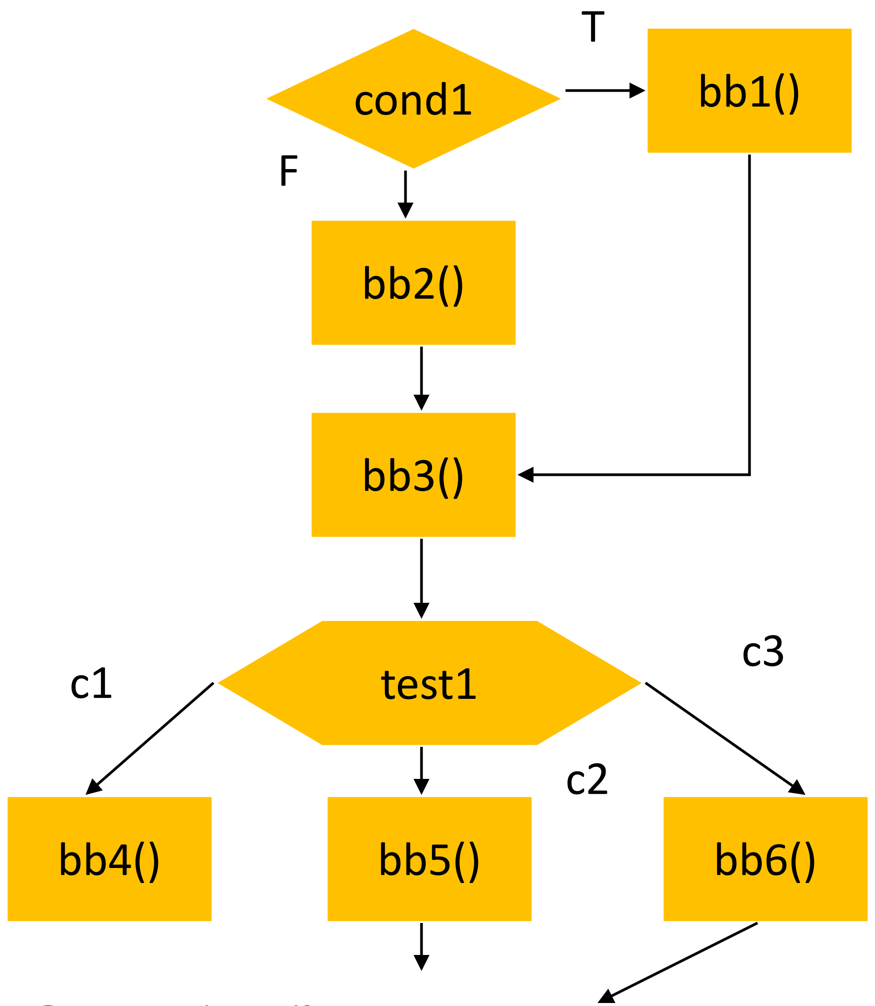
Control



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();

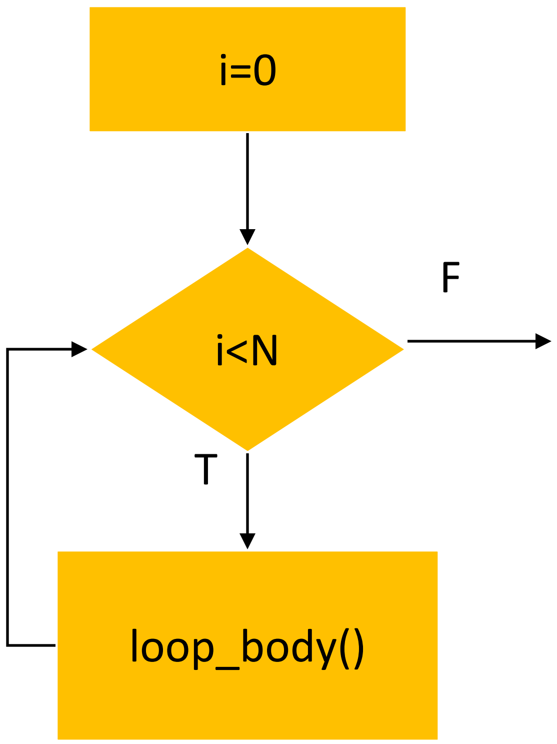
for loop

i=0;

while (i<N) {

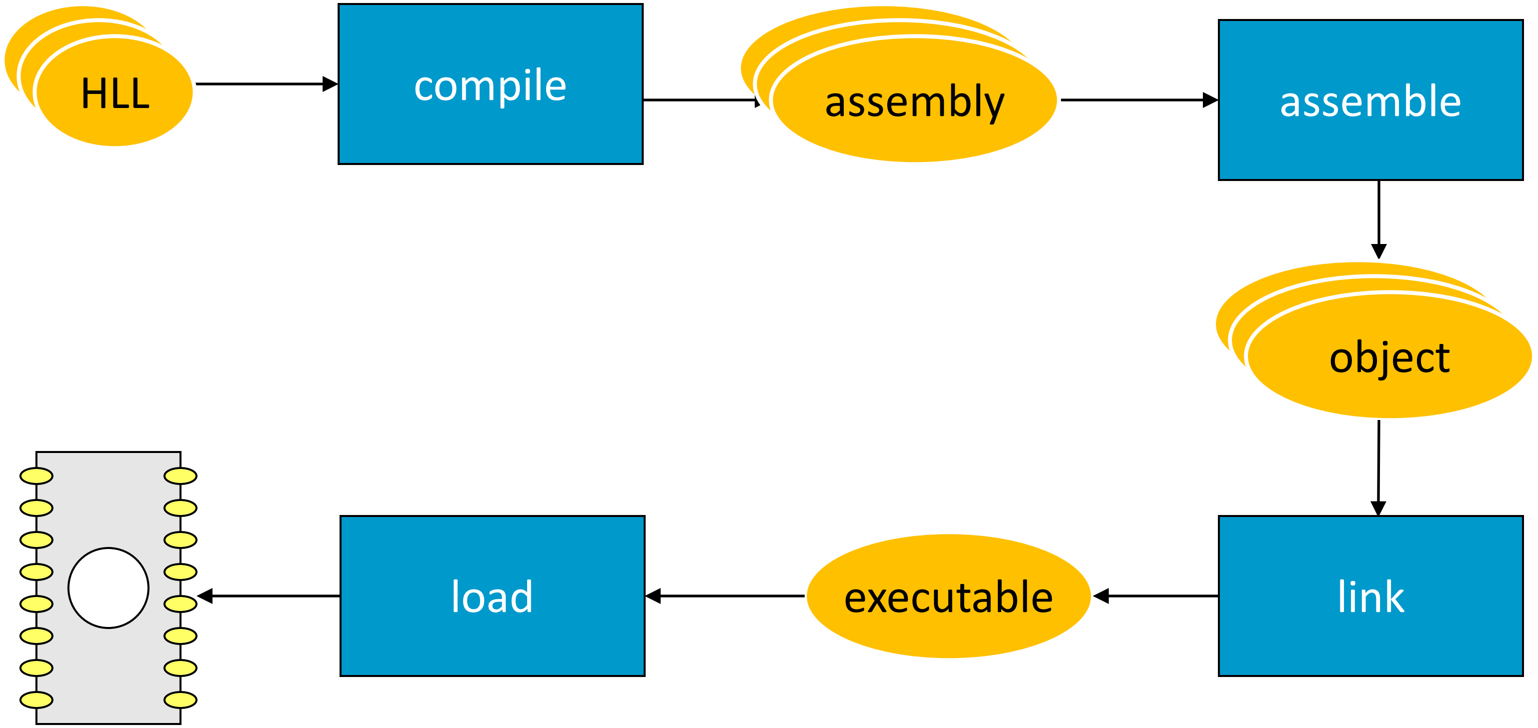
  loop_body(); i++; }

equivalent



Assembly and linking

- Last steps in compilation:



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	ADD r3,r4,r5
	CMP r0,r3
yy	SUB r5,r6,r7

assembly code

xx	0x8
yy	0x10

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

r2

PLC=0x8

xx r5

PLC=0x9

PLC=0x10

yy r7

xx	0x8
yy	0x10

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.

```
       ORG 100
label1 ADR r4,c
       LDR r0,[r4]
label2 ADR r4,d
       LDR r1,[r4]
label3 SUB r0,r0,r1
```

```
                   ORG 100
        label1     ADR r4,c
                   LDR r0,[r4]
        label2     ADR r4,d
                   LDR r1,[r4]
PLC=116 → label3   SUB r0,r0,r1
```

Code

label1	100
label2	108
label3	116

Symbol table

```
        ADD r0,r1,r2
FOO     EQU 5
BAZ     SUB r3,r4,#FOO
```

Linking

- Combines several object modules into a single executable module.
- Jobs:
  - put modules in order;
  - resolve labels across modules.

Externals and entry points

entry point

xxx	ADD r1,r2,r3	a
	B a	
yyy	%1	

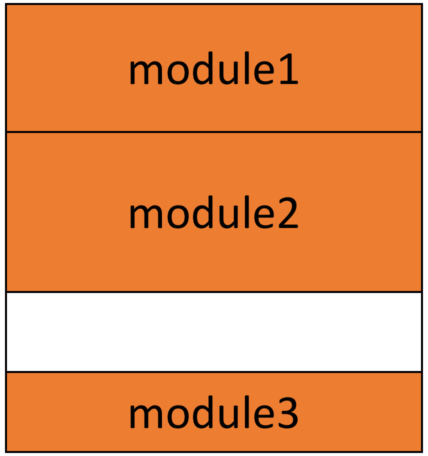
ADR r4,yyy

ADD r3,r4,r5

external reference

Module ordering

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



Dynamic linking

- Some operating systems link modules dynamically at run time:
  - shares one copy of library among all executing programs;
  - allows programs to be updated with new versions of libraries.

