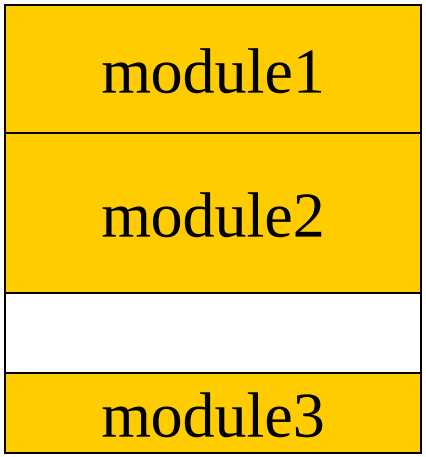
Linking

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

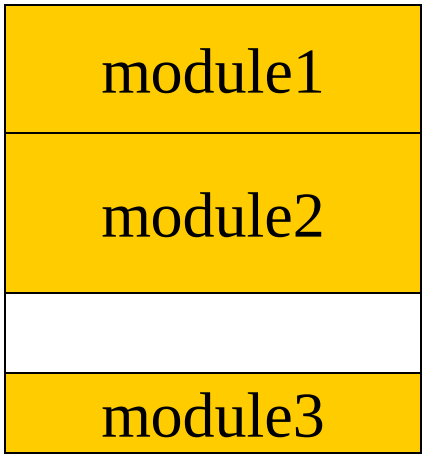
Module ordering

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



Module ordering

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



Linker and Loader

- The linker does the symbol resolution
- The loader does the program loading
- Either of them can do the relocation.

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

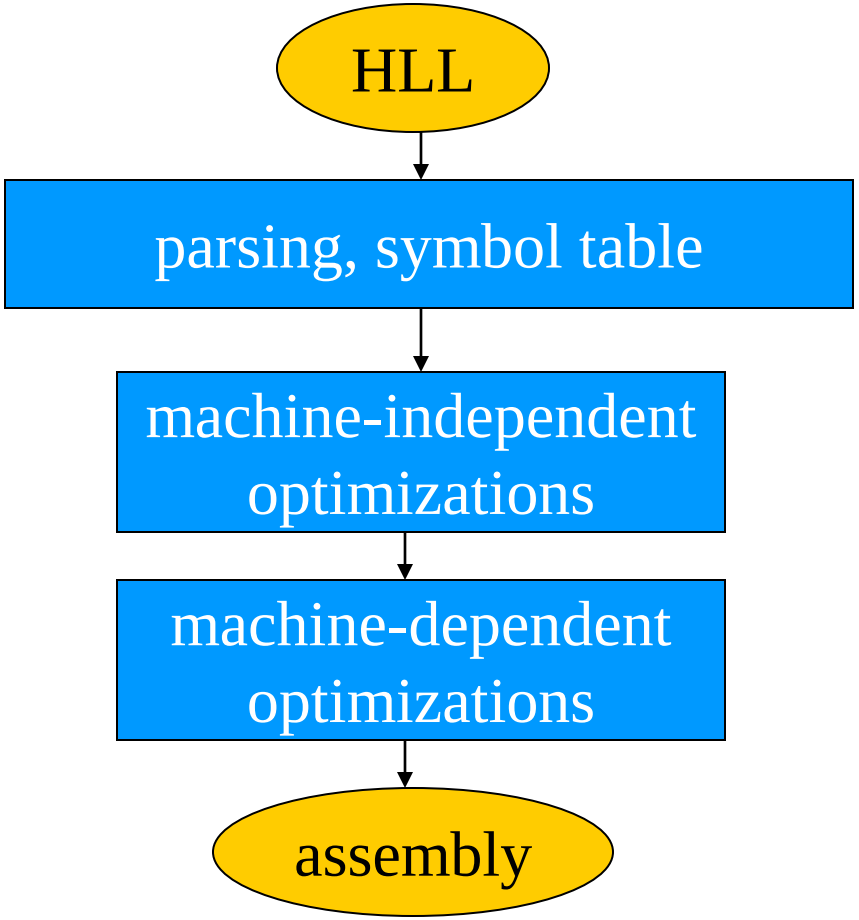
5.4 Basic compilation techniques

- 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



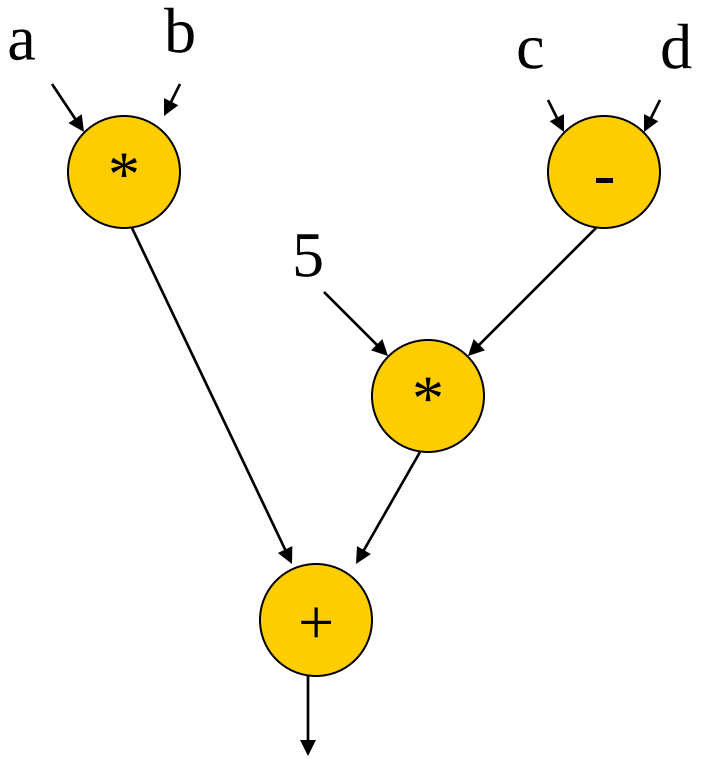
Statement translation and optimization

- 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

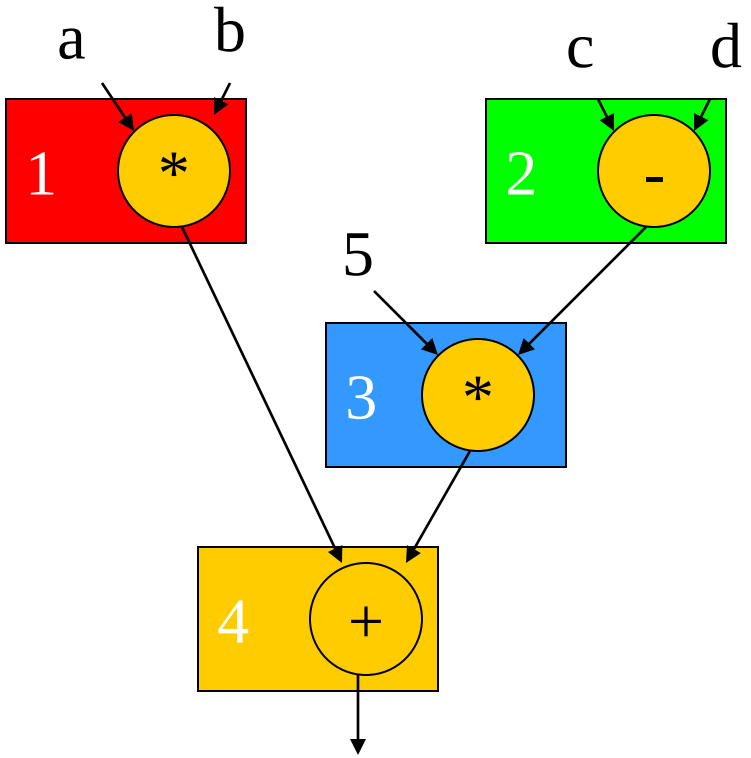
a*b + 5*(c-d)

expression



DFG

Arithmetic expressions



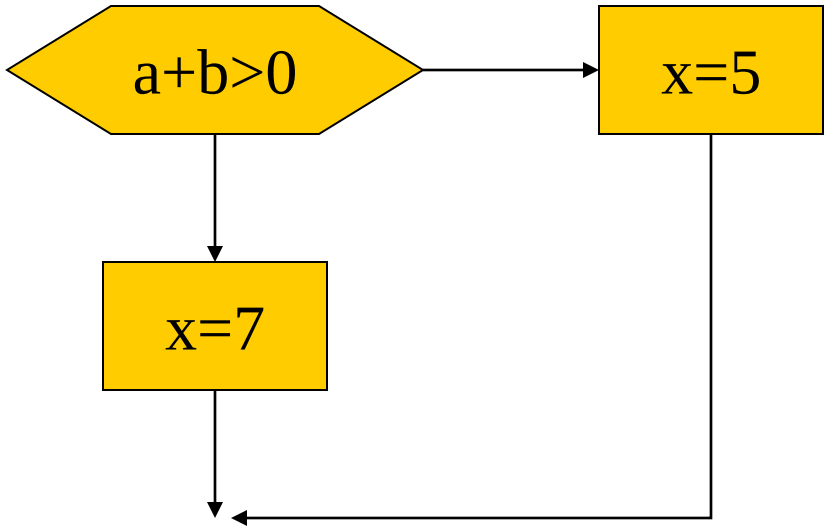
DFG

```
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
```

