Program design and analysis

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

Design Pattern :Components of embedded system

- Components mainly used for embedded software: the state machine, the circular buffer, queue
- State diagram to describe behavior suited for reactive systems
- Circular buffer ,queue used for DSP
- Sequence diagram to show how classes interact

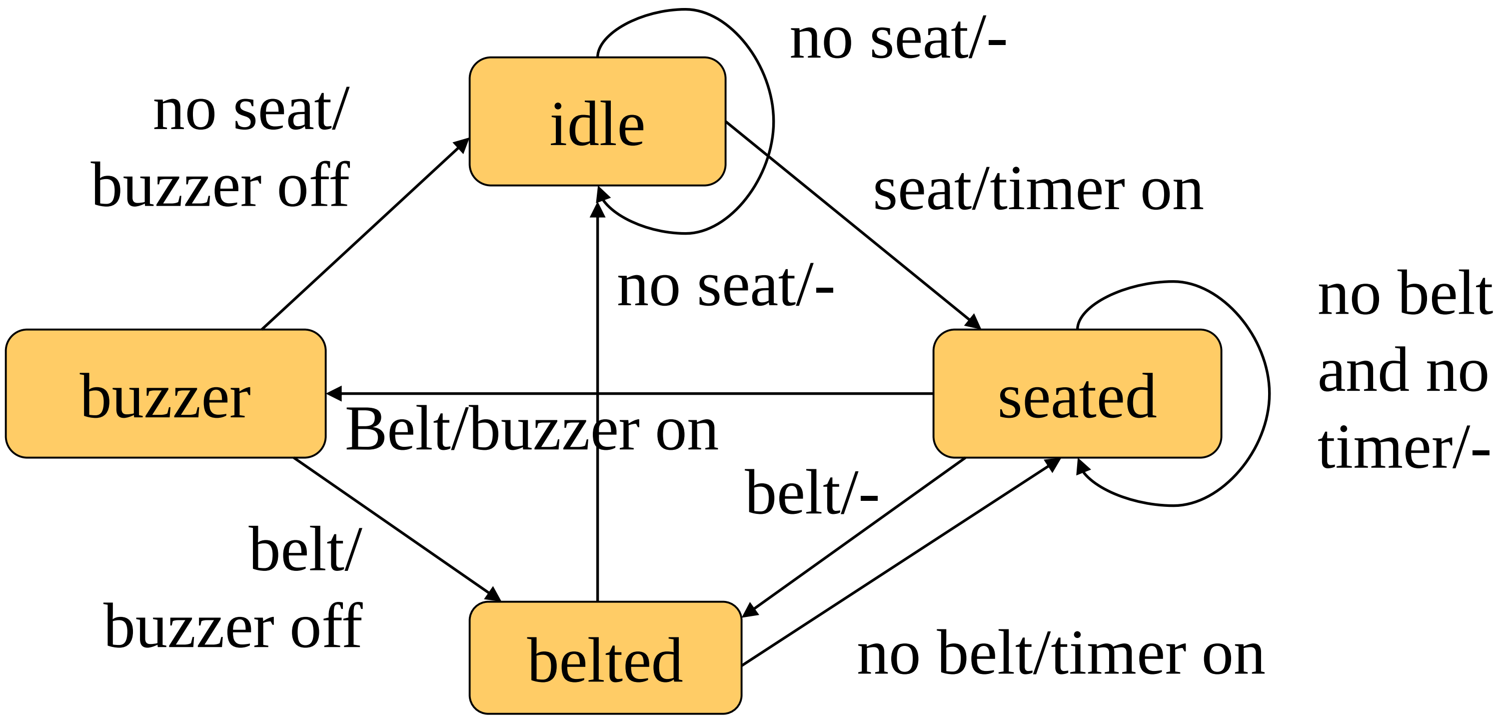
Different types of Design Pattern

- The digital filter is easily described as design pattern
- Data structures and their associated actions can be described
- Reactive system that reacts to external stimuli

Software state machine

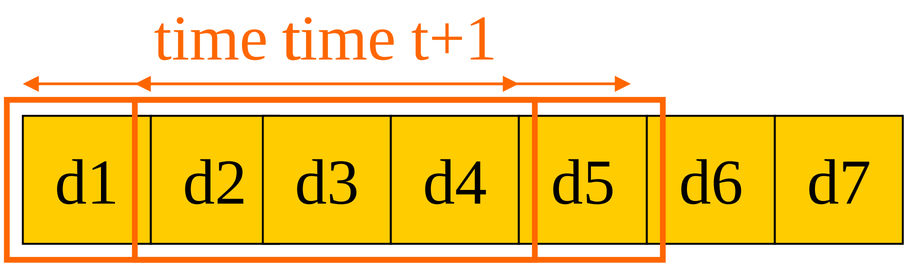
- State machine keeps internal state as a variable, changes state based on inputs.
- Uses:
  - control-dominated code;
  - reactive systems.

State machine example (Seat belt controller)



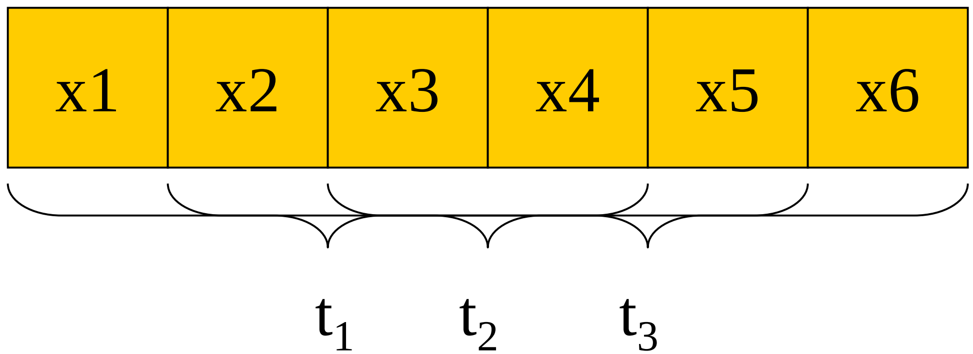
Signal processing and circular buffer

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

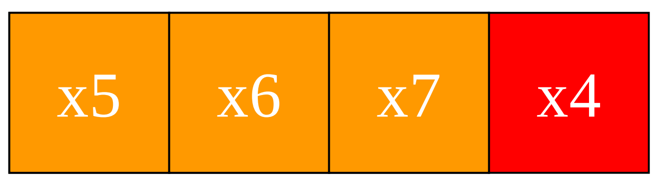


- Use a circular buffer to hold the data stream.