Reentrancy

- Interrupting program with another call to the function does not change results.
  - Changing global variables compromises reentrancy.
- Recursive code:

```
int foo = 1;
int task1() {
   foo = foo + 1;
   return foo;
   }
```

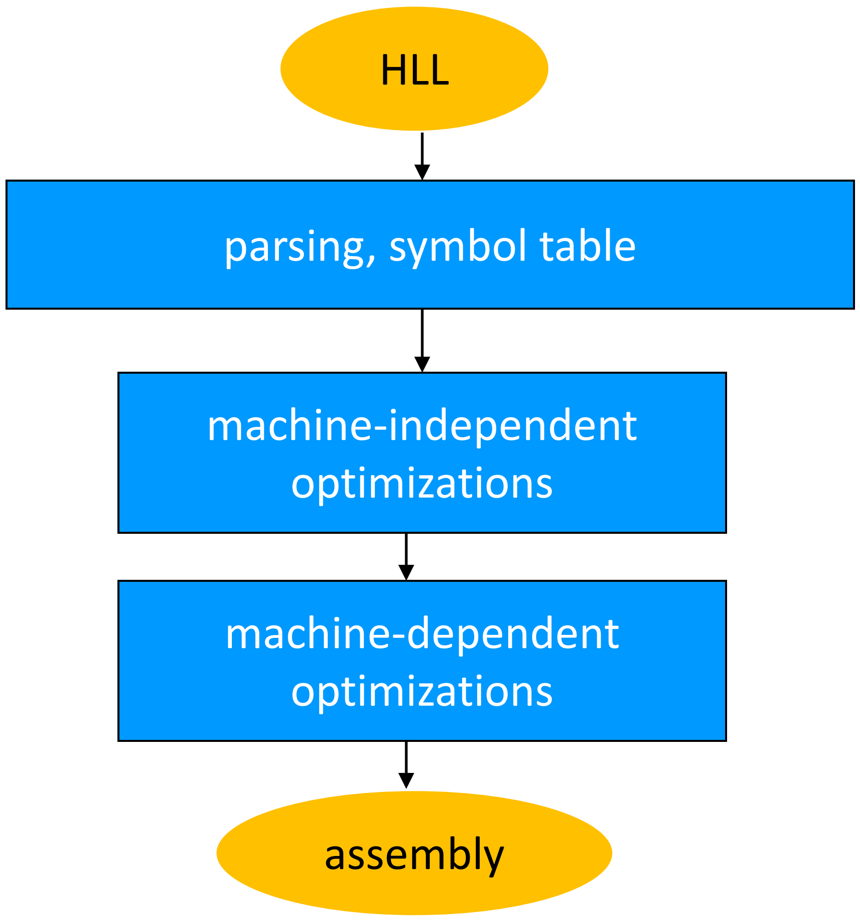
Program design and analysis

- 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



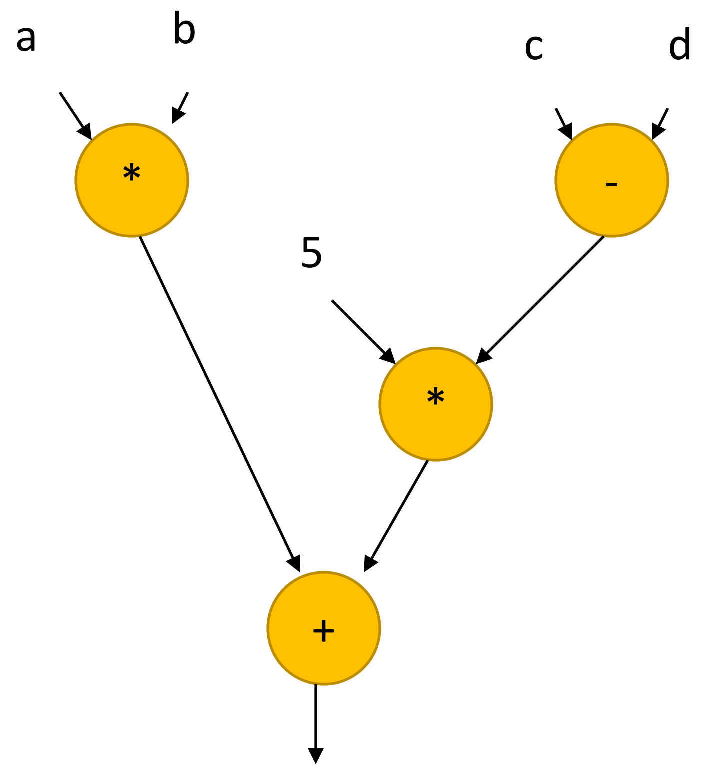
Statement translation and optimization

- 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

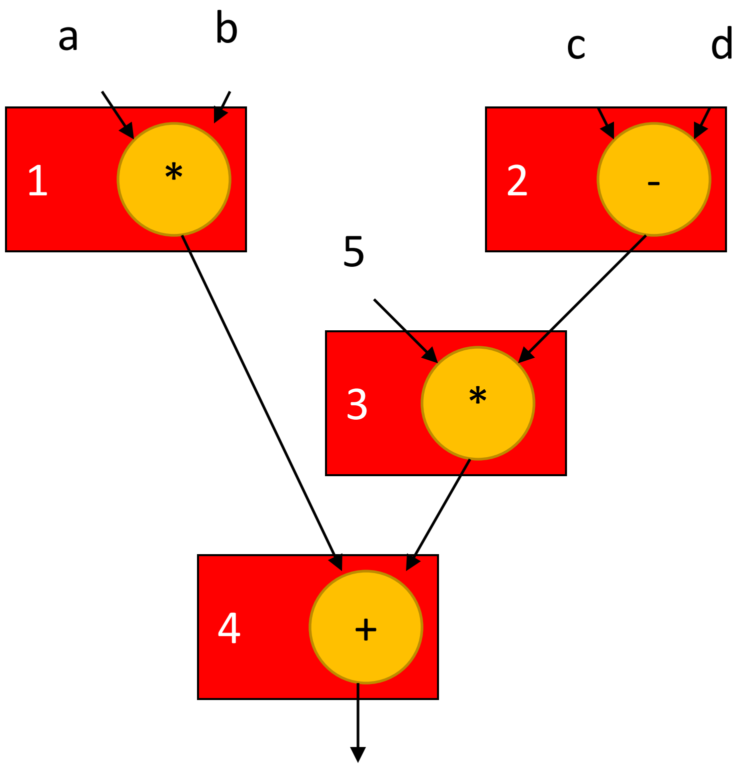
a*b + 5*(c-d)

expression



DFG

Arithmetic expressions, cont'd.



DFG

```
ADR r4,a
MOV r1,[r4]
ADR r4,b
MOV r2,[r4]
MUL r3,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
```

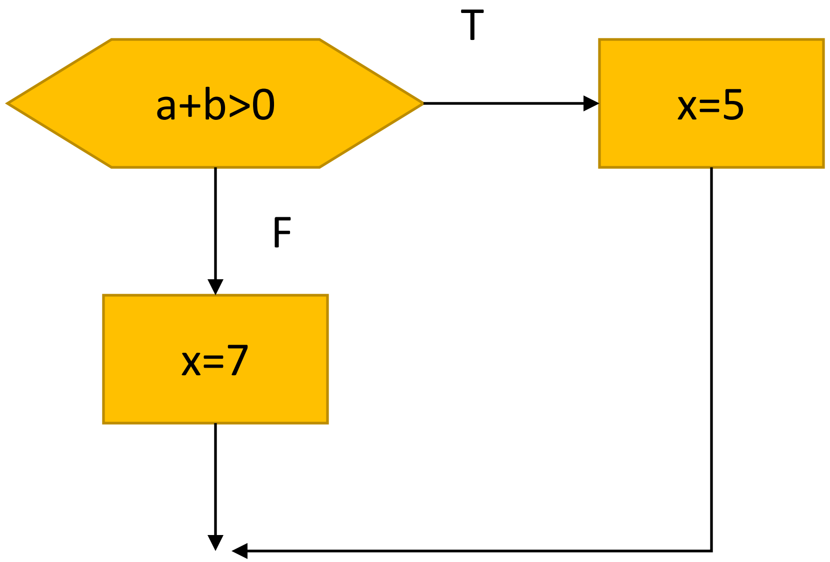
code

Compiled code for arithmetic expressions

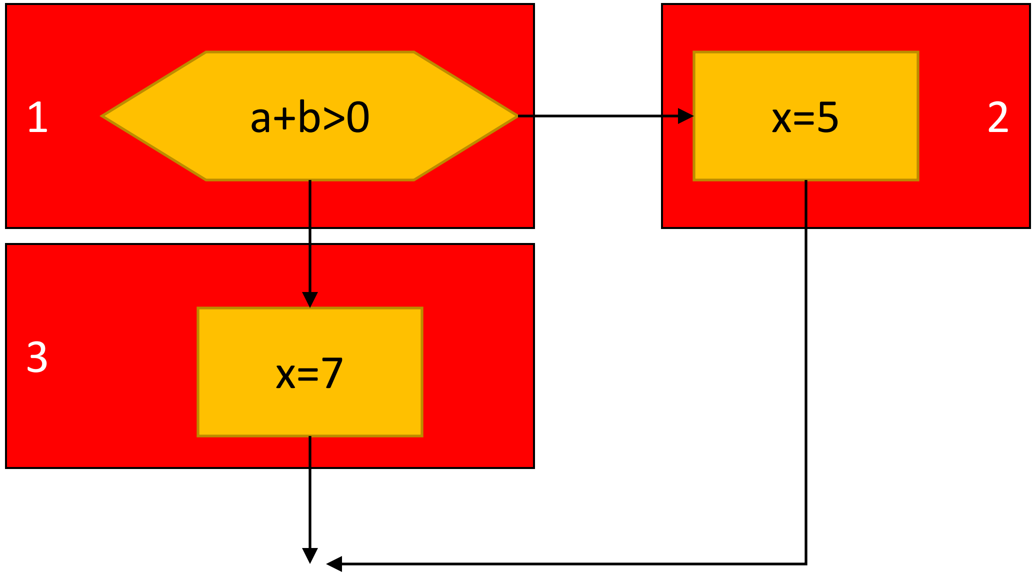
```
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
mul r1, r3, r2 ; multiply
ldr r2, [fp, #-24]
ldr r3, [fp, #-28]
rsb r2, r3, r2 ; subtract
mov r3, r2
mov r3, r3, asl #2
add r3, r3, r2 ; add
add r3, r1, r3 ; add
str r3, [fp, #-32] ; assign
```

Control code generation

```
if (a+b > 0)
  x = 5;
else
  x = 7;
```



Control code generation, cont'd.




```
ADR r5,a
LDR r1,[r5]
ADR r5,b
LDR r2,[r5]
ADD r3,r1,r2
BLE label3
   LDR r3,#5
   ADR r5,x
   STR r3,[r5]
   B stmtent
label3 LDR r3,#7
   ADR r5,x
   STR r3,[r5]
stmtent ...
```