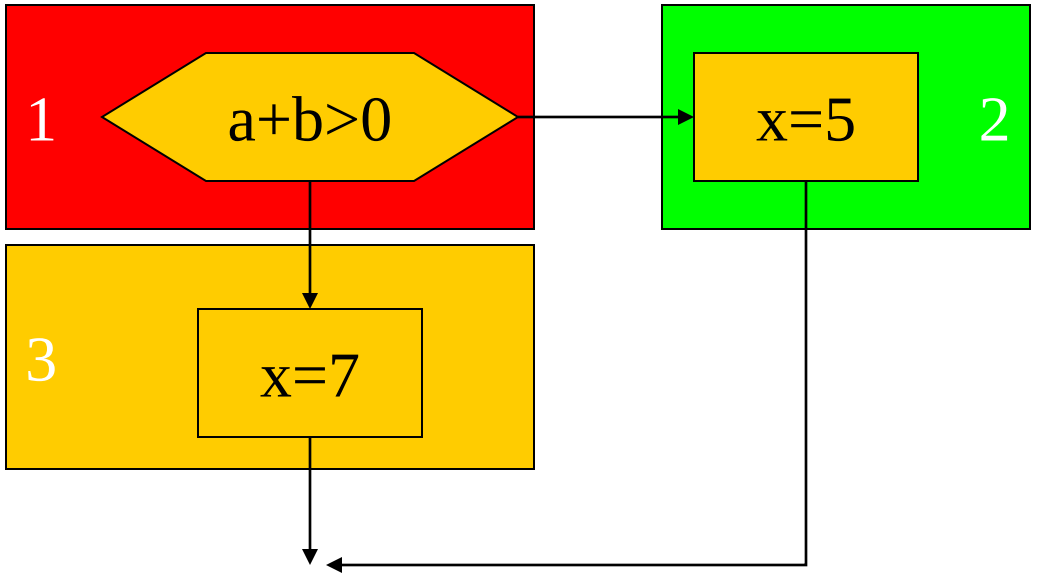
code

Control code generation

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



Control code generation

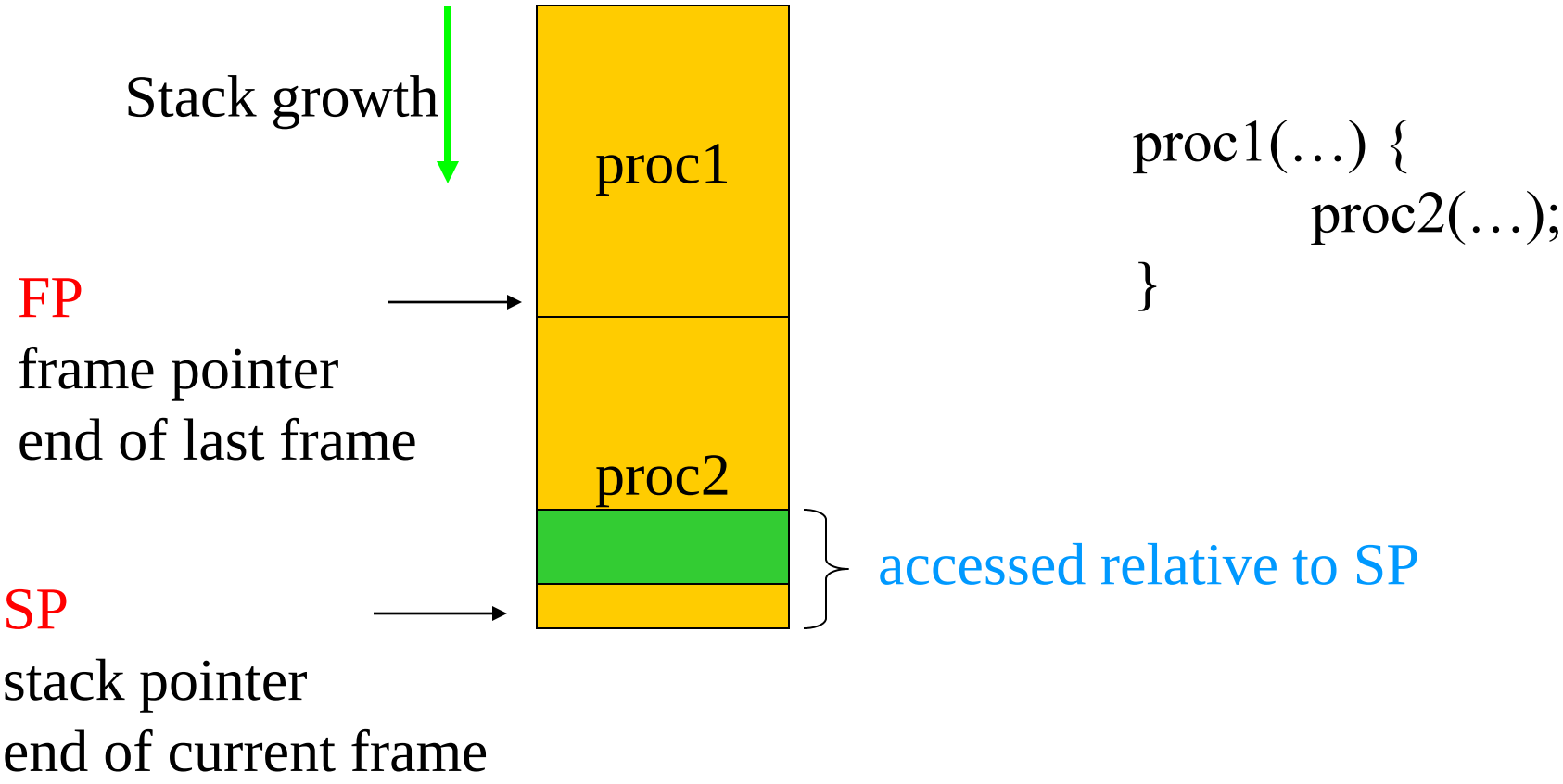



```
        ADR r5,a
        LDR r1,[r5]
        ADR r5,b
        LDR r2,b
        ADD r3,r1,r2
        BLE label3
        LDR r3,#5
        ADR r5,x
        STR r3,[r5]
        B label4
label3  LDR r3,#7
        ADR r5,x
        STR r3,[r5]
label4  ...
```

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

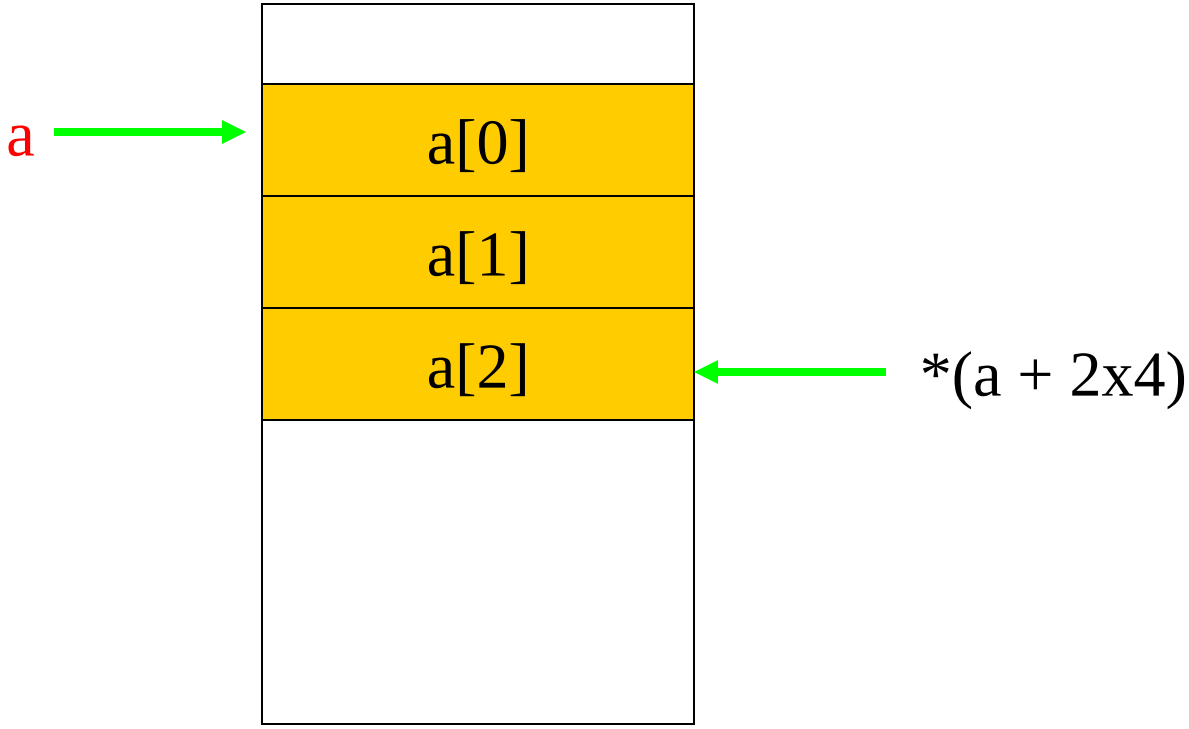
- APCS (ARM Procedure Call Standard):
  - r0-r3 pass parameters into procedure. Extra parameters are put on stack frame.
  - r0 holds return value.
  - r4-r7 hold register values.
  - r11 is frame pointer, r13 is stack pointer.
  - r10 holds limiting address on stack size to check for stack overflows.