Circular buffer



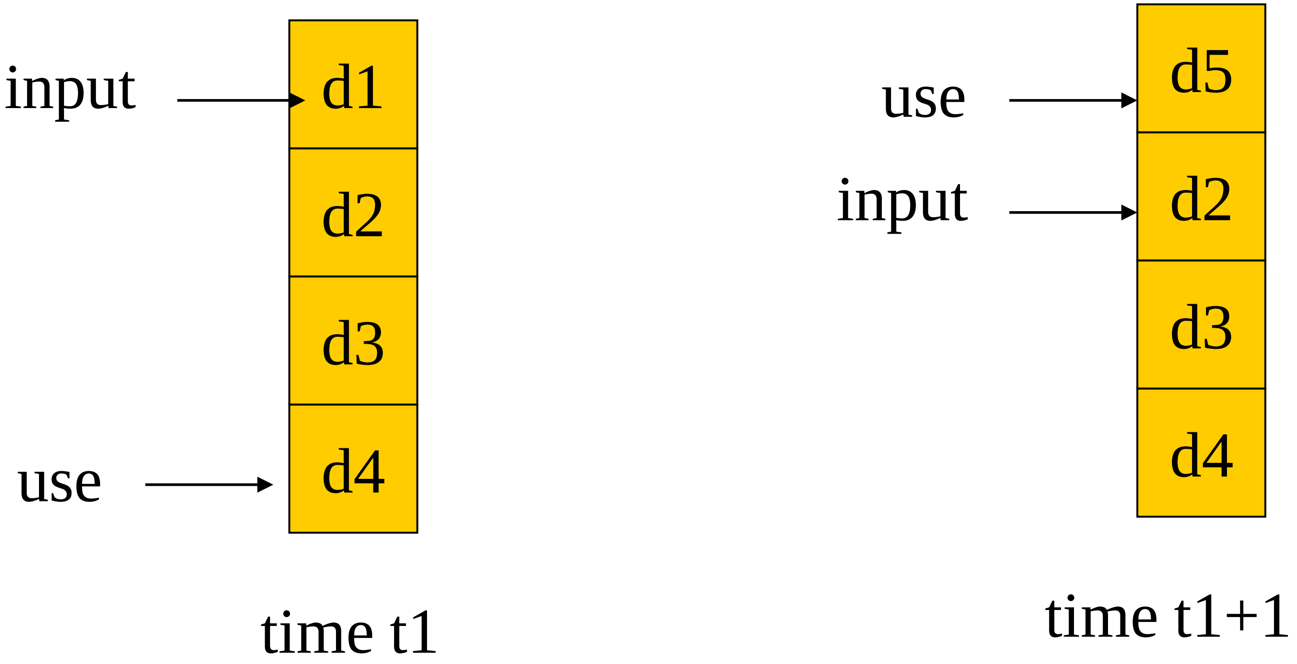
Data stream



Circular buffer

Circular buffers

- Indexes locate currently used data, current input data:



Queues

- Elastic buffer: holds data that arrives irregularly.

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.
- Round nodes-denote operators
- Square node-represent values

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

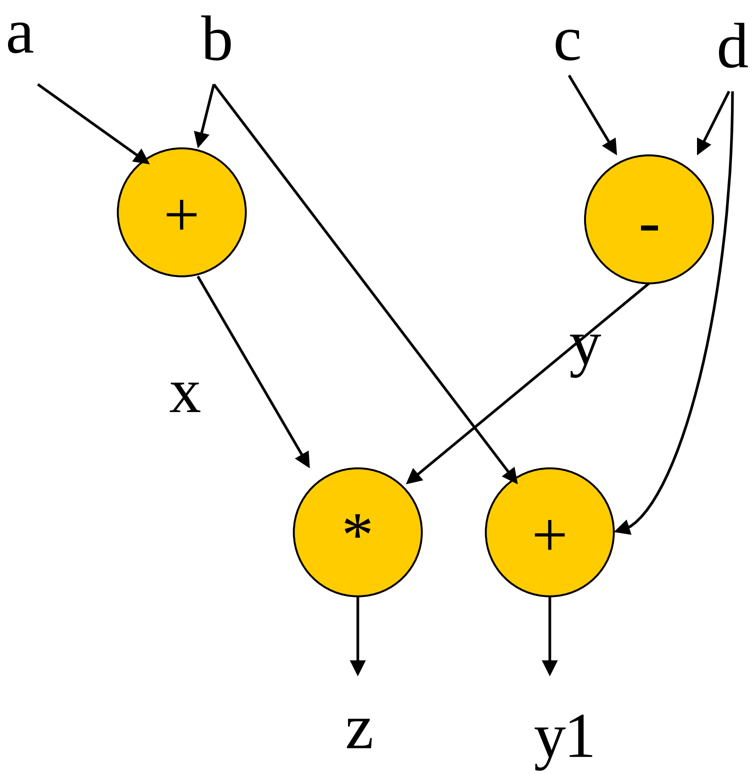
x = a + b;

y = c - d;

z = x * y;

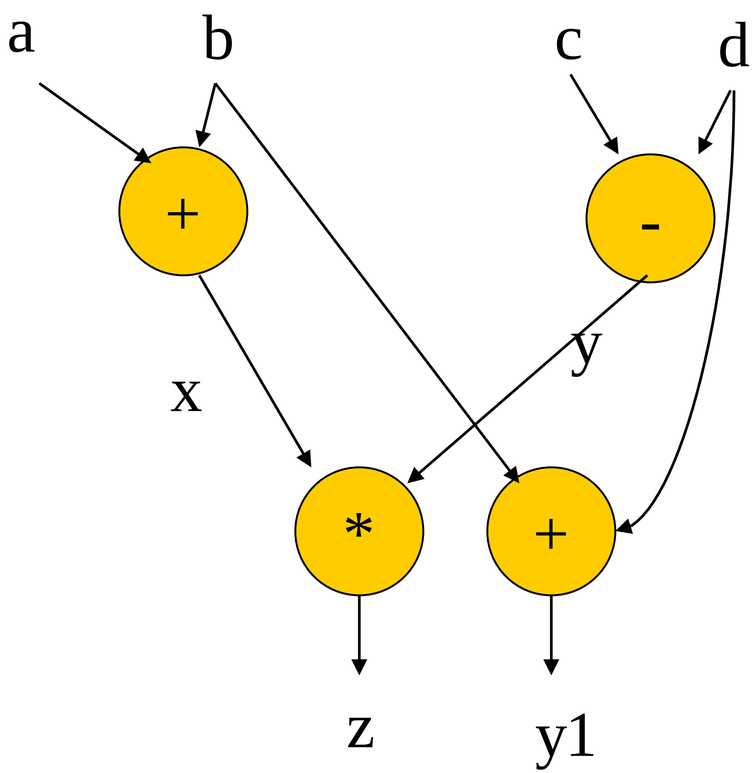
y1 = b + d;

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 node;
  - data flow node.