Compiled code for control

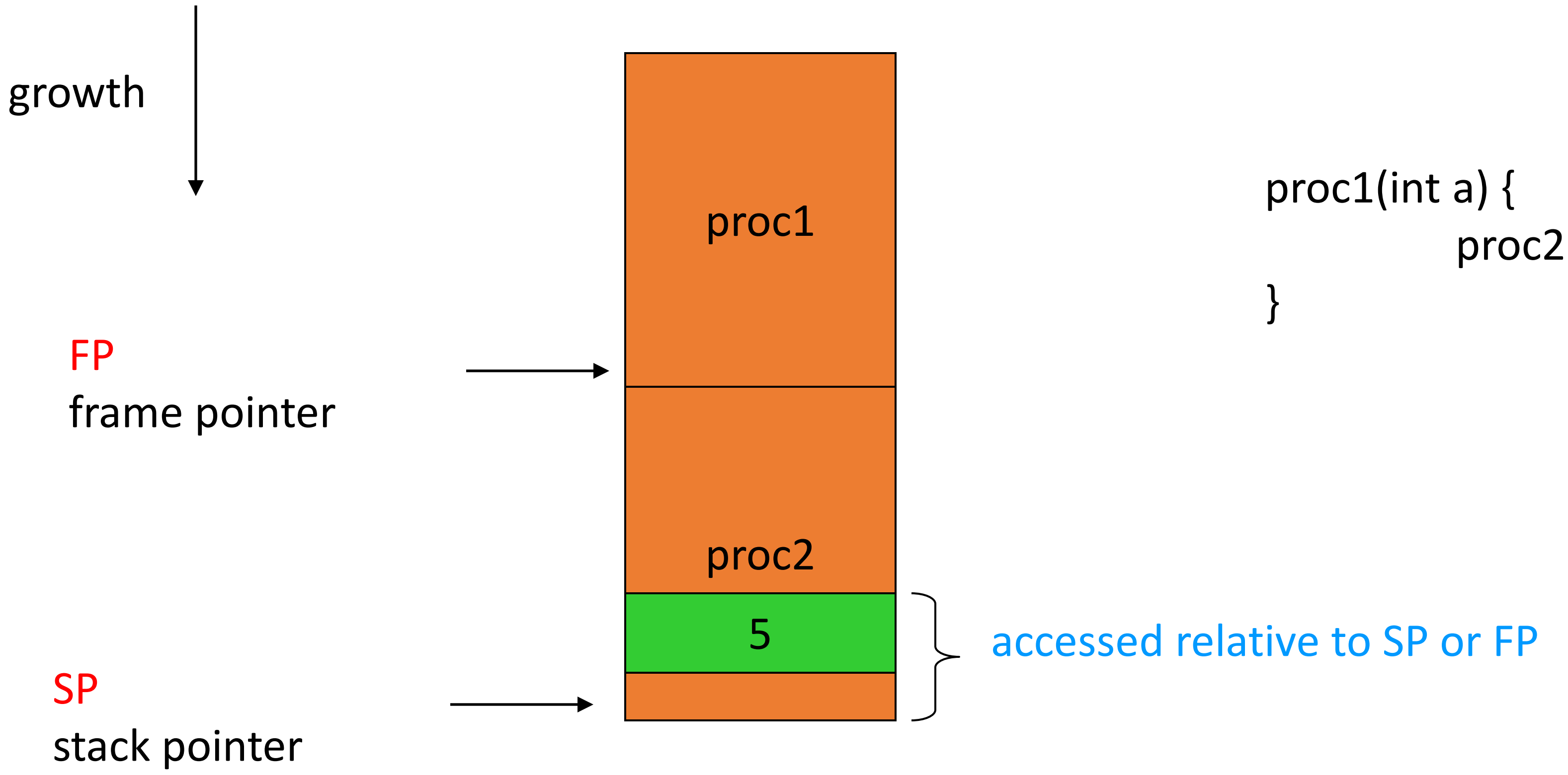
```
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
add r3, r2, r3
cmp r3, #0 ; test the branch condition
ble .L3 ; branch to false block if <=
```

```
mov r3, #5 ; true block
str r3, [fp, #-32]
b .L4 ; go to end of if statement
.L3: ; false block
mov r3, #7
str r3, [fp, #-32]
.L4:
```

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.
- Local variables are stored in the stack.

Procedure stacks




```
proc1(int a) {
        proc2(5);
}
```

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.

```
int p1(int a, int b, int c, int d, int e) {
    return a + e;
    }
```

```
mov    ip, sp                ; procedure entry
stmfd  sp!, {fp, ip, lr, pc}
sub    fp, ip, #4
sub    sp, sp, #16
str    r0, [fp, #-16]        ; put first four args on stack
str    r1, [fp, #-20]
str    r2, [fp, #-24]
str    r3, [fp, #-28]
ldr    r2, [fp, #-16]        ; load a
ldr    r3, [fp, #4]          ; load e
add    r3, r2, r3            ; compute a + e
mov    r0, r3                ; put the result into r0 for return
ldmea  fp, {fp, sp, pc}      ; return
```

Compiled procedure call code

```
ldr r3, [fp, #-32] ; get e
str r3, [sp, #0] ; put into p1()’s stack frame
ldr r0, [fp, #-16] ; put a into r0
ldr r1, [fp, #-20] ; put b into r1
ldr r2, [fp, #-24] ; put c into r2
ldr r3, [fp, #-28] ; put d into r3
bl p1 ; call p1()
mov r3, r0 ; move return value into r3
str r3, [fp, #-36] ; store into y in stack frame
```

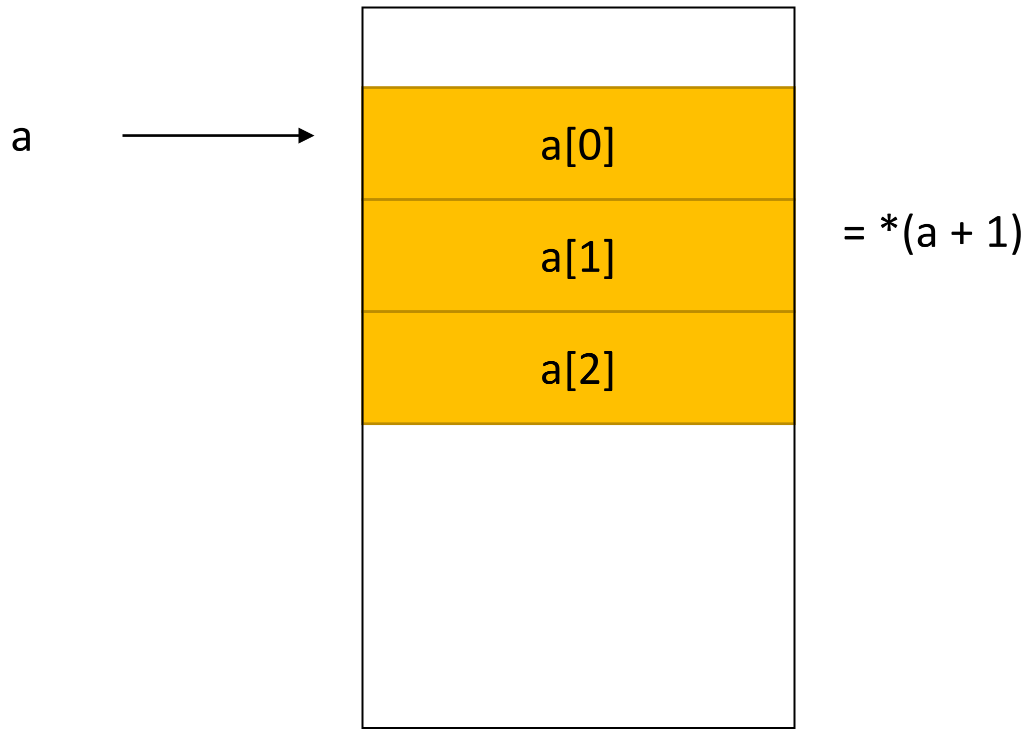
```
y = p1(a,b,c,d,x);
```

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.

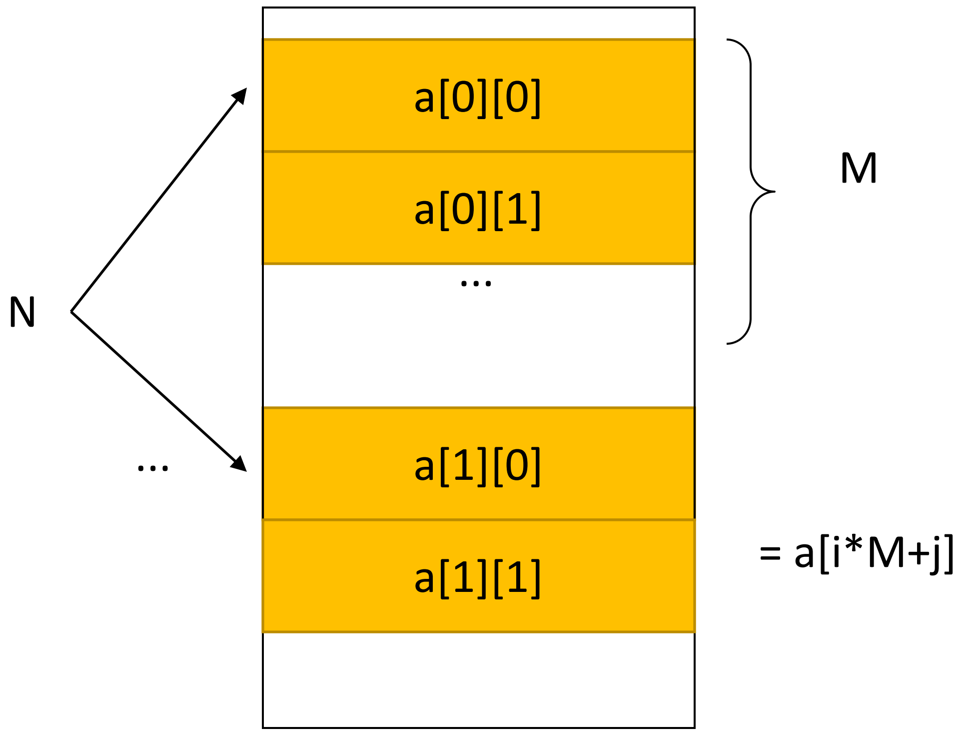
One-dimensional arrays

- C array name points to 0th element:



Two-dimensional arrays

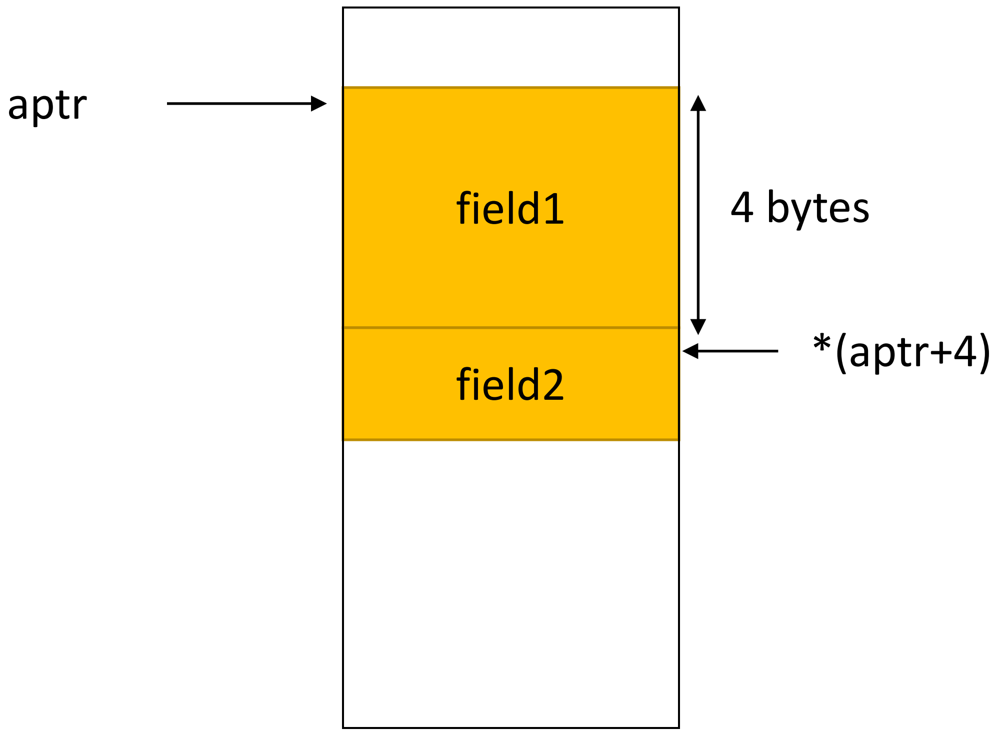
- Row-major layout:



Structures

- Fields within structures are static offsets:

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

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



Expression simplification

- Constant folding:
  - 8+1 = 9
- Algebraic:
  - a*b + a*c = a*(b+c)
- Strength reduction:
  - a*2 = a<<1

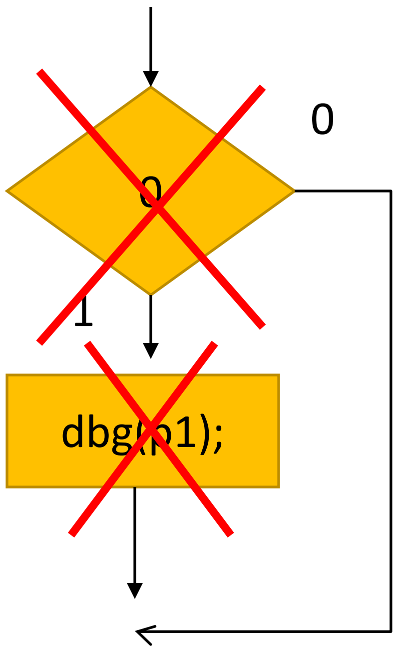
Dead code elimination

- Dead code:

#define DEBUG 0

if (DEBUG) dbg(p1);

- Can be eliminated by analysis of control flow, constant folding.



Procedure inlining

- Eliminates procedure linkage overhead:

```
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;
  - improve memory system performance.

Loop unrolling

- Reduces loop overhead, enables some other optimizations.

```
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];
   }
```

- Distribution breaks one loop into two.
- Changes optimizations within loop body.

Register allocation

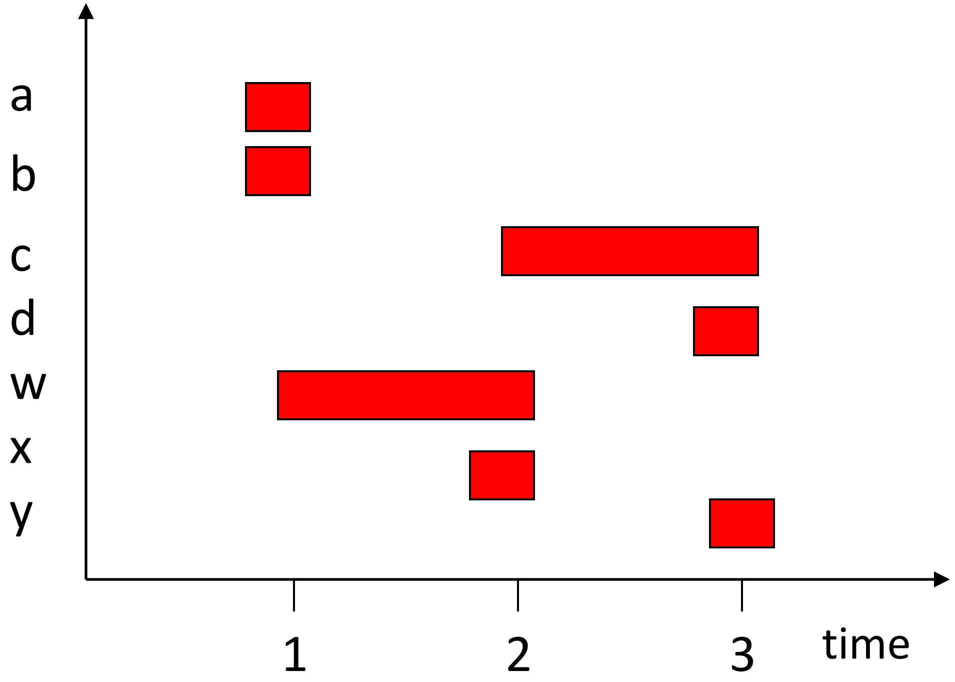
- Goals:
  - choose register to hold each variable;
  - determine lifespan of varible in the register.
- Basic case: within basic block.

Register lifetime graph

w = a + b; t=1

x = c + w; t=2

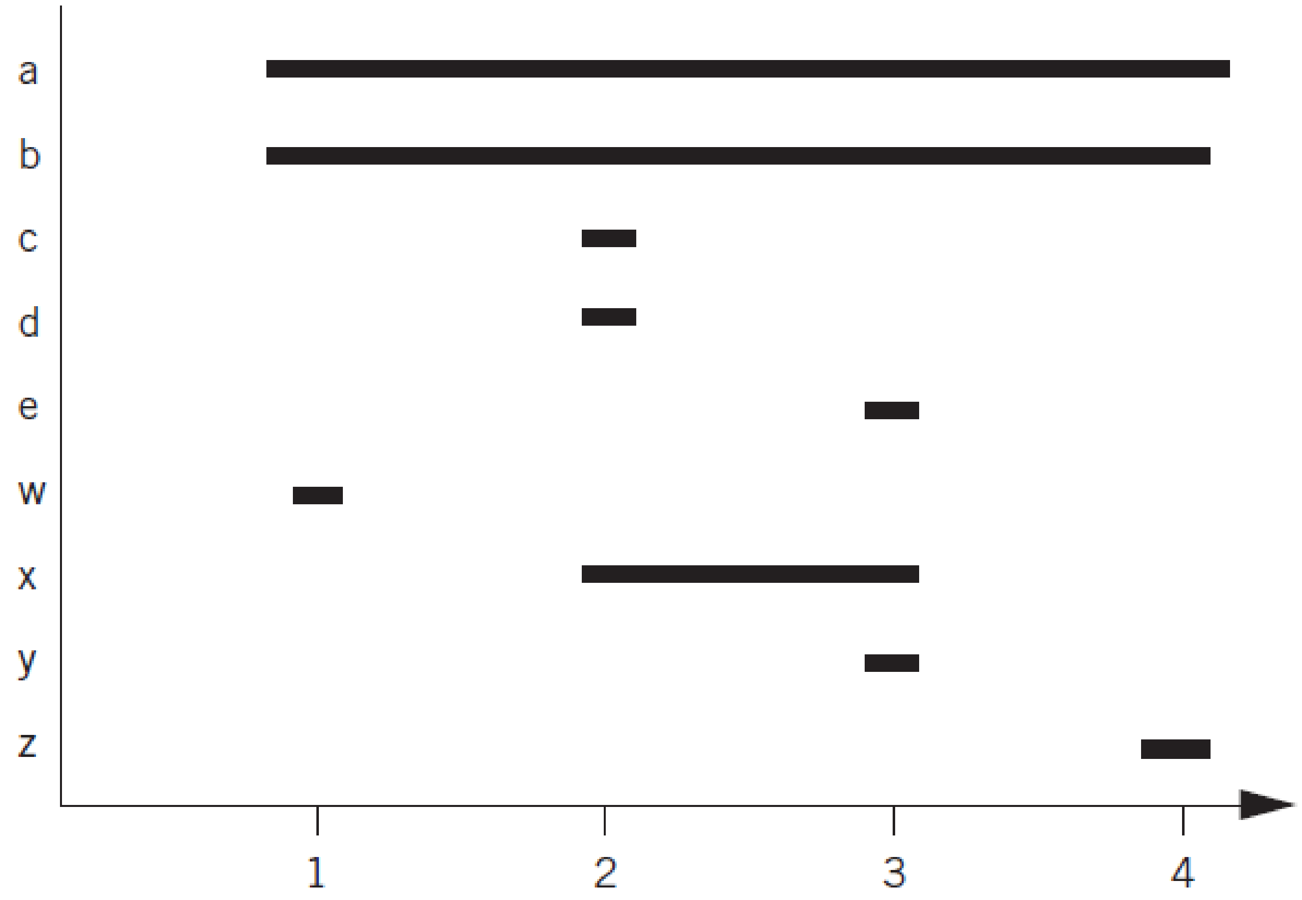
y = c + d; t=3



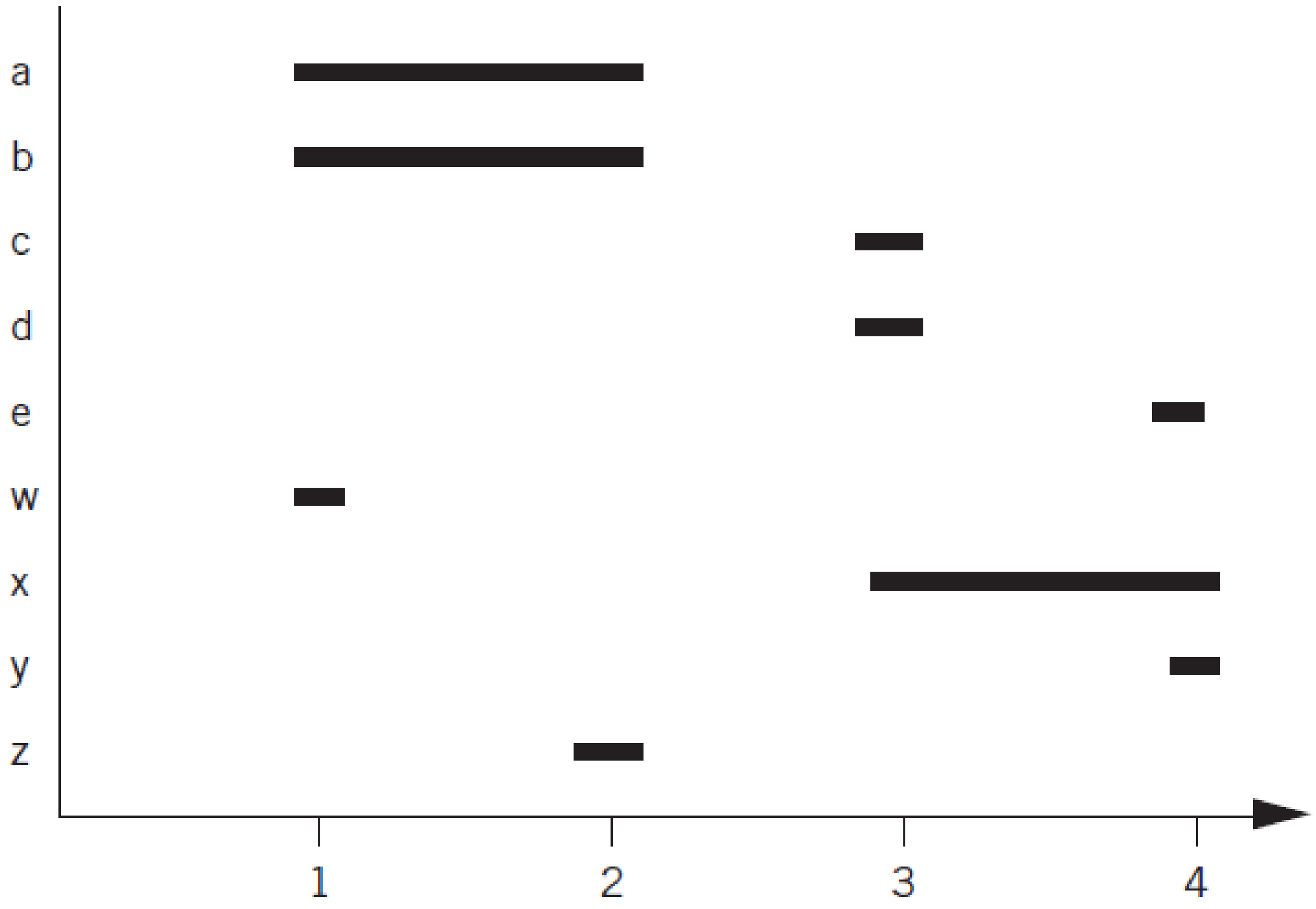
Instruction scheduling

- 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.