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

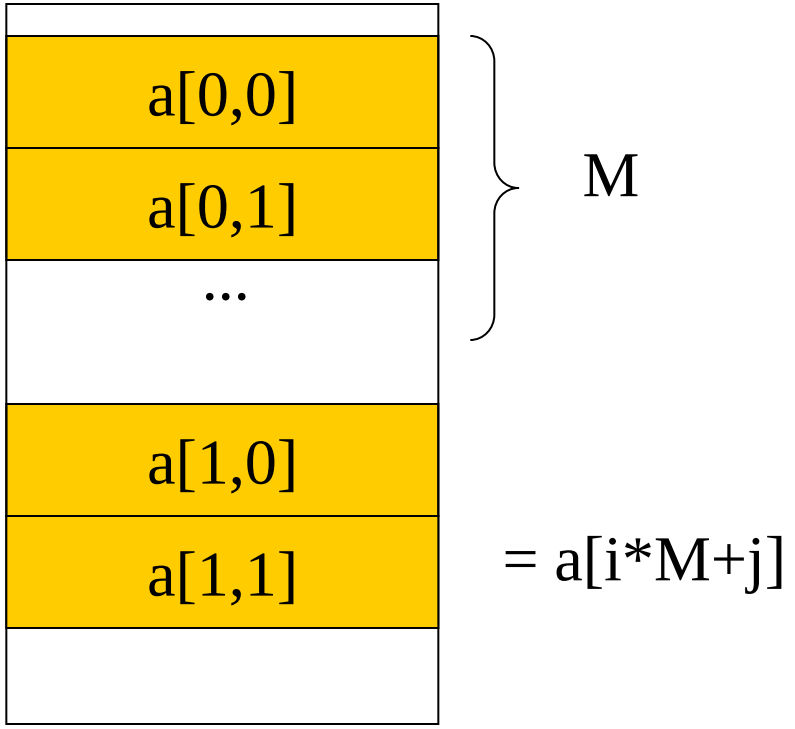
- C array name points to 0th element:



Two-dimensional arrays

- Row-major layout: a[i,j]

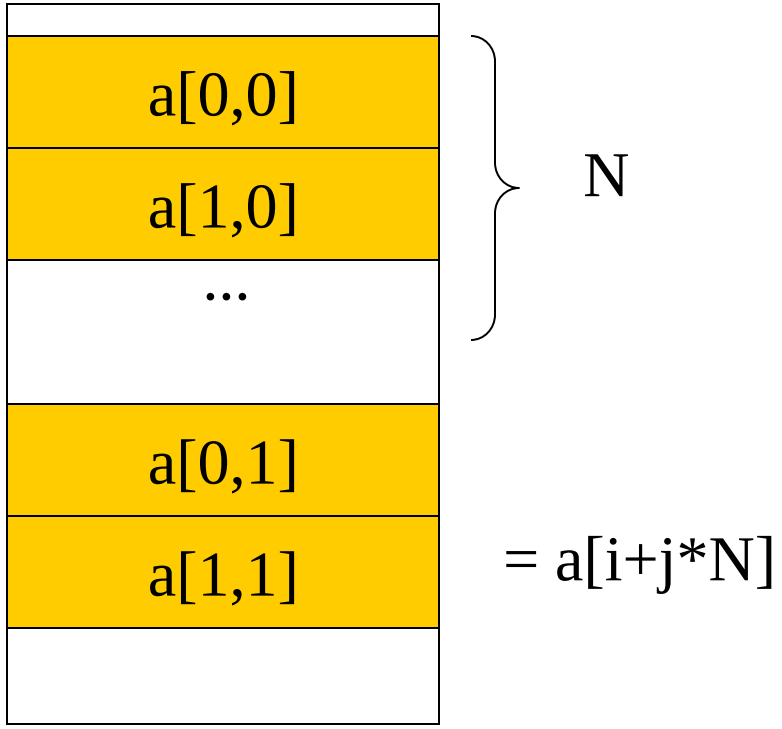
Inner variable j varies more quickly



Array size: a[N,M]

Two-dimensional arrays

- Column-major layout: a[i,j] FORTRAN



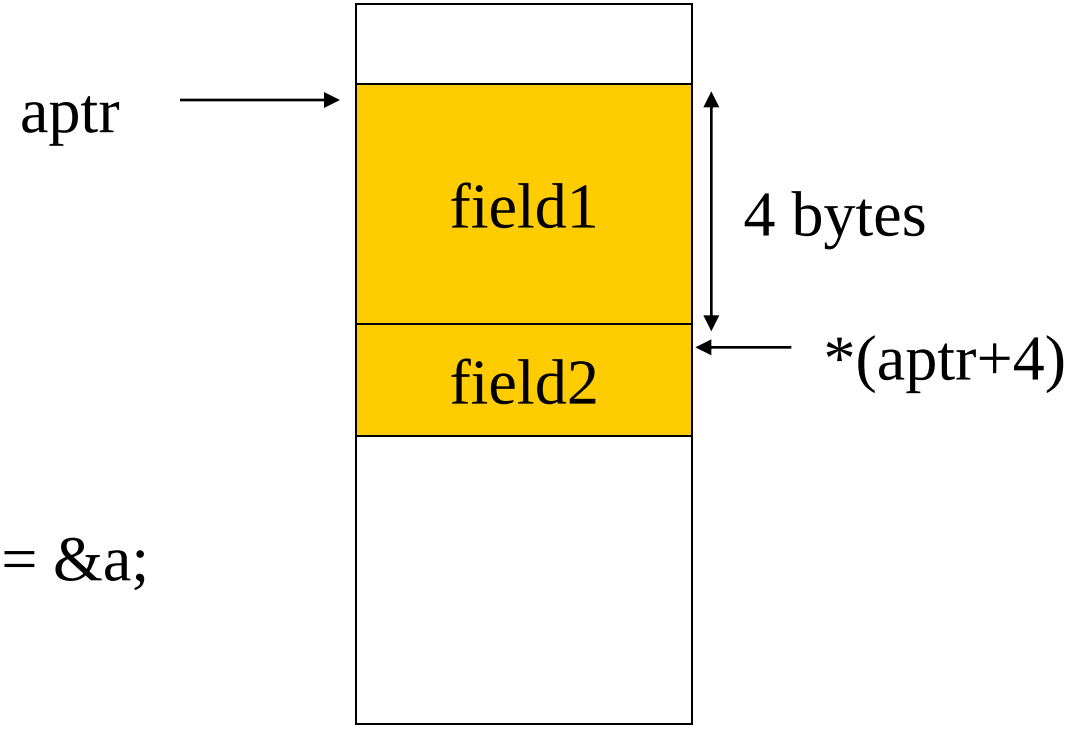
Array size: a[N,M]

Structures

- Fields within structures are static offsets:

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

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



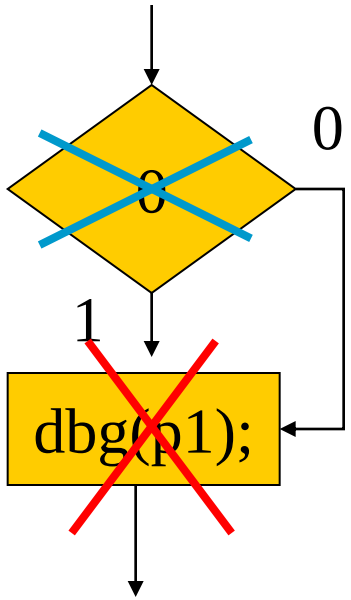
Expression simplification

- Machine independent transformation
- Constant folding:
  - 8+1 = 9
- Expression simplification:
  - a*b + a*c = a*(b+c)
- Strength reduction:
  - a*2 = a<<1

Dead code elimination

- Dead code: code that never be executed
- 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 one:

```
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.

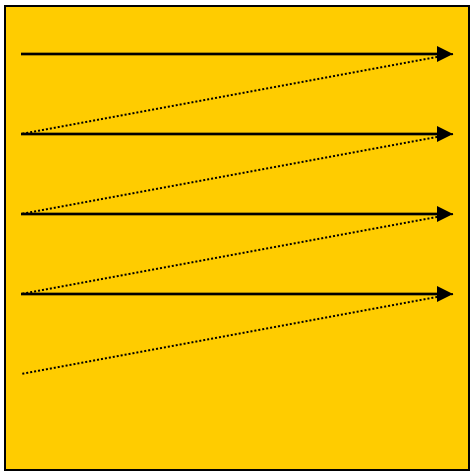
Loop tiling

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

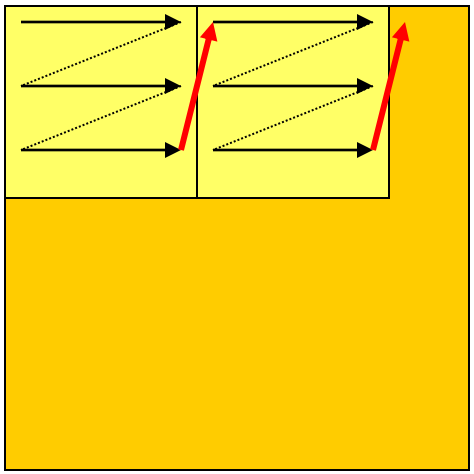
Loop tiling example