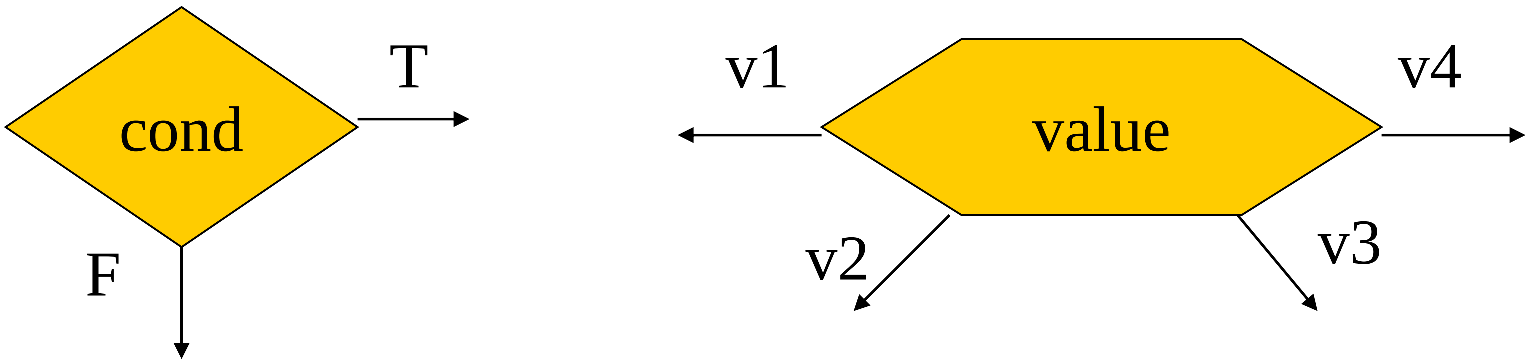
Data flow node

Encapsulates a data flow graph:

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

Write operations in basic block form for simplicity.