```
w = a + b; /* statement 1 */
x = c + d; /* statement 2 */
y = x + e; /* statement 3 */
z = a - b; /* statement 4 */
```



```
w = a + b;  /*statement 1 */
z = a – b;  /* statement 29 */
x = c + d;  /*statement 39 */
y = x + e;  /*statement 49 */
```



Reservation table

- A reservation table relates instructions/time to CPU resources.

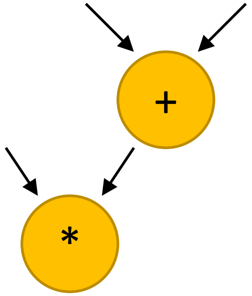
Time/instr	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

Software pipelining

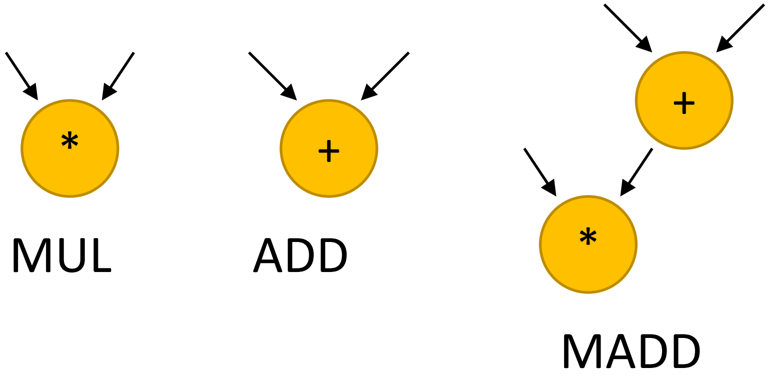
- Schedules instructions across loop iterations.
- Reduces instruction latency in iteration i by inserting instructions from iteration i+1.

Instruction selection

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



expression



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

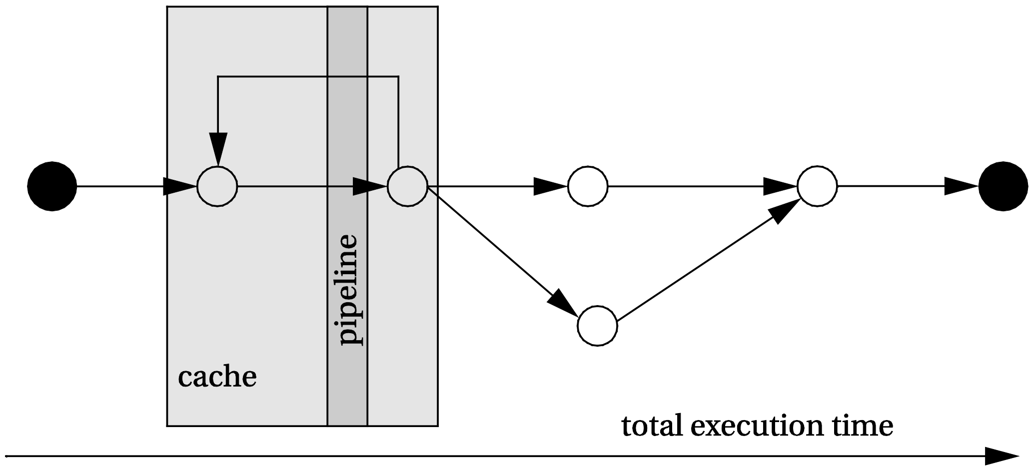
- Interpreter: translates and executes program statements on-the-fly.
- JIT compiler: compiles small sections of code into instructions during program execution.
  - Eliminates some translation overhead.
  - Often requires more memory.

Program design and analysis

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

Program-level 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.

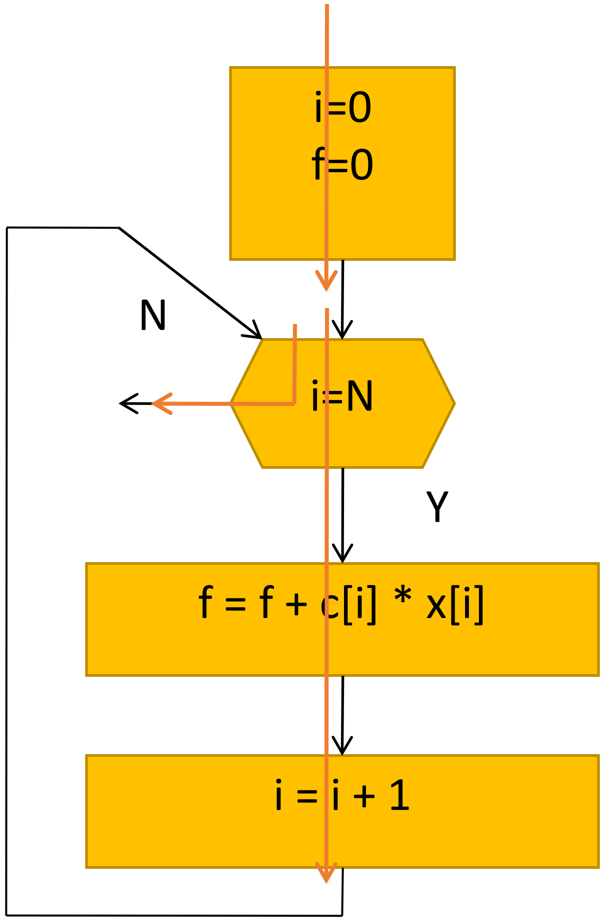
Data-dependent paths in an if statement

```
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

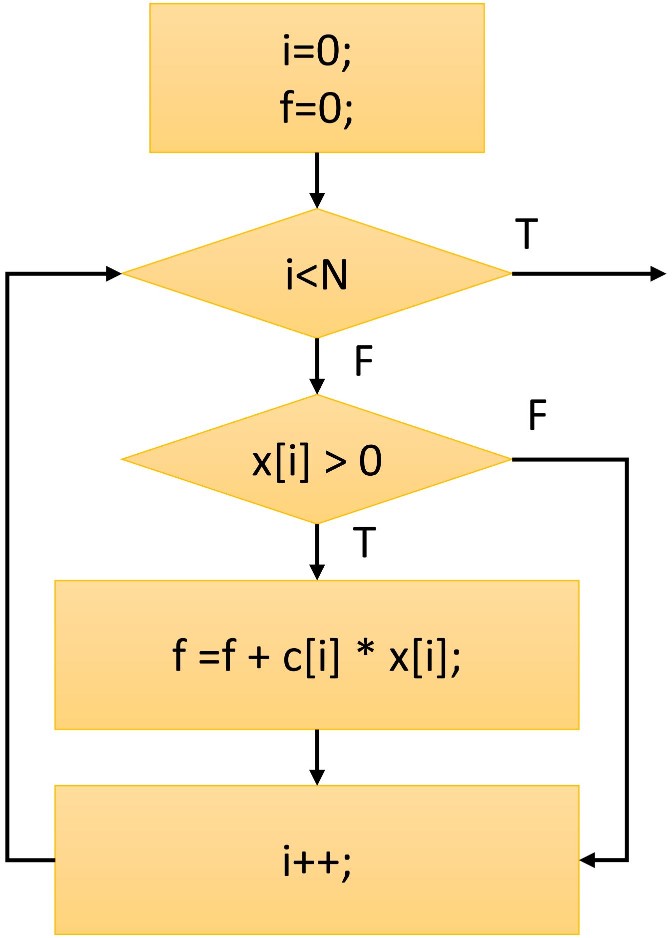
Paths in a loop

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



Paths in a loop with conditions

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



executed 1 time

executed N+1 times

executed N times

executed $[0, N]$ times

executed N times

Instruction timing

- Not all instructions take the same amount of time.
  - Multi-cycle instructions.
  - Fetches.
- Execution times of instructions are not independent.
  - Pipeline interlocks.
  - Cache effects.
- Execution times may vary with operand value.
  - Floating-point operations.
  - Some multi-cycle integer operations.

Caching effects

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

$$t_{loop} = 2N + \frac{N}{L} t_{miss} + N\left(1 - \frac{1}{L}\right) t_{hit}$$

line 0	Word 0	Word 1	Word 2	Word 3
line 1	Word 4	Word 5	Word 6	Word 7

- First access to a cache line causes a miss and prefetch.
- Later accesses to that line result in cache hits.

Mesaurement-driven performance analysis

- 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.

Feeding the program

- Need to know the desired input values.
- May need to write software scaffolding to generate the input values.
- Software scaffolding may also need to examine outputs to generate feedback-driven inputs.

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.