```
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];
```



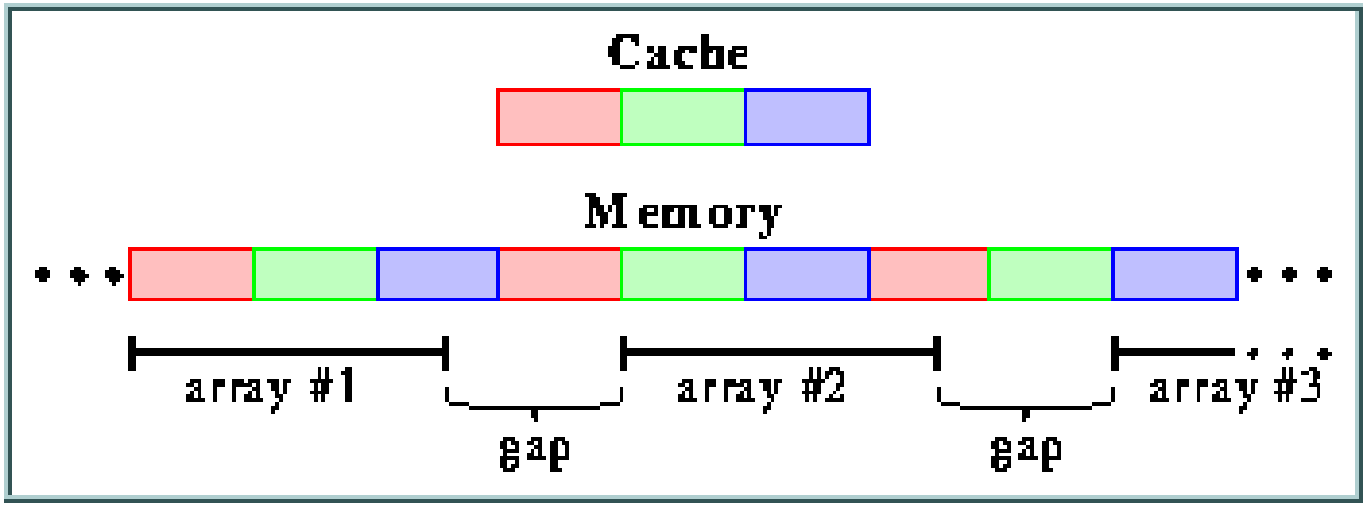
3N



3n

Array padding

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



Register allocation

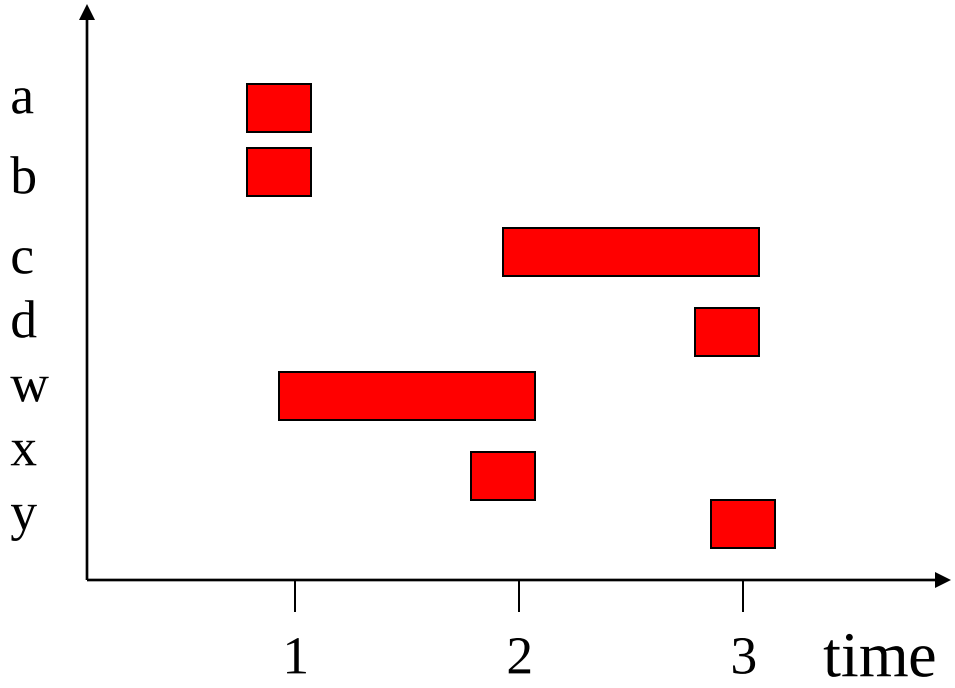
- Goals:
  - choose register to hold each variable;
  - determine lifespan of variable in the register.
- Basic case: within basic block.
- Spilling registers: problematic

Register lifetime graph

w = a + b; t=1

x = c + w; t=2

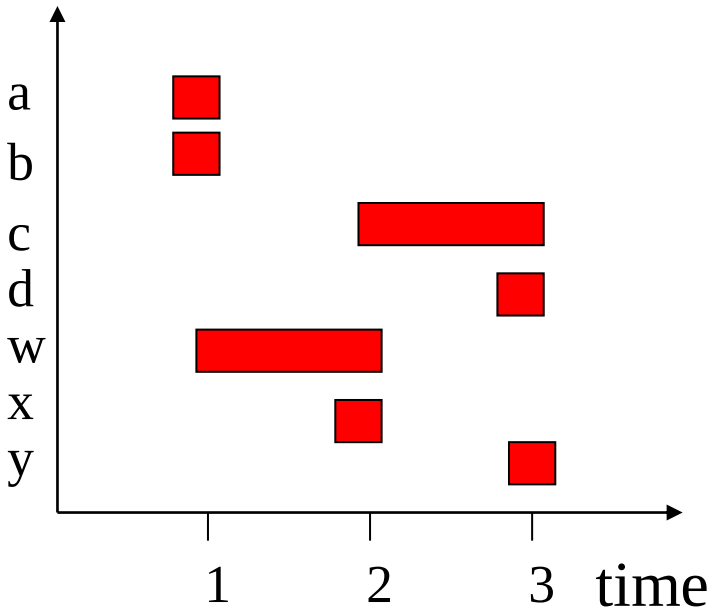
y = c + d; t=3

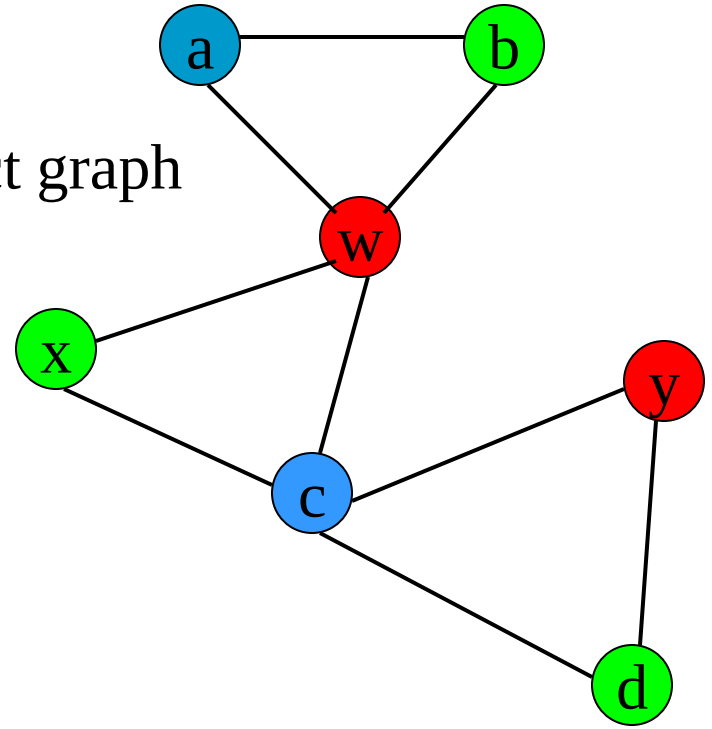


Register assignment

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

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 relates instructions/time to CPU resources.

Resource	A	B
instr1	X	
instr2	X	X
instr3	X	
instr4		X

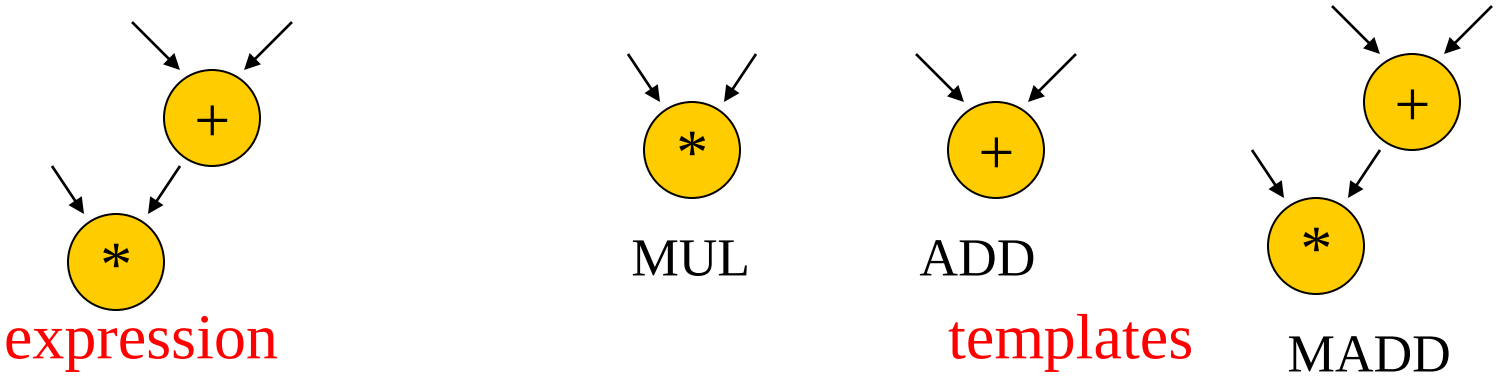
time

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.



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

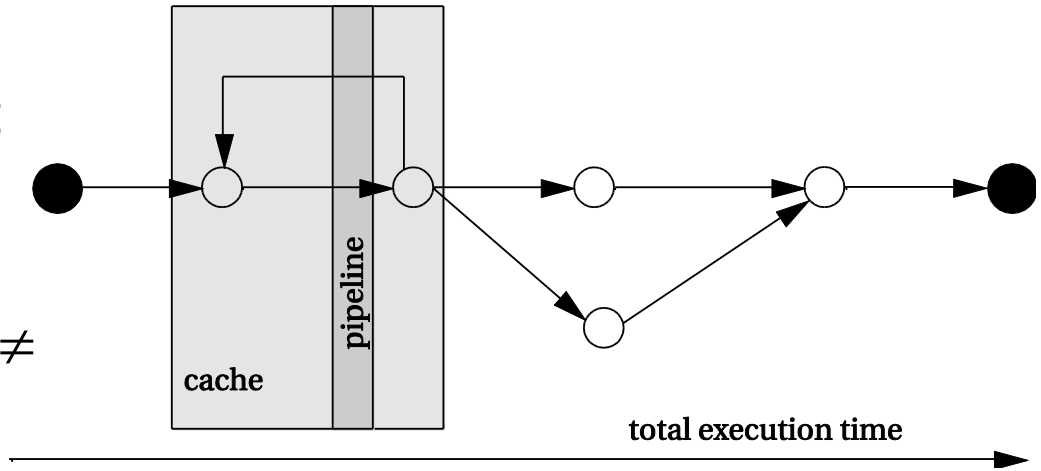
- Interpreter: translates and executes program statements on-the-fly.
- JIT compiler: compiles small sections of code into instructions during program execution.
  - Eliminates some translation overhead.
  - Often requires more memory.
- Javascript: script executed in web browser

5.6 Program level performance analysis

- Optimizing for:
  - Execution time.
  - Energy/power.
  - Program size.
- Program validation and testing.

Program-level performance analysis

- Need to understand performance in detail:
  - Real-time behavior, not just typical.
  - On complex platforms.
- Program performance ≠ 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.

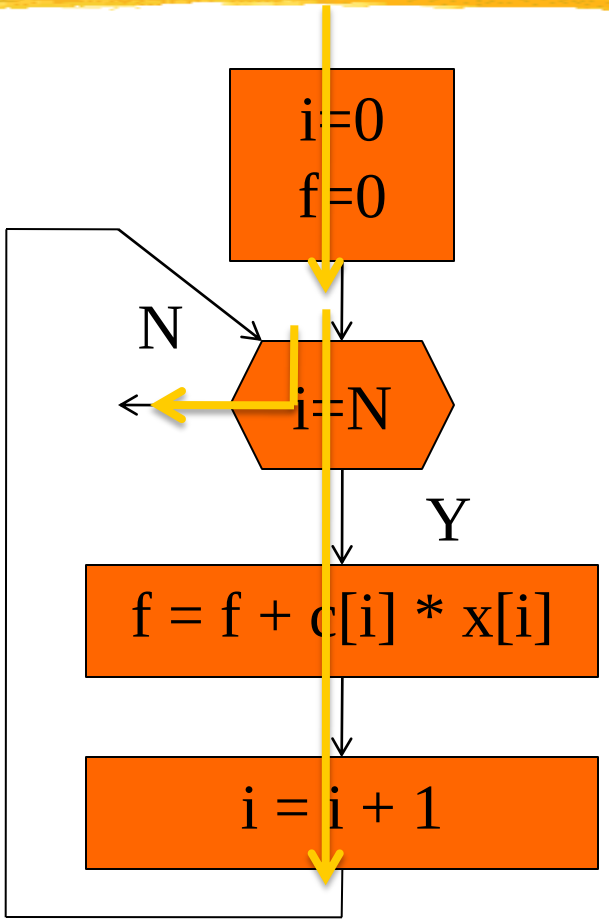
Data-dependent paths in an if statement