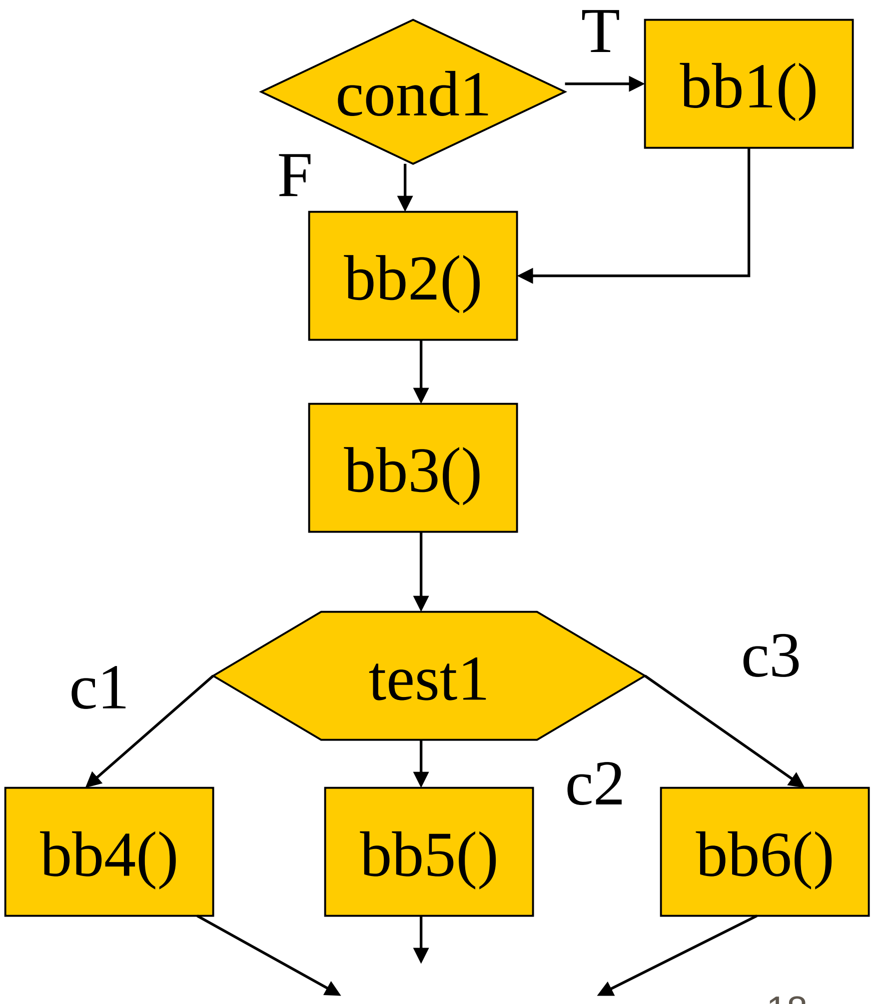
Control



Equivalent forms

CDFG example

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



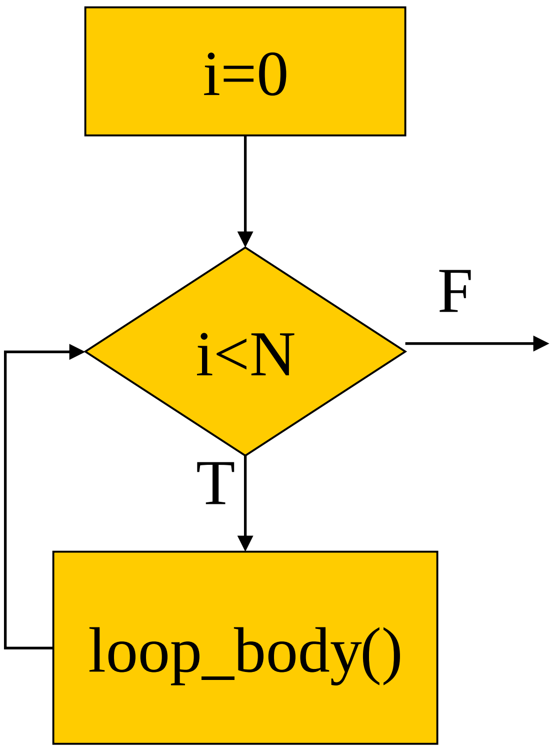
for loop

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

for loop

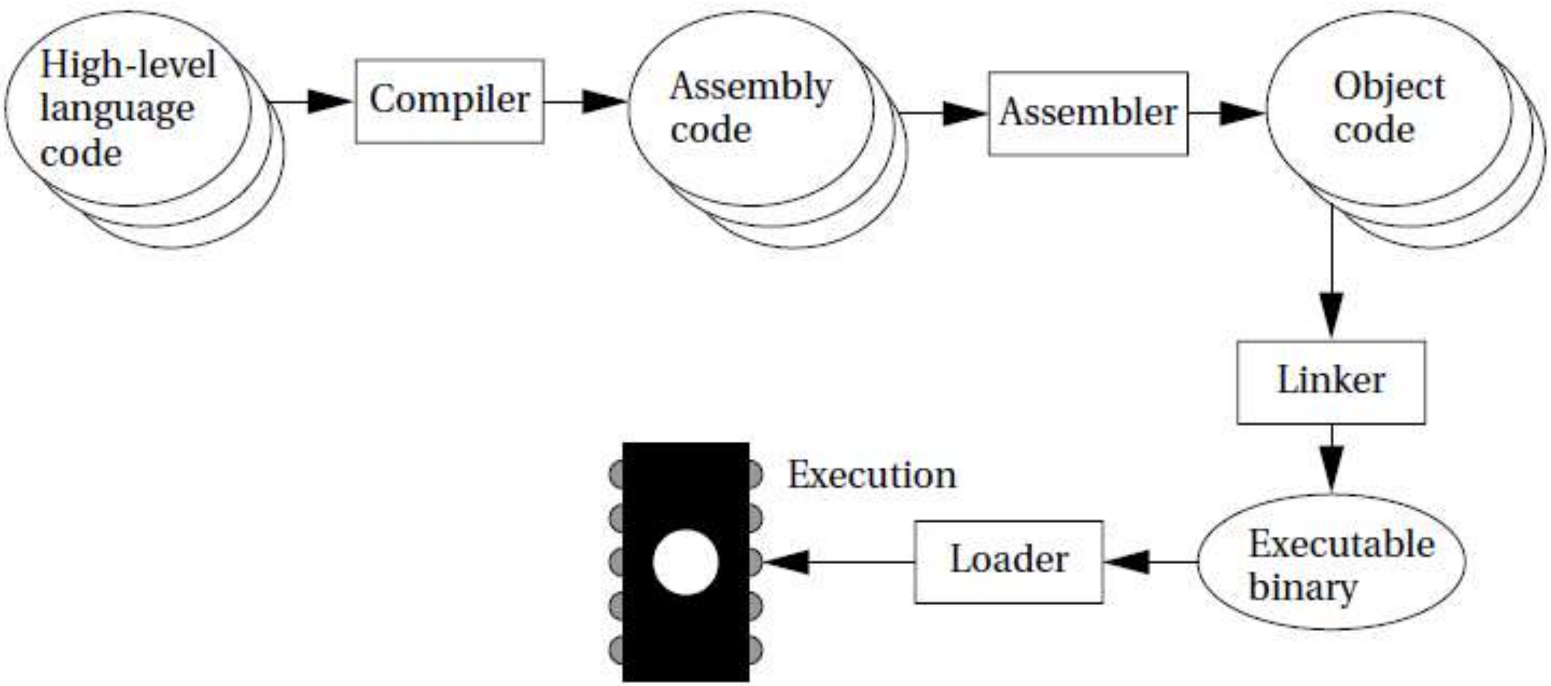
```
i=0;
while (i<N) {
    loop_body(); i++; }
```

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	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=0x7
PLC=0x7
PLC=0x7

```
DD r0,r1,r2
DD r3,r4,r5
MP r0,r3
yy   SUB r5,r6,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.

Linking

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

Externals and entry points

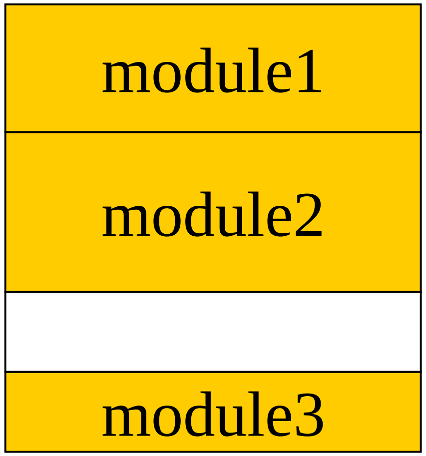
entry point