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 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];
}
```

COUNT	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

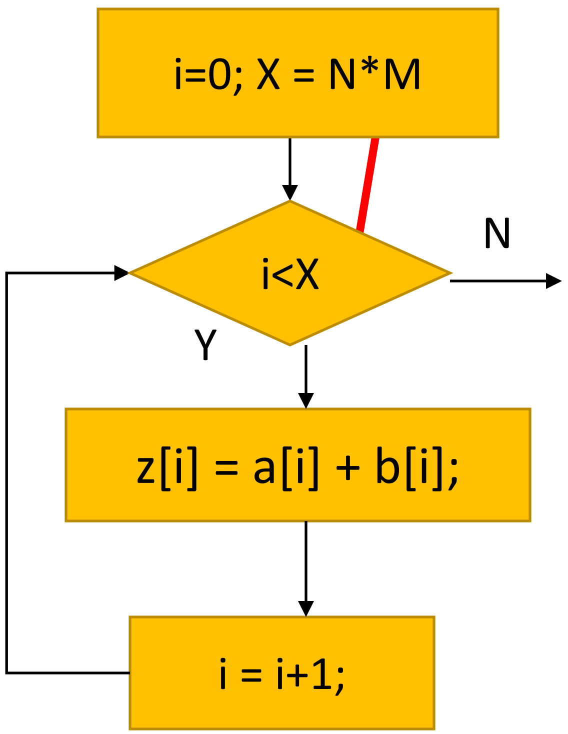
- Best results come from analyzing optimized instructions, not high-level language code:
  - non-obvious translations of HLL statements into instructions;
  - code may move;
  - cache effects are hard to predict.

Loop optimizations

- Loops are good targets for optimization.
- Basic loop optimizations:
  - code motion;
  - induction-variable elimination;
  - strength reduction (x*2 -> x<<1).

Code motion

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



Induction variable elimination

- Induction variable: loop index.
- Consider loop:

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

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

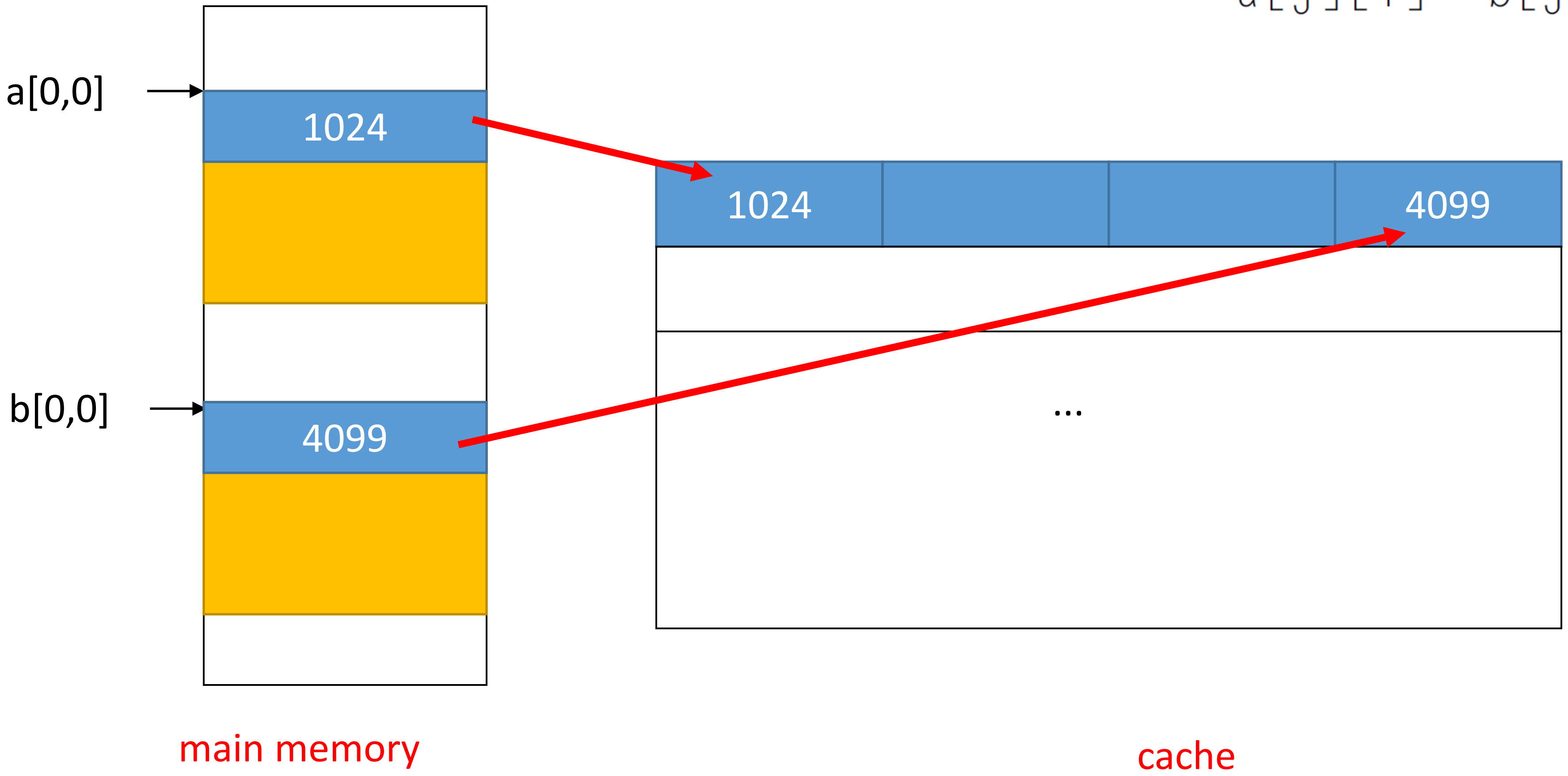
```
for (i = 0; i <N; i++)
      for (j = 0; j <M; j++) {
            zbinduct = i*M + j;
            *(zptr + zbinduct) = *(bptr + zbinduct);

zbinduct = 0;
for (i = 0; i <N; i++) {
      for (j = 0; j <M; j++) {
            *(zptr + zbinduct) = *(bptr + zbinduct);
            zbinduct++;
            }
      }
```

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

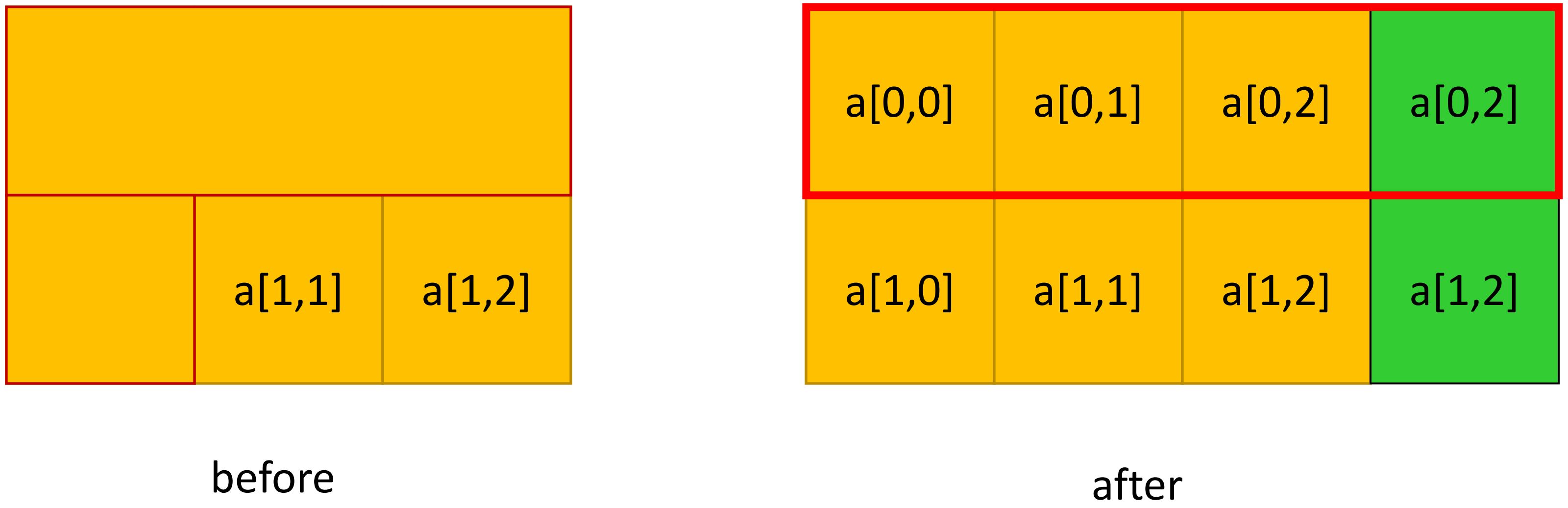


Array conflicts, cont'd.

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

Array padding

- Add array elements to change mapping into cache:



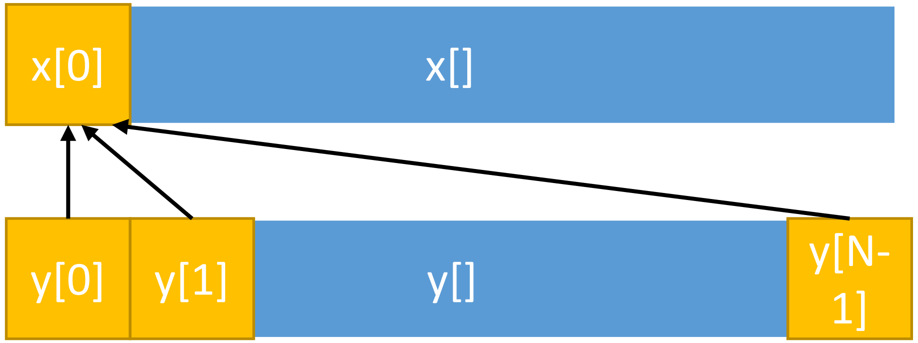
Loop tiling

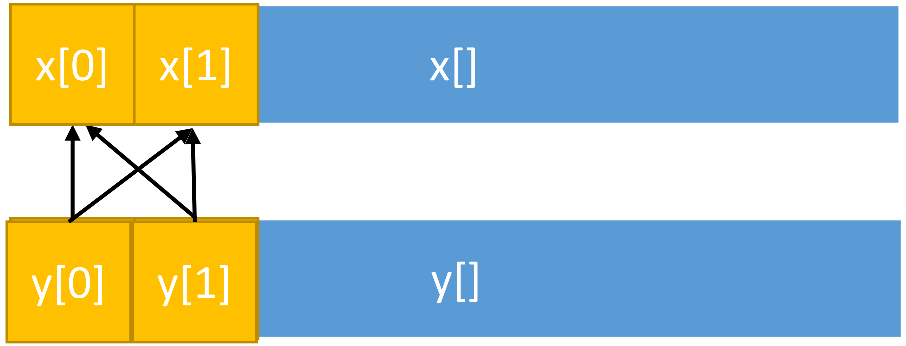
- Breaks one loop into a nest of loops.
- Changes order of accesses within array.
  - Changes cache behavior.

Loop tiling example

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        z[i][j] = x[i] * y[j];
    }
}
```

```
for (j=0; j < N; j += TILE) {
    for (i=0; i<N; i++) {
        for (jj=0; j<TILE; jj++) {
        z[i][j + jj] = x[i] * y[j + jj];
        }
    }
}
```