```
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];
```



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.

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, f=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

Loop set up: 1
Loop test: N+1

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.

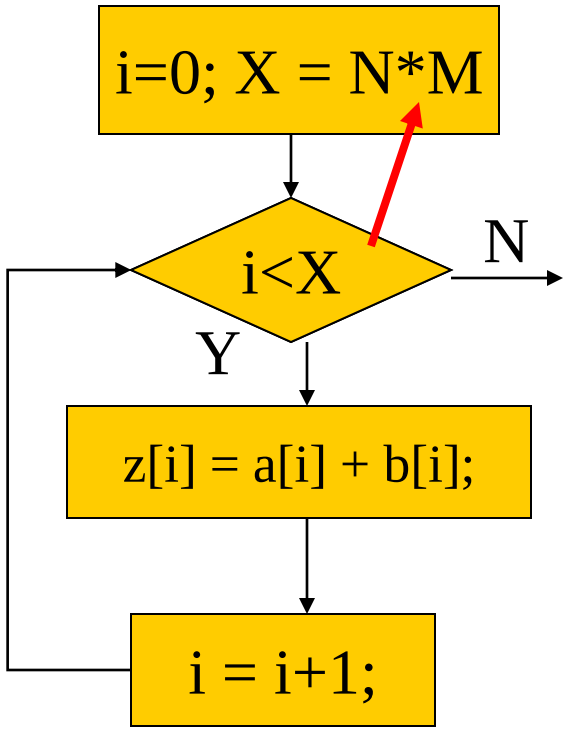
Loop optimizations

- Loops are good targets for optimization
  - Why?
- 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];
```

Performed (NM-1) times



Induction variable elimination

- Induction variable: its value is derived form the 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.