```
xxx   ADD r1,r2,r3          a    ADR r4,yyy
      B a                        ADD r3,r4,r5
yyy   %1
```

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.

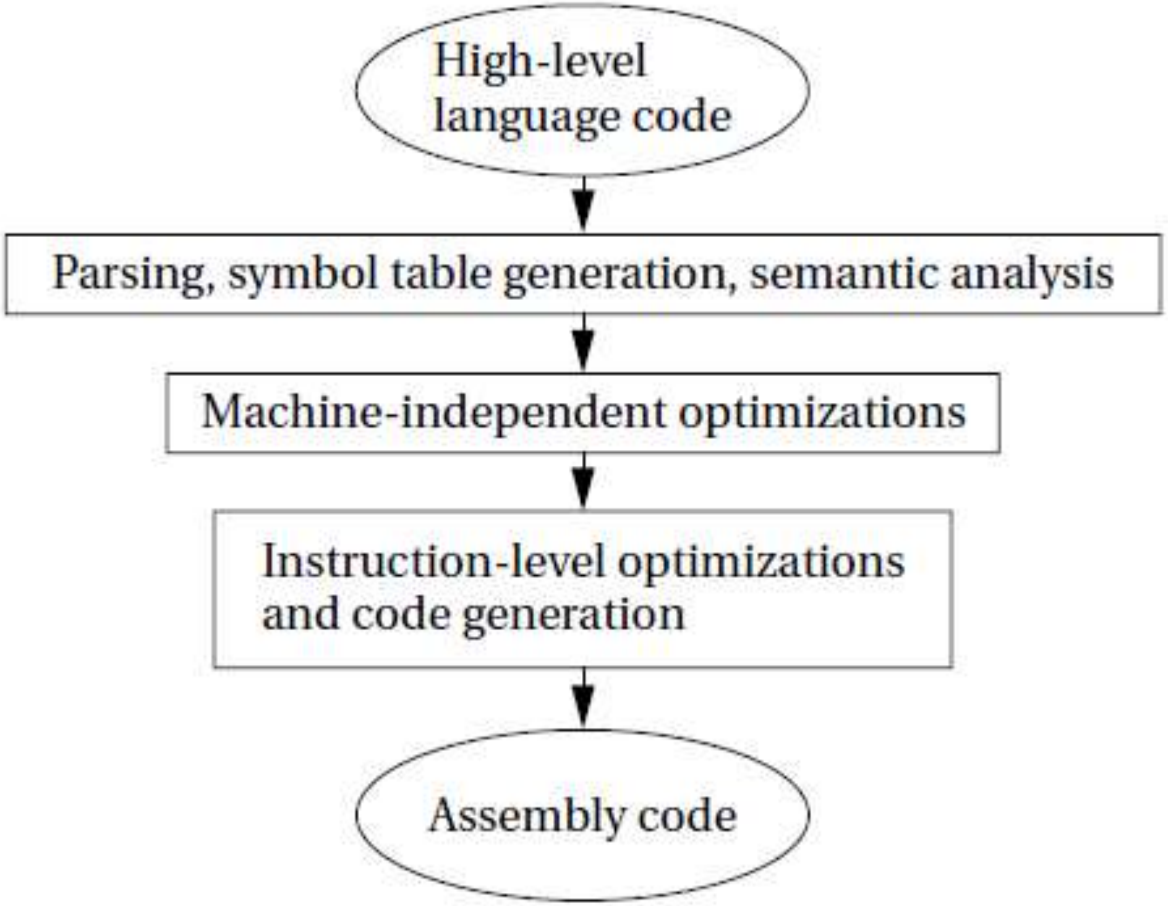
Program design and analysis

- 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



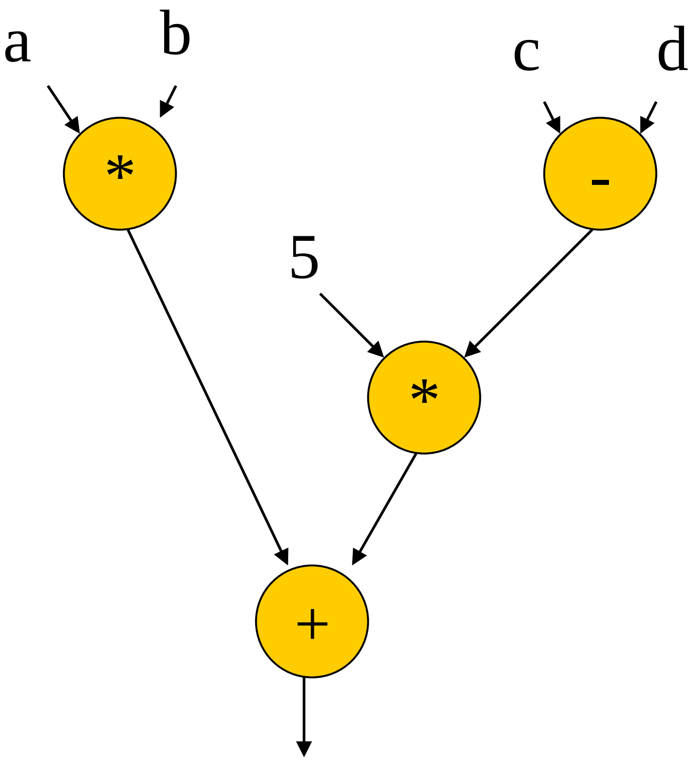
Statement translation and optimization

- ⌘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

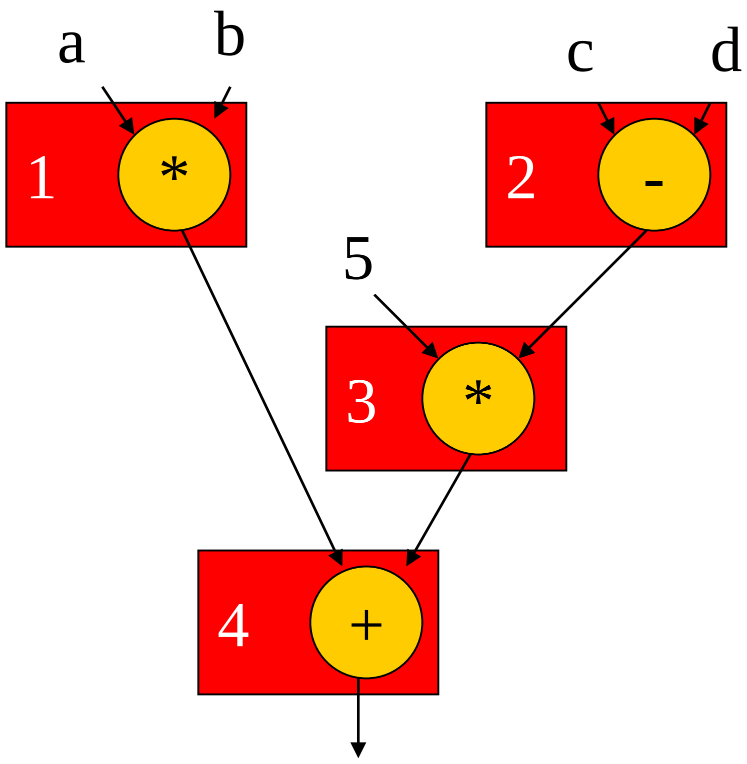
a*b + 5*(c-d)

expression



DFG

Arithmetic expressions, cont'd.

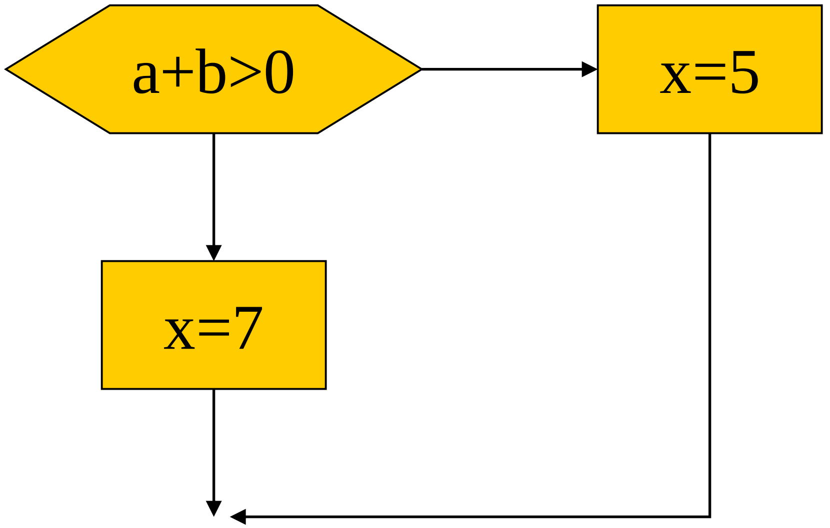