Performance optimization hints

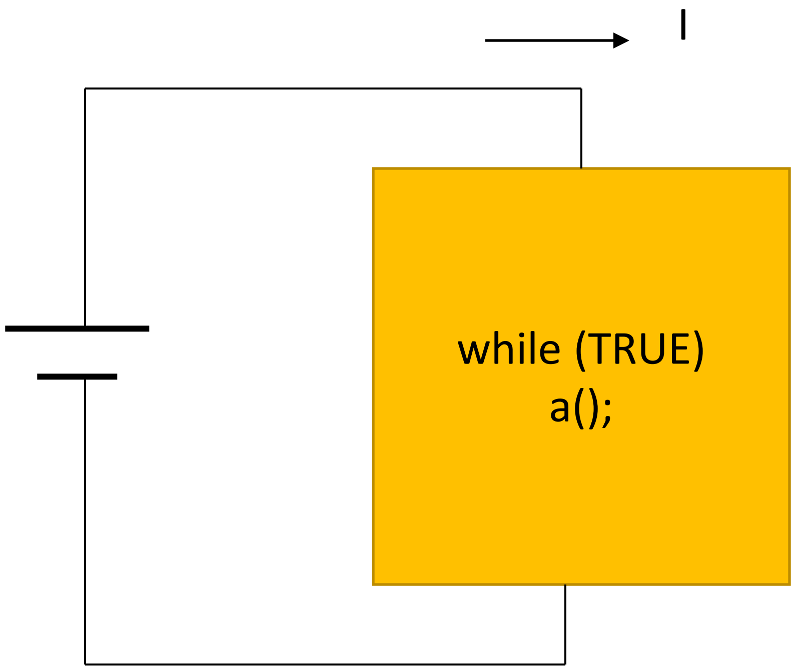
- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
    - instruction conflicts can be handled by rewriting code, rescheudling;
    - conflicting scalar data can easily be moved;
    - conflicting array data can be moved, padded.

Energy/power optimization

- Energy: ability to do work.
  - Most important in battery-powered systems.
- Power: energy per unit time.
  - Important even in wall-plug systems---power becomes heat.

Measuring energy consumption

- Execute a small loop, measure current:



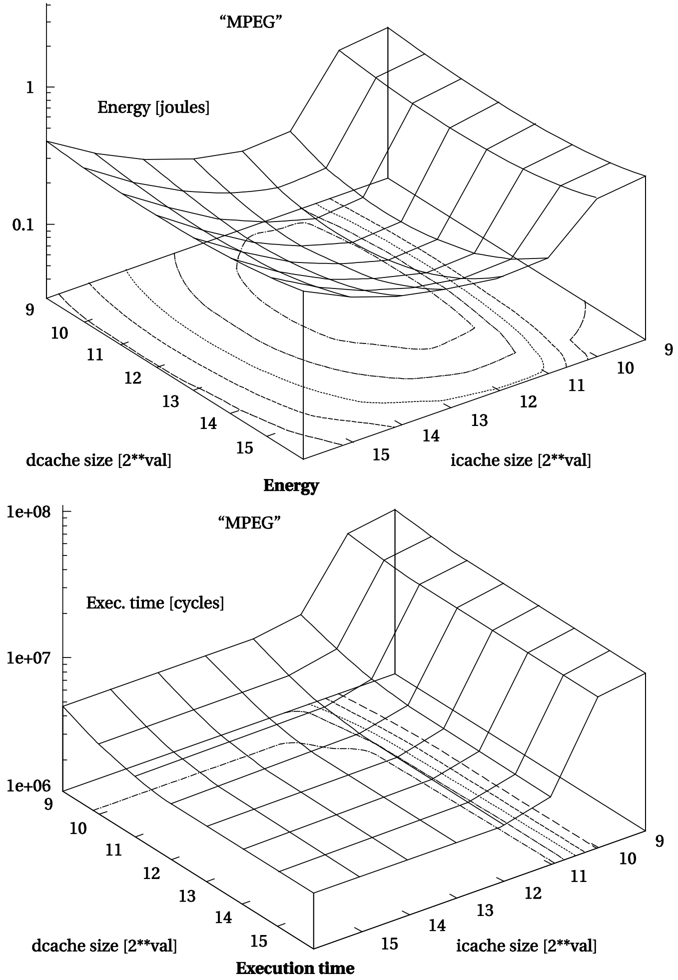
Sources of energy consumption

- Relative energy per operation (Catthoor et al):
  - memory transfer: 33
  - external I/O: 10
  - SRAM write: 9
  - SRAM read: 4.4
  - multiply: 3.6
  - add: 1

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.

Cache sweet spot



[Li98] © 1998 IEEE

Optimizing for energy

- First-order optimization:
  - high performance = low energy.
- Not many instructions trade speed for energy.

Optimizing for energy, cont'd.

- 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.

Efficient loops

- General rules:
  - Don’t use function calls.
  - Keep loop body small to enable local repeat (only forward branches).
  - Use unsigned integer for loop counter.
  - Use <= to test loop counter.
  - Make use of compiler---global optimization, software pipelining.

Single-instruction repeat loop example

```
STM #4000h,AR2
  ; load pointer to source
STM #100h,AR3
  ; load pointer to destination
RPT #(1024-1)
MVDD *AR2+,*AR3+
  ; move
```

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 and testing

- But does it work?
- Concentrate here on functional verification.
- Major testing strategies:
  - Black box doesn’t look at the source code.
  - Clear box (white box) does look at the source code.

Clear-box testing

- 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.

Controlling and observing programs

```
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 desired N values.
  - Other code governs how we access the buffer.
- Observability:
  - Want to examine firout before limit testing.

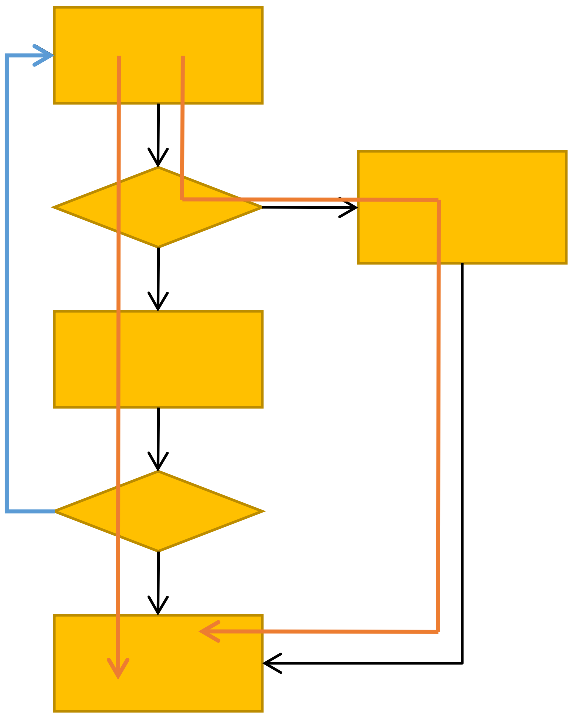
Execution paths and testing

- 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

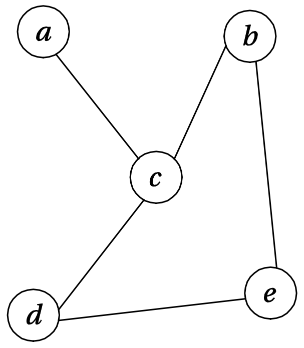
- Possible criteria:
  - Execute every statement at least once.
  - Execute every branch direction at least once.
- Equivalent for structured programs.
- Not true for gotos.

not covered



Basis paths

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



Graph

$$
\begin{array}{c c}
 & \begin{matrix} a & b & c & d & e \end{matrix} \\
\begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} &
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

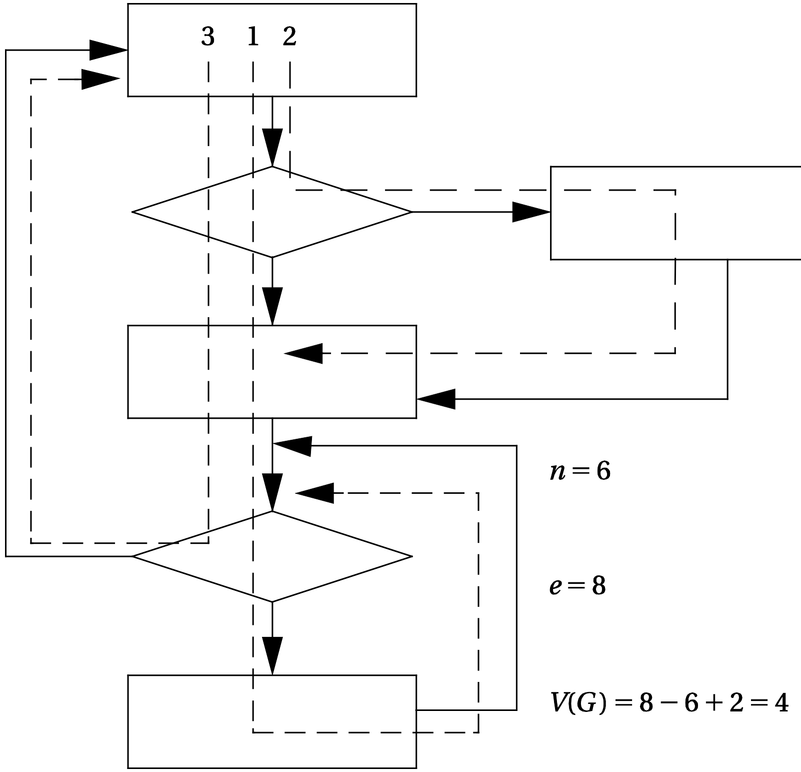
Incidence matrix

$$
\begin{array}{c c}
\begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} &
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\end{array}
$$

Basis set

Cyclomatic complexity

- Cyclomatic complexity is a bound on the size of basis sets:
  - e = # edges
  - n = # nodes
  - p = number of graph components
  - M = e – n + 2p.



Branch testing

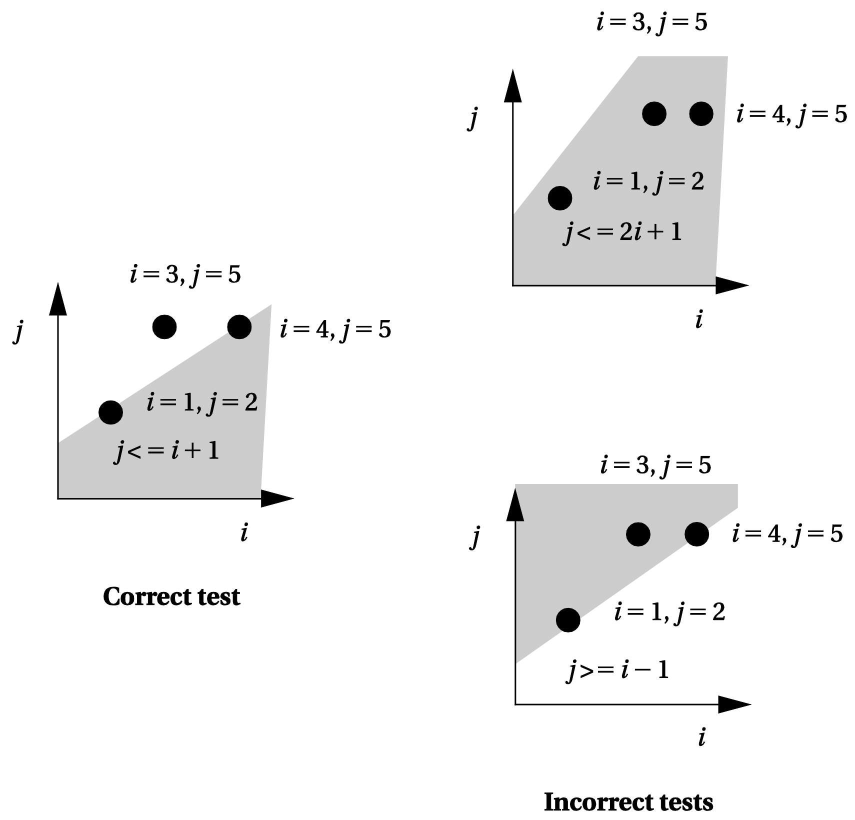
- Heuristic for testing branches.
  - Exercise true and false branches of conditional.
  - Exercise every simple condition at least once.

Branch testing example

- Correct:
  - if (a || (b >= c)) { printf(“OK\n”); }
- Incorrect:
  - if (a && (b >= c)) { printf(“OK\n”); }
- Test:
  - a = F
  - (b >=c) = T
- Example:
  - Correct: [0 || (3 >= 2)] = T
  - Incorrect: [0 && (3 >= 2)] = F

Domain testing

- Heuristic test for linear inequalities.
- Test on each side + boundary of inequality.



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.