Strength reduction

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

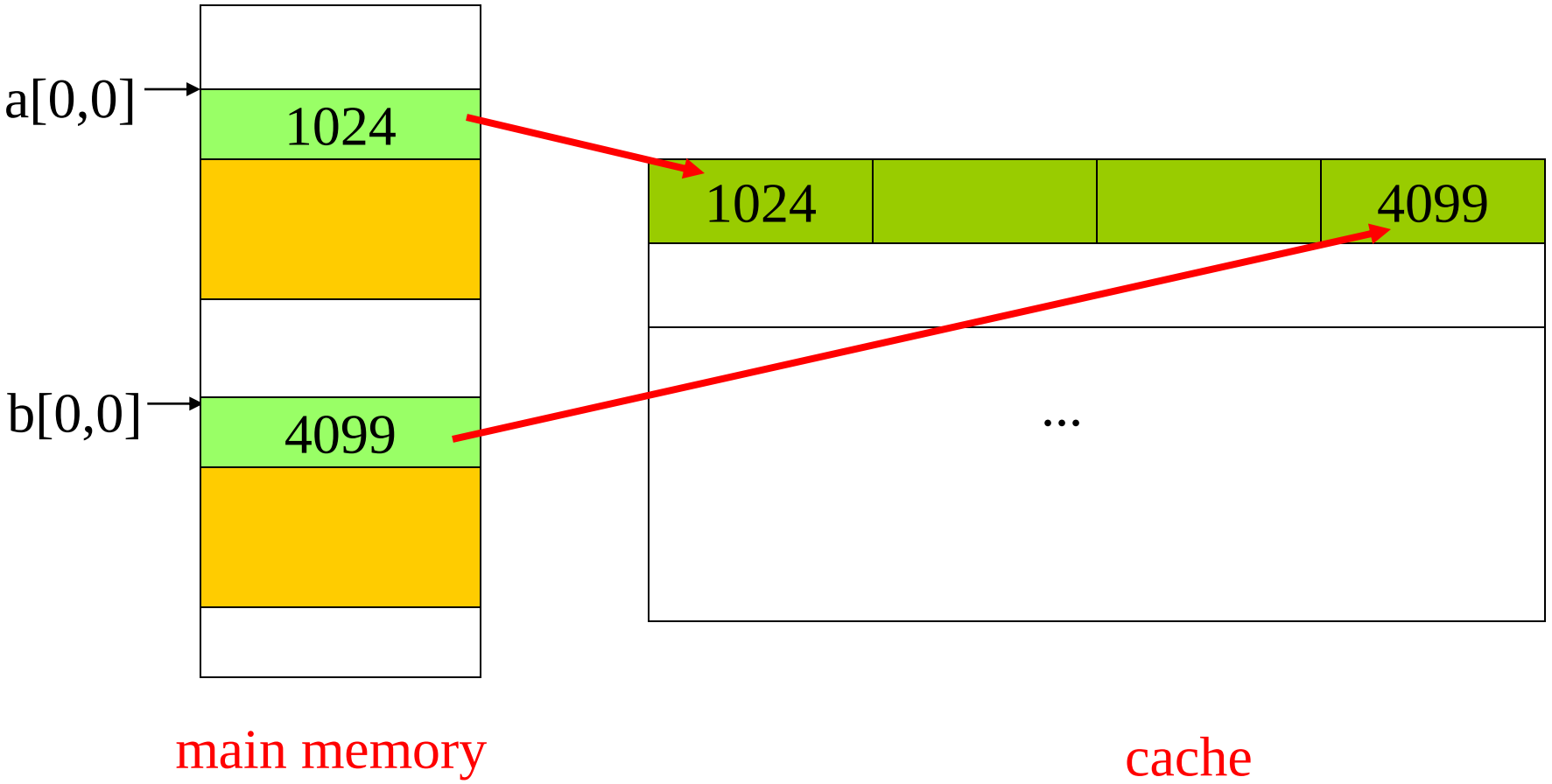
- Better code with strength reduction

```
xbinduct = 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



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.

Performance optimization hints

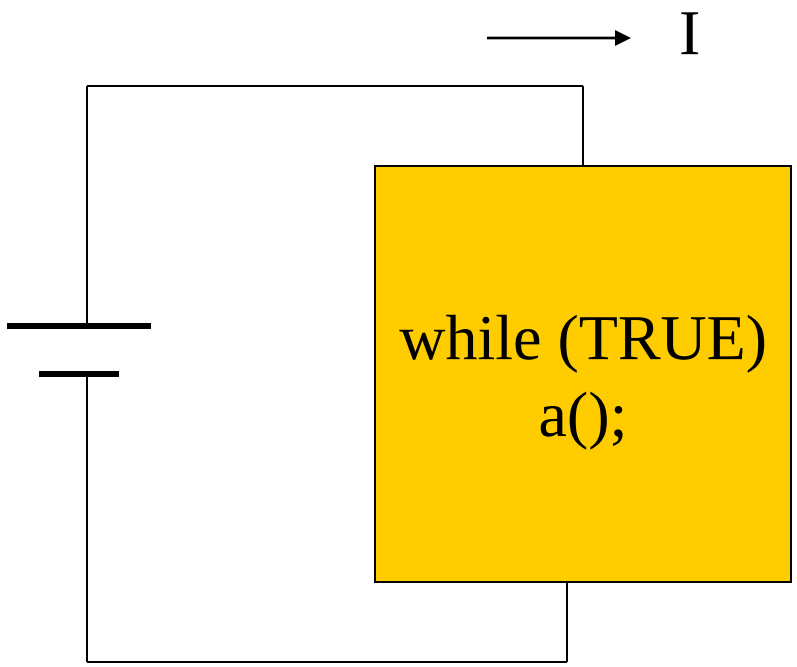
- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
  - instruction conflicts can be handled by rewriting code, rescheudling;
  - conflicting scalar data can easily be moved;
  - conflicting array data can be moved, padded.

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:



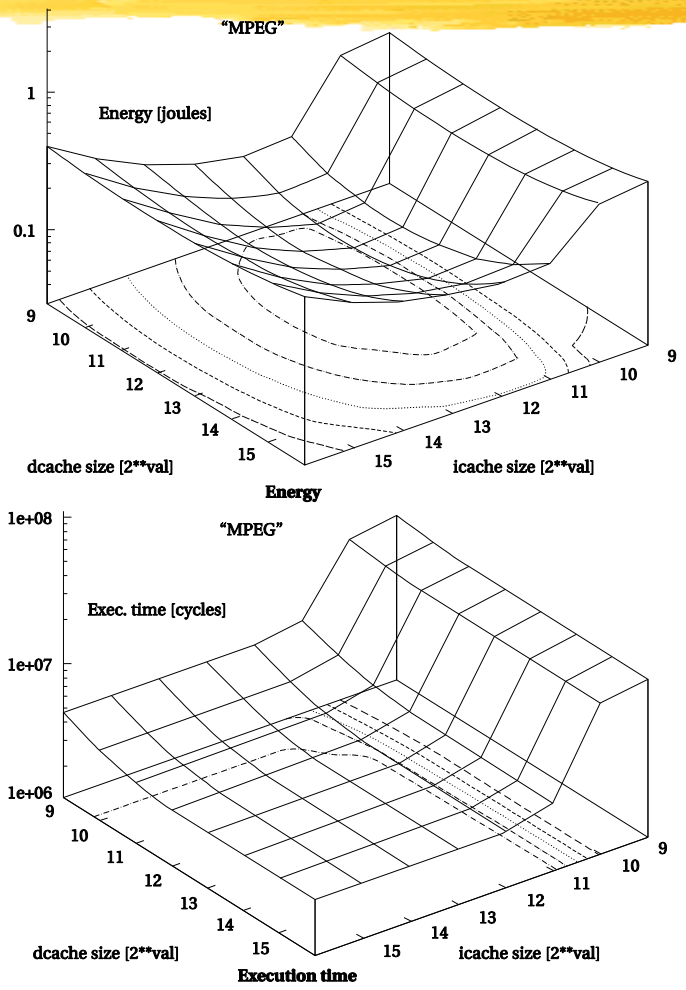
Sources of energy consumption

- 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.

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: rovide 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;
```

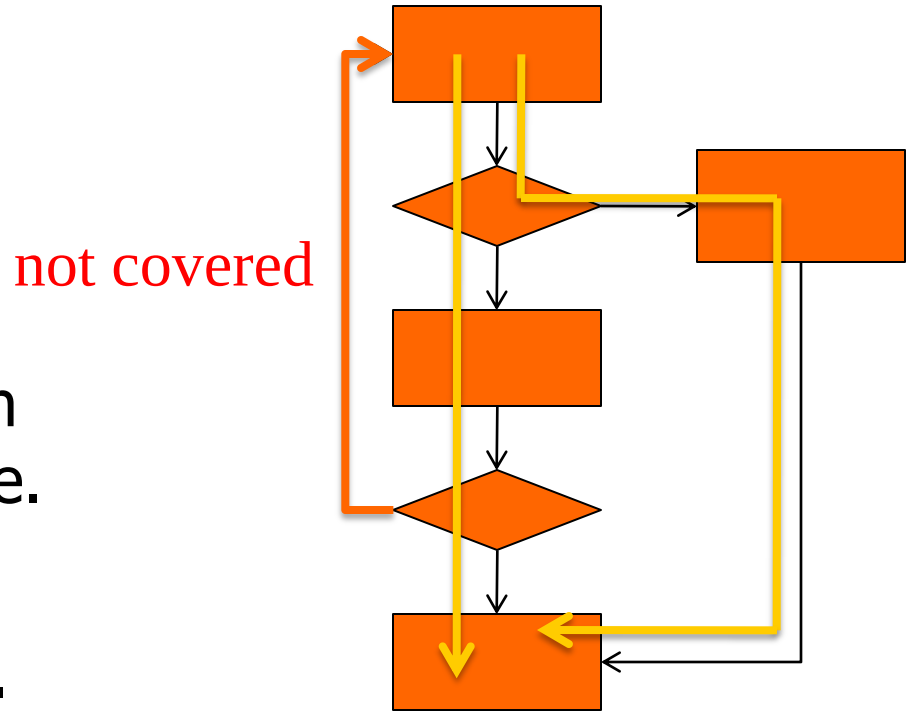
- Controllability:
  - Must fill circular buffer with desired N values.
  - Other code governs how we access the buffer.
- Observability:
  - Want to examine firout before limit testing.

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.

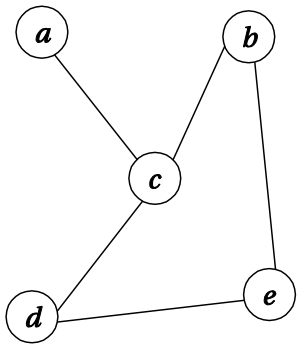
Choosing the paths to test

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



Basis paths

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



Graph

	a	b	c	d	e
a	0	0	1	0	0
b	0	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

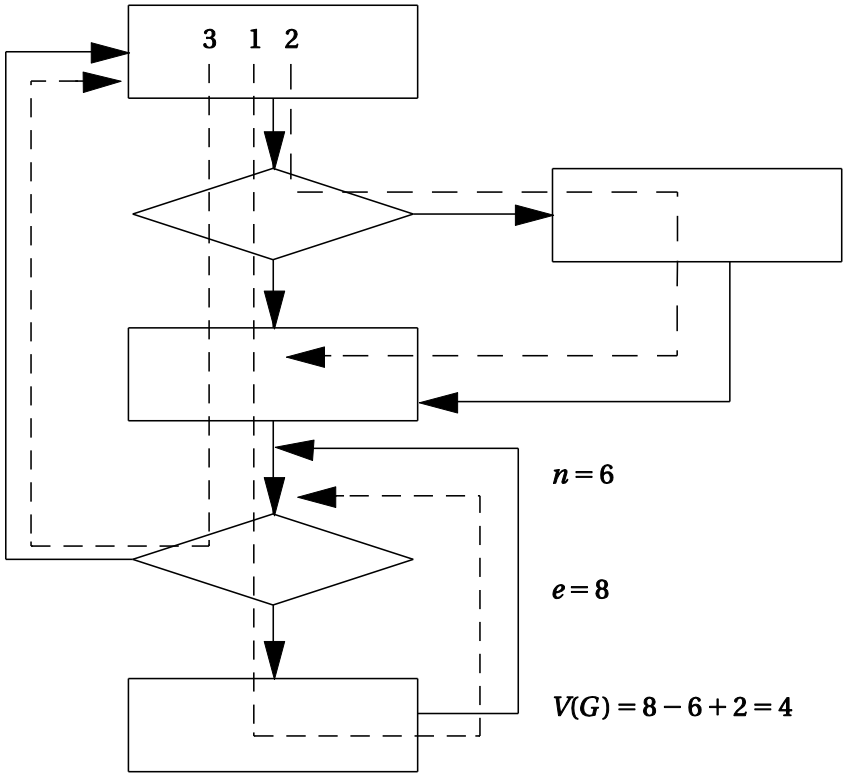
Incidence matrix

a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
e	0	0	0	0	1

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

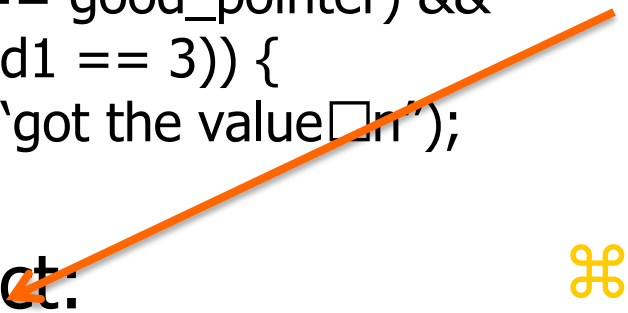
- Heuristic for testing branches.
  - Exercise true and false branches of conditional.
  - Exercise every simple condition at least once.