```
ADR r4,a
MOV r1,[r4]
ADR r4,b
MOV r2,[r4]
ADD r3,r1,r2
ADR r4,c
MOV r1,[r4]
ADR r4,d
MOV r5,[r4]
SUB r6,r4,r5
MUL r7,r6,#5
ADD r8,r7,r3
```

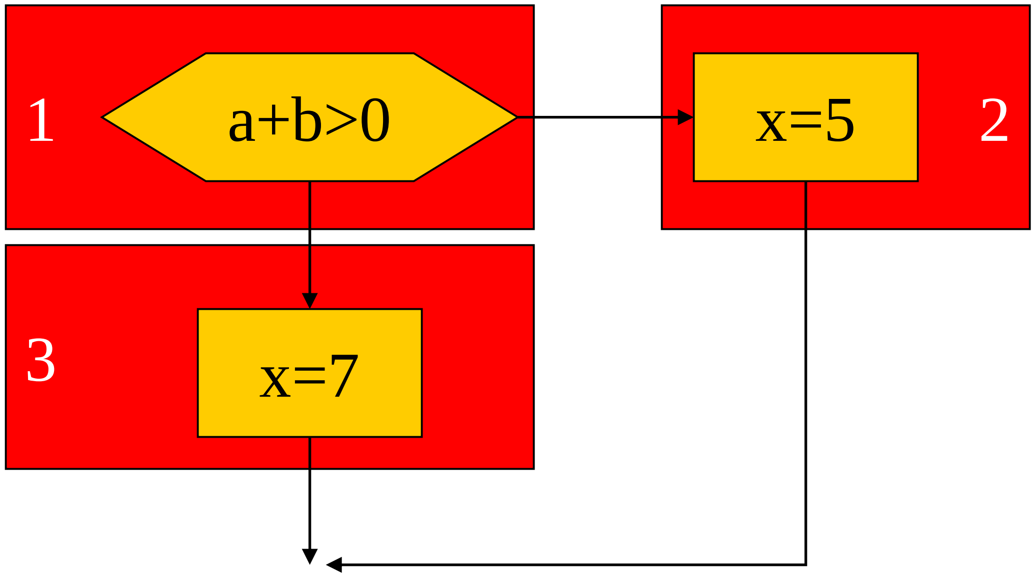
DFG code

Control code generation

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



Control code generation, cont'd.

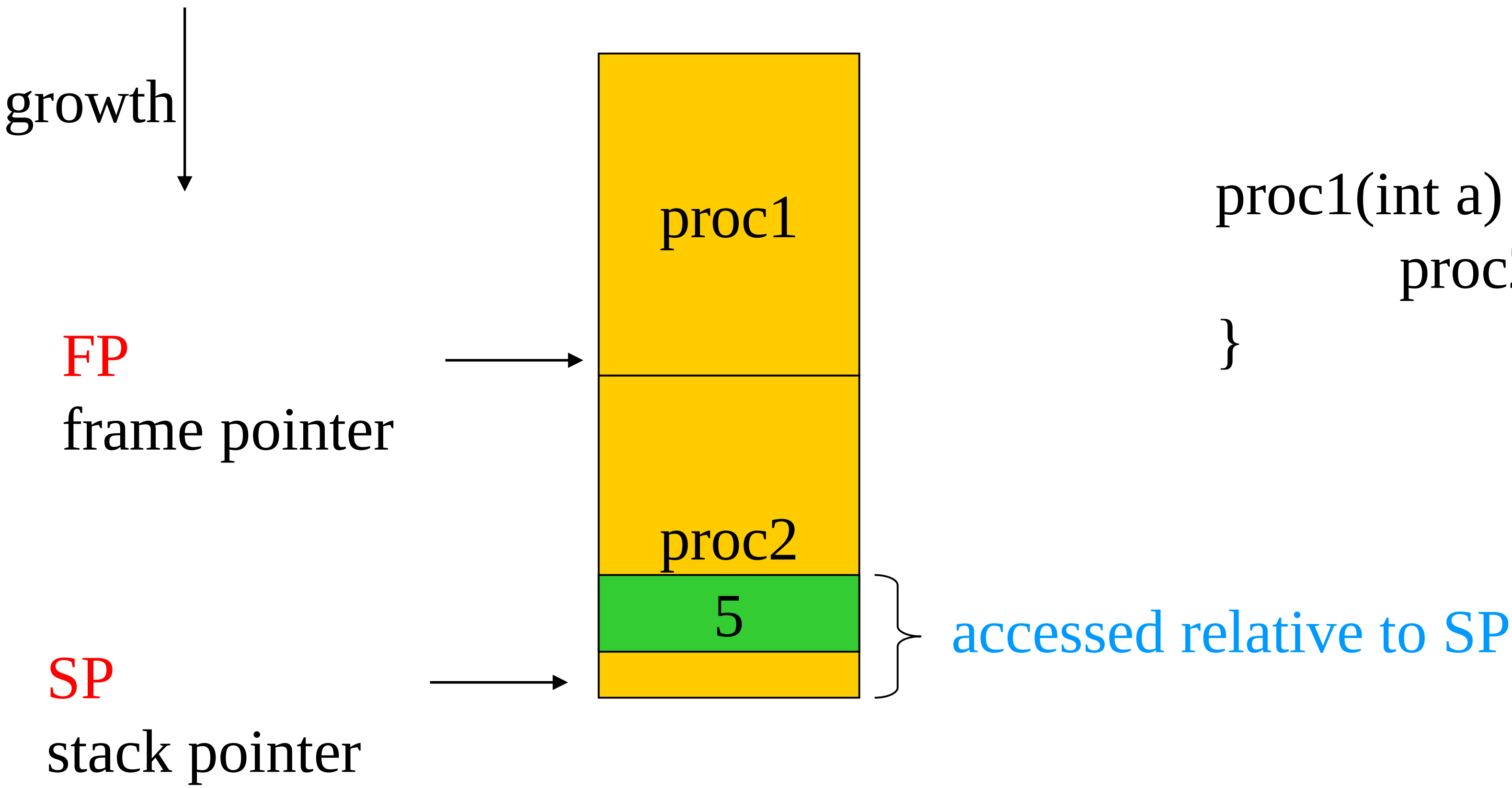