```
a = mypointer;
if (c > 5){
    while (a->field1 != val1)
        a = a->next;
}
if (a->field2 == val2)
    someproc(a,b);
```

Loop testing

- Loops need specialized tests to be tested efficiently.
- Heuristic testing strategy:
  - Skip loop entirely.
  - 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.

Security-related bugs

- Buffer overflows provide attackers with attack vectors.
- Failure to initialize buffers can leak information to attackers.

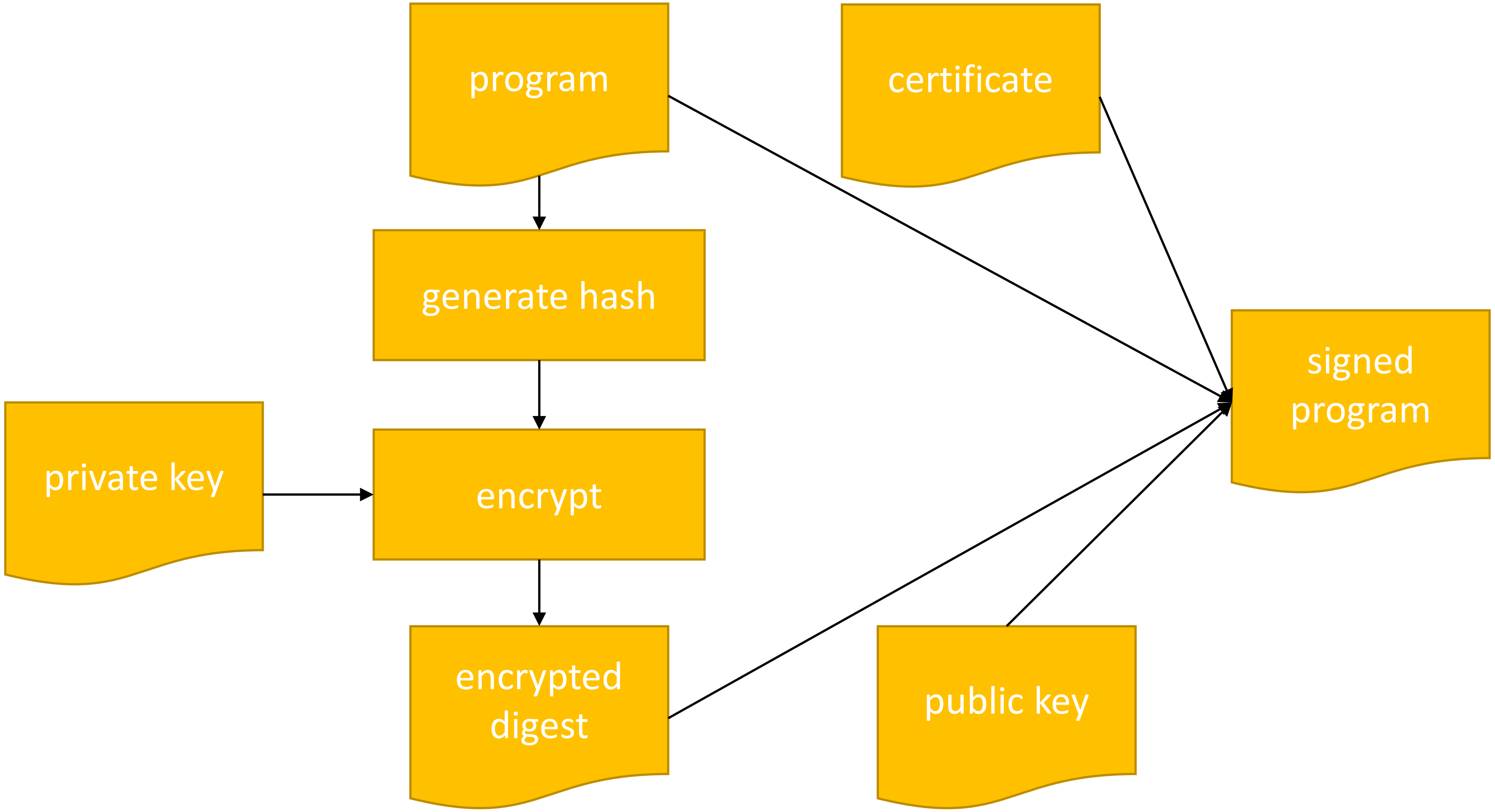
Code certificates

- Provide assurance that code comes from a trusted source:
  - Certification authority provides certificate service.
- Certificate:
  - Certification authority is given code/data.
  - Generates certificate with source for data and associated encryption date.
  - May include expiration date.
- Recipient checks source identifier before decrypting and using code/data.

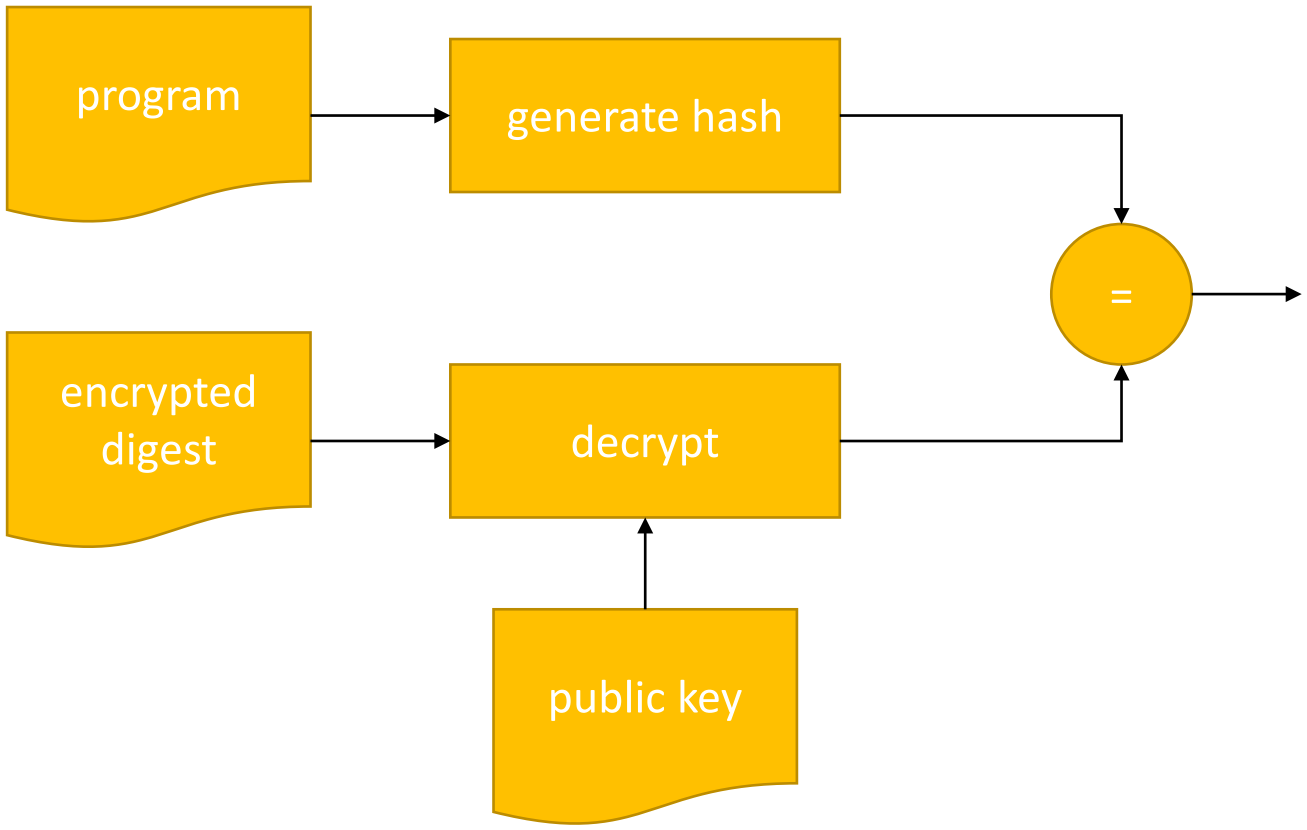
Code signing

- Combines signing certificate with digital signature.
  - Hash code to provide a digest.
  - Signature includes original program, encrypted digest, public key, signing certificate.

Code signing process



Checking signed code



Passwords

- Passwords should be stored in encrypted form.