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

Another branch testing example

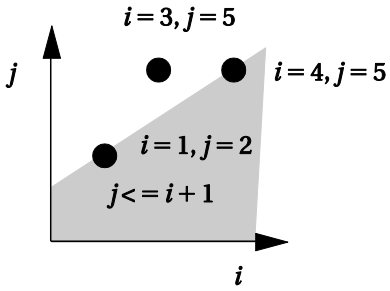
- Correct:
  - if ((x == good_pointer) && x->field1 == 3)) { printf("got the value\n"); }
- Incorrect:
  - if ((x = good_pointer) && x->field1 == 3)) { printf("got the value\n"); }



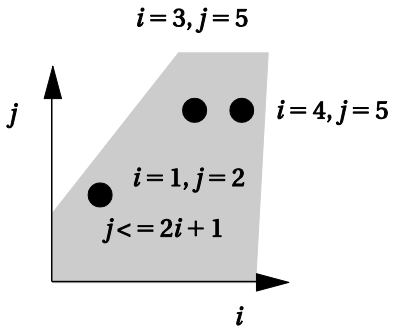
- Incorrect code changes pointer.
  - Assignment returns new LHS in C.
- Test that catches error:
  - (x != good_pointer) && x->field1 = 3)

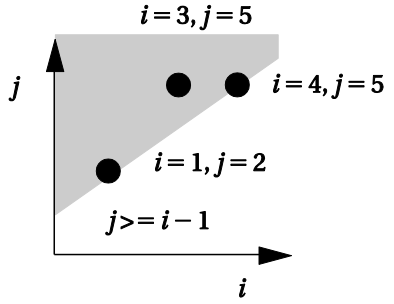
Domain testing

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



Correct test

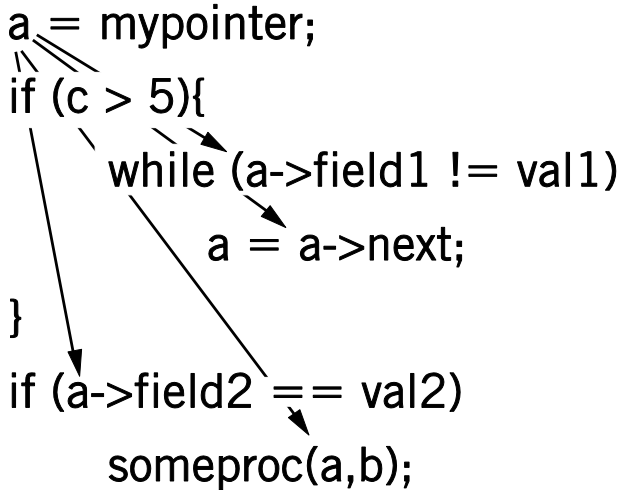




Incorrect tests

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.



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?

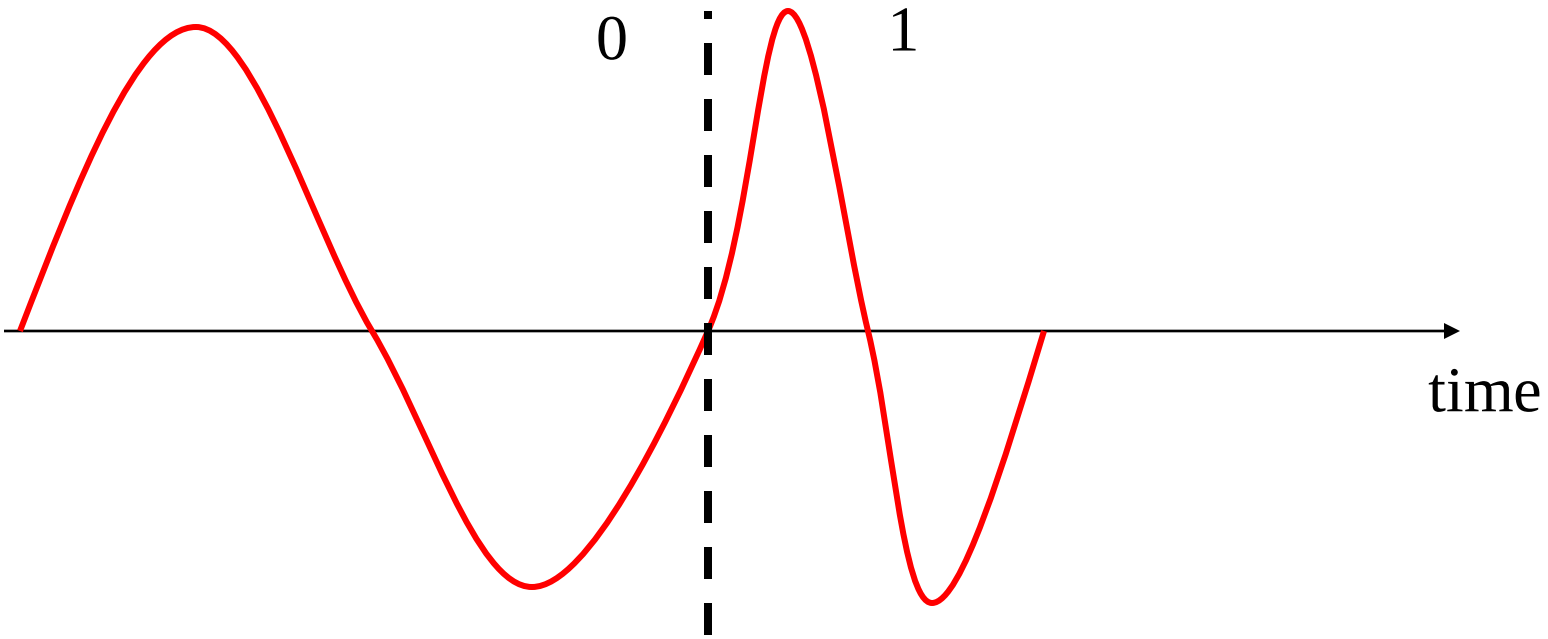
- Exhaustive testing is impractical.
- One important measure of test quality---bugs escaping into field.
- Good organizations can test software to give very low field bug report rates.
- Error injection measures test quality:
  - Add known bugs.
  - Run your tests.
  - Determine % injected bugs that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

Program design and analysis

- Software modem.

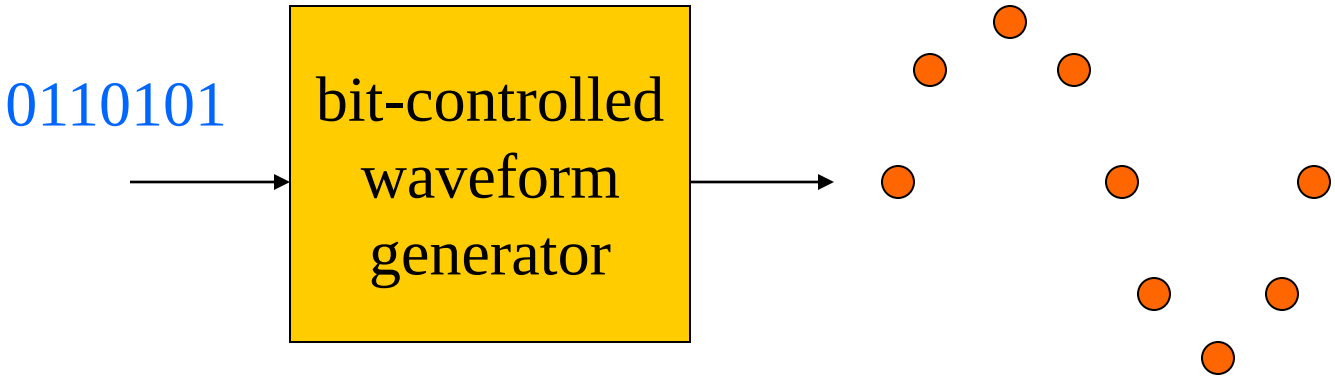
Theory of operation

- Frequency-shift keying:
  - separate frequencies for 0 and 1.