```
ADR r5,a
LDR r1,[r5]
ADR r5,b
LDR r2,b
ADD r3,r1,r2
BLE label3
 LDR r3,#5
 ADR r5,x
 STR r3,[r5]
 B stmtent
 LDR r3,#7
 ADR r5,x
 STR r3,[r5]
stmtent ...
```

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



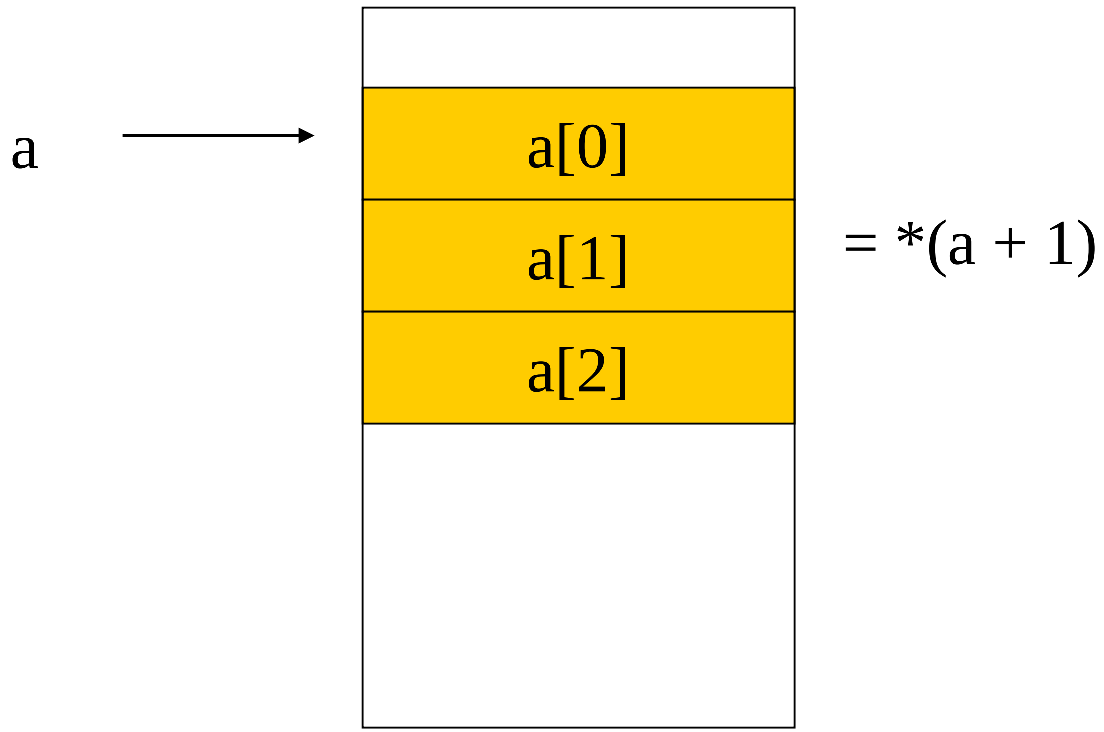