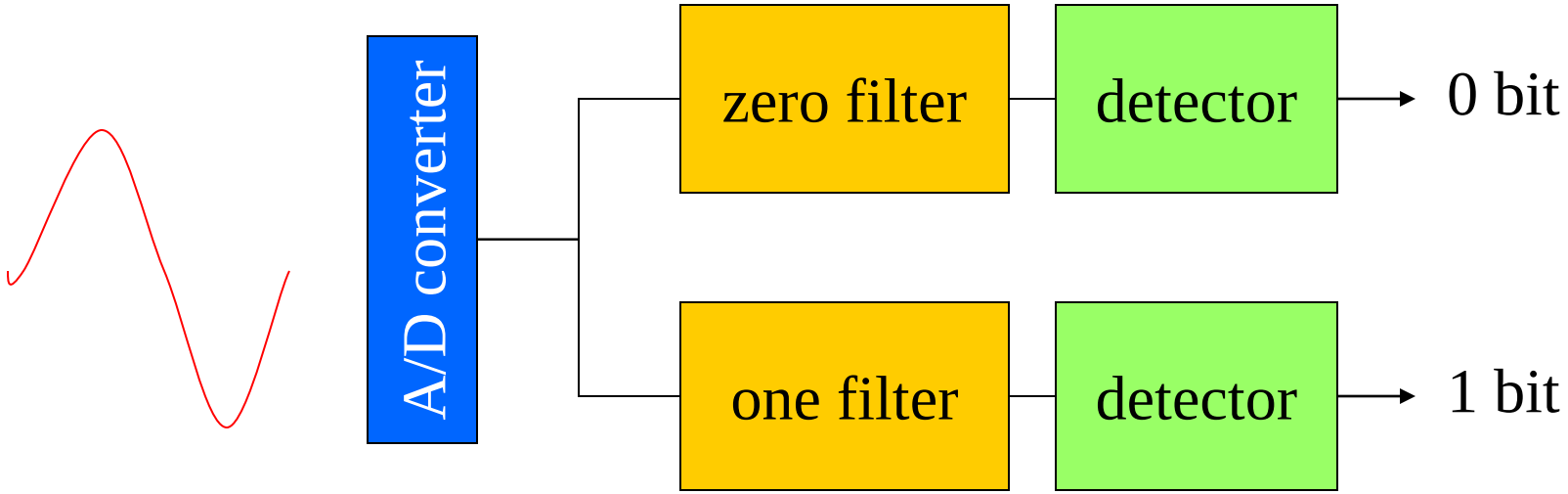


FSK encoding

- Generate waveforms based on current bit:

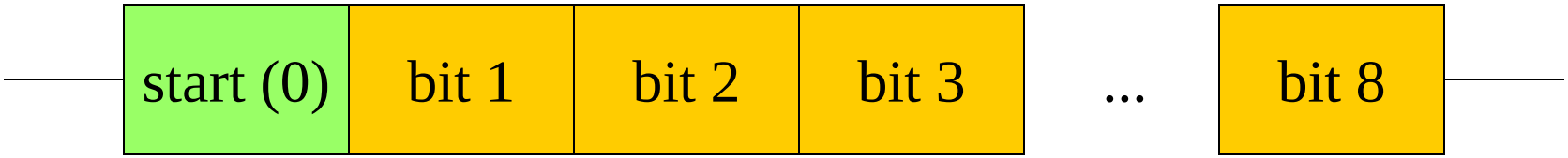


FSK decoding



Transmission scheme

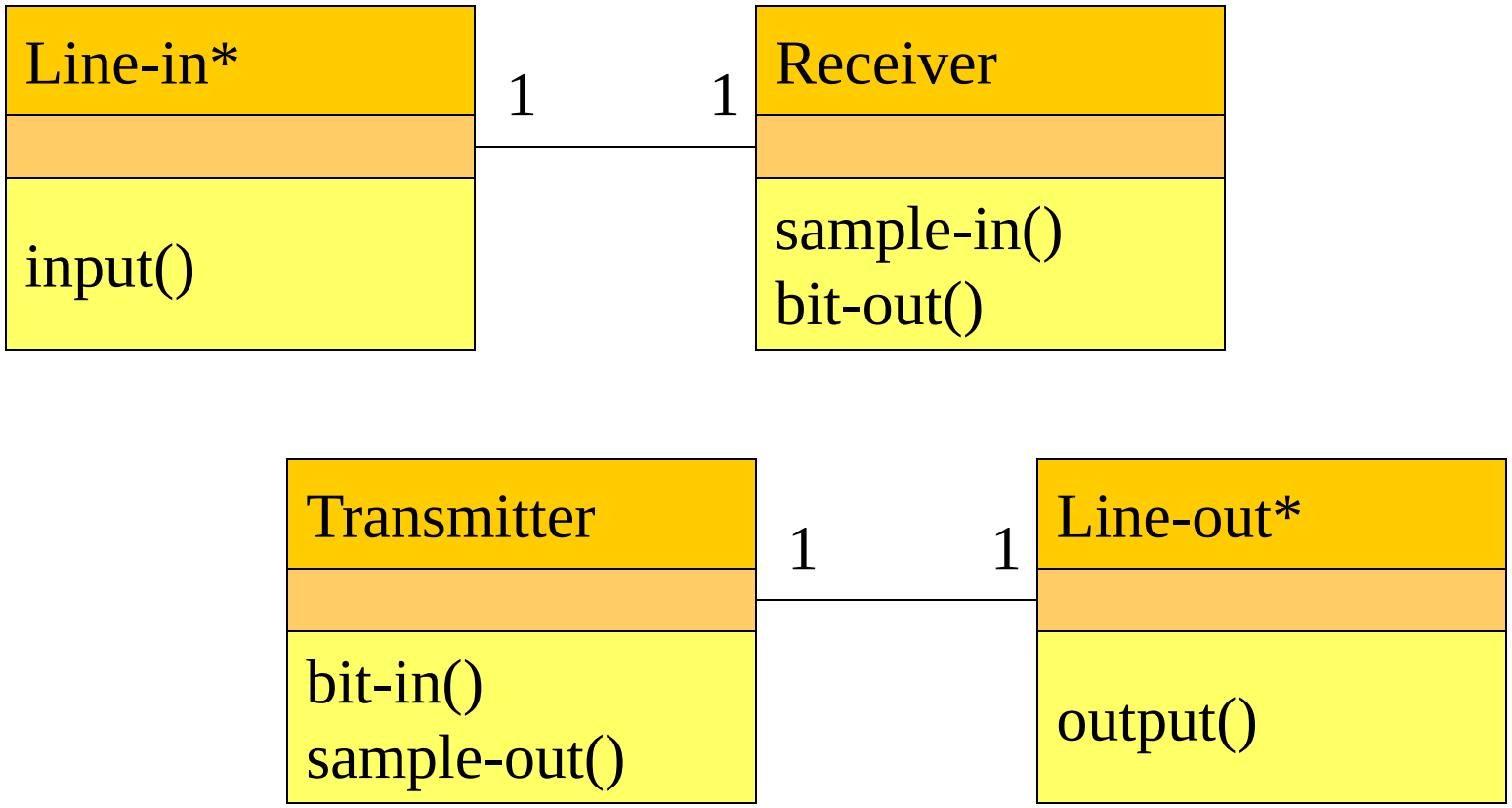
- Send data in 8-bit bytes. Arbitrary spacing between bytes.
- Byte starts with 0 start bit.
- Receiver measures length of start bit to synchronize itself to remaining 8 bits.



Requirements

Inputs	Analog sound input, reset button.
Outputs	Analog sound output, LED bit display.
Functions	Transmitter: Sends data from memory in 8-bit bytes plus start bit. Receiver: Automatically detects bytes and reads bits. Displays current bit on LED.
Performance	1200 baud.
Manufacturing cost	Dominated by microprocessor and analog I/O
Power	Powered by AC.
Physical size/weight	Small desktop object.

Specification

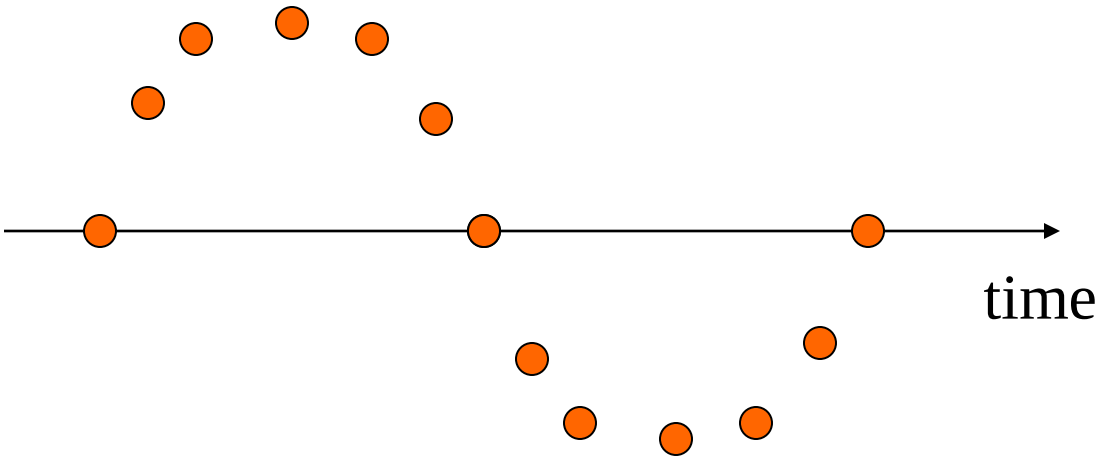


System architecture

- Interrupt handlers for samples:
  - input and output.
- Transmitter.
- Receiver.

Transmitter

- Waveform generation by table lookup.
  - float sine_wave[N_SAMP] = { 0.0, 0.5, 0.866, 1, 0.866, 0.5, 0.0, -0.5, -0.866, -1.0, -0.866, -0.5, 0};



Receiver

- Filters (FIR for simplicity) use circular buffers to hold data.
- Timer measures bit length.
- State machine recognizes start bits, data bits.

Hardware platform

- CPU.
- A/D converter.
- D/A converter.
- Timer.

Component design and testing

- Easy to test transmitter and receiver on host.
- Transmitter can be verified with speaker outputs.
- Receiver verification tasks:
  - start bit recognition;
  - data bit recognition.

System integration and testing

- Use loopback mode to test components against each other.
  - Loopback in software or by connecting D/A and A/D converters.