```
proc1(int a) {
        proc2(5);
}
```

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.

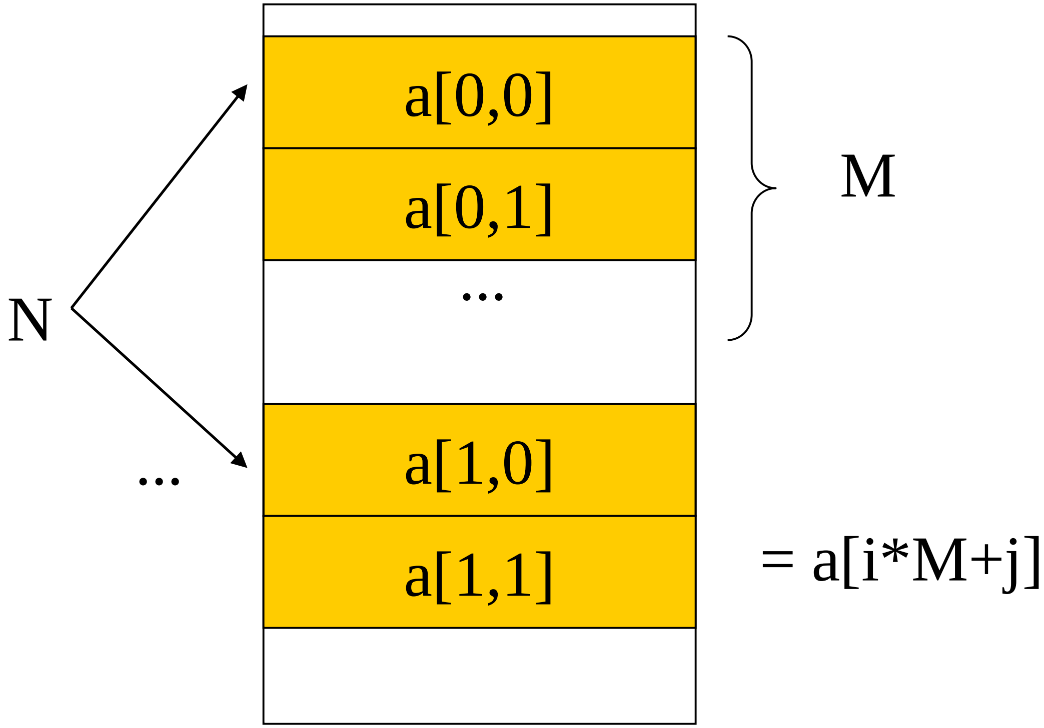
One-dimensional arrays

- C array name points to 0th element:



Two-dimensional arrays

- Column-major layout:

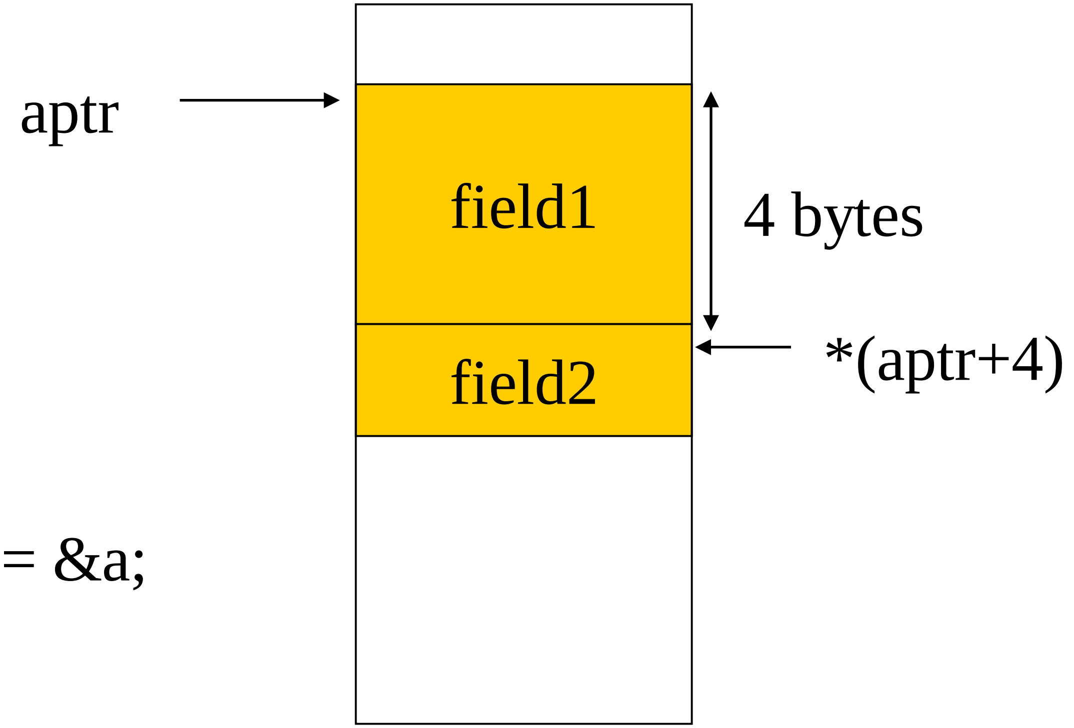


Structures

- Fields within structures are static offsets:

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

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



Expression simplification

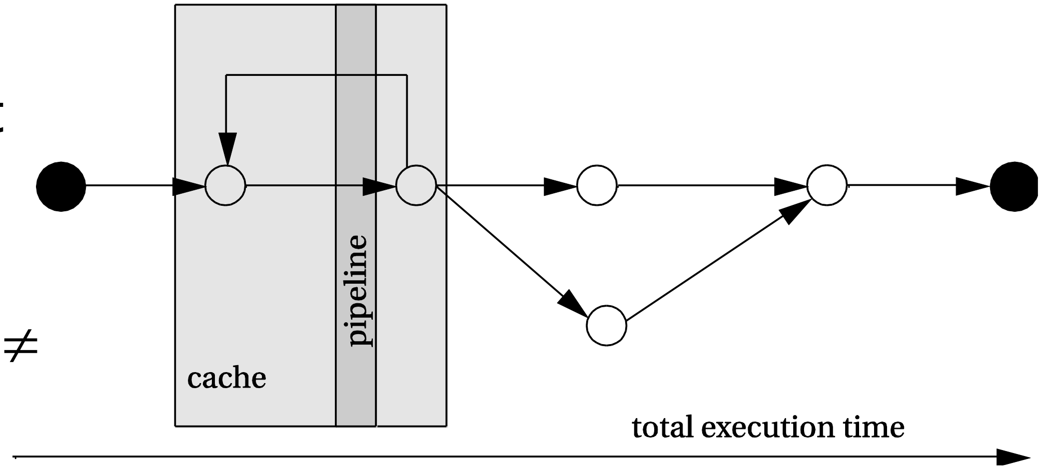
- Constant folding:
  - 8+1 = 9
- Algebraic:
  - a*b + a*c = a*(b+c)
- Strength reduction:
  - a*2 = a<<1

Program design and analysis

- 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.

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.

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.

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.

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.

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.

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

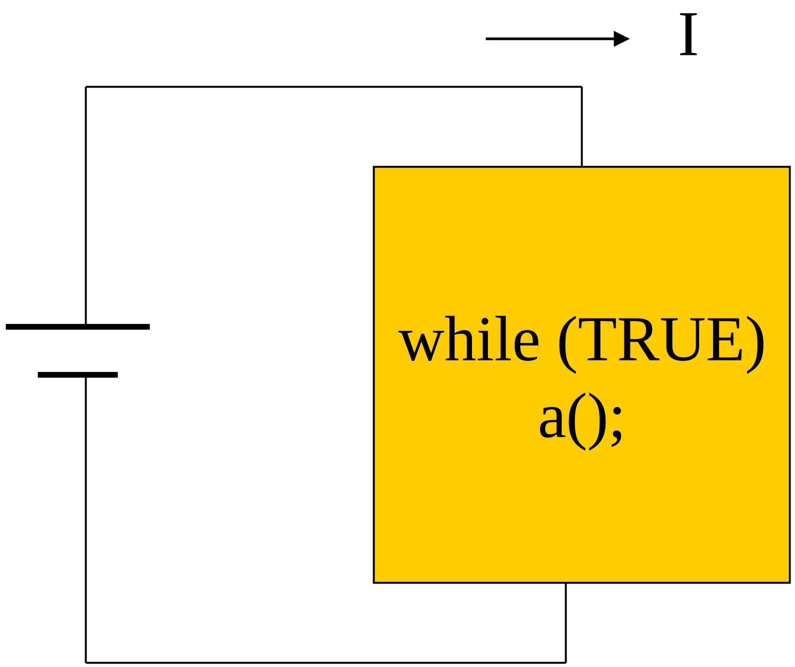
- General rules:
  - Don’t use function calls.
  - Keep loop body small to enable local repeat (only forward branches).
  - Use unsigned integer for loop counter.
  - Use <= to test loop counter.
  - Make use of compiler---global optimization, software pipelining.

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.

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.

  (Generate tests without looking at the internal structure of the program)

  - Clear box (white box) does look at the source code(Generate tests based on the program).

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.

Execution path

- The most fundamental concept in clear-box testing is the path of execution through a program

- Is it possible to execute every complete path in an arbitrary program? The answer is no, because the program may contain a while loop that is not guaranteed to terminate
- simple measure, **Cyclomatic complexity** [McC76], allows us to measure the control complexity of a program.
- A simple condition testing strategy is known as **branch testing** [Mye79].
- This strategy requires the true and false branches of a conditional and every simple condition in the conditional's expression to be tested at least once.
- Another testing strategy known as **data flow testing** makes use of **def-use analysis** (short for definition-use analysis). It selects paths that have some relationship to the program's function.
- The terms def and use come from compilers, which use def-use analysis for optimization [Aho06]. A variable's value is **defined** when an assignment is made to the variable; it is **used** when it appears on the right side of an assignment (sometimes called a **C-use** for computation use) or in a conditional expression (sometimes called **P-use** for predicate use). A **def-use pair** is a definition of a variable's value and a use of that value.

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.
  - Random values are generated with a given distribution. The expected values are computed independently of the system, and then the test inputs are applied. A large number of tests must be applied for the results to be statistically significant, but the tests are easy to generate.
- 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.