NTNU
Norwegian University of
Science and Technology

Lecture 7: Program Design and Analysis

Asbjørn Djupdal
ARM Norway, IDI NTNU
2013

Lecture overview

- Models of programs
- Compiler toolchain
  - Precompiler
  - Compiler and compilation techniques
  - Assembler
  - Linker
- Optimizing execution time
- Energy aware programming

Models of programs

- Represent the program in a form easy to analyze
- Source code
  - Too hard to manipulate
  - Language specific
- Need a model independent of language
- A compiler uses a temporary representation (IR: Intermediate Representation) for manipulating programs
- Typical data structure: graph

Data flow graph (DFG)

- Models a *basic block*
- A basic block:
  - Code with one start point and one end point where there are no conditional branches
- Gives a partial order of operations in a directed acyclic graph (DAG)
  - Partial because some operations do not depend on each other and can be executed in any order

Single assignment form

- Necessary for creating DFG for a basic block
- Requirement: Each variable is written to only once

x = a + b;	x = a + b;
y = c - d;	y = c - d;
z = x * y;	z = x * y;
y = b + d;	**y1** = b + d;
original basic block	*single assignment form*

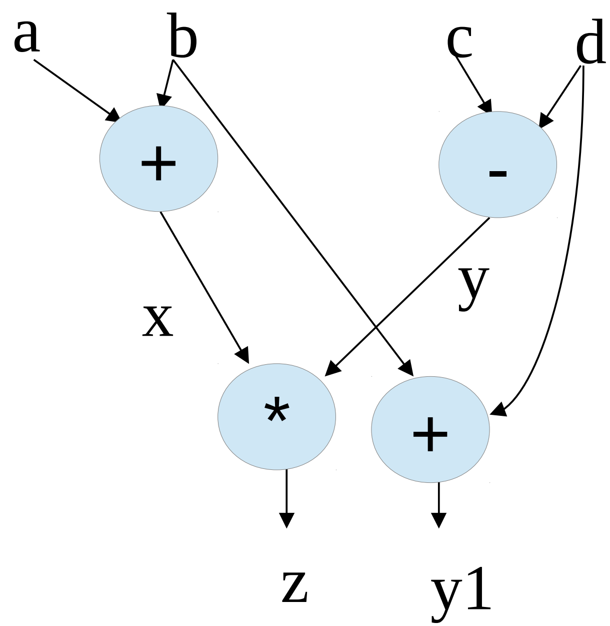
Data flow graph (DFG)

x = a + b;

y = c - d;

z = x * y;

y1 = b + d;



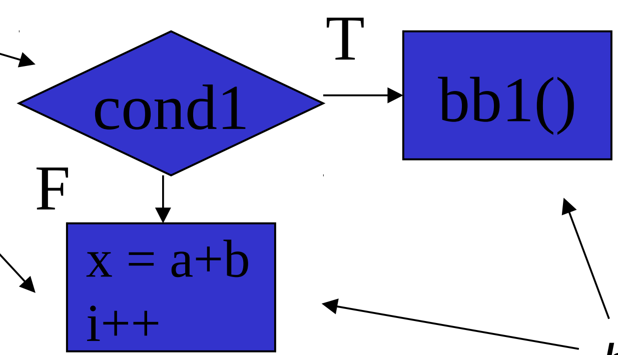
Basic block in single assignment form

Partial orders:

1. a+b and c-d
2. x*y and b+d

Control data flow graph (CDFG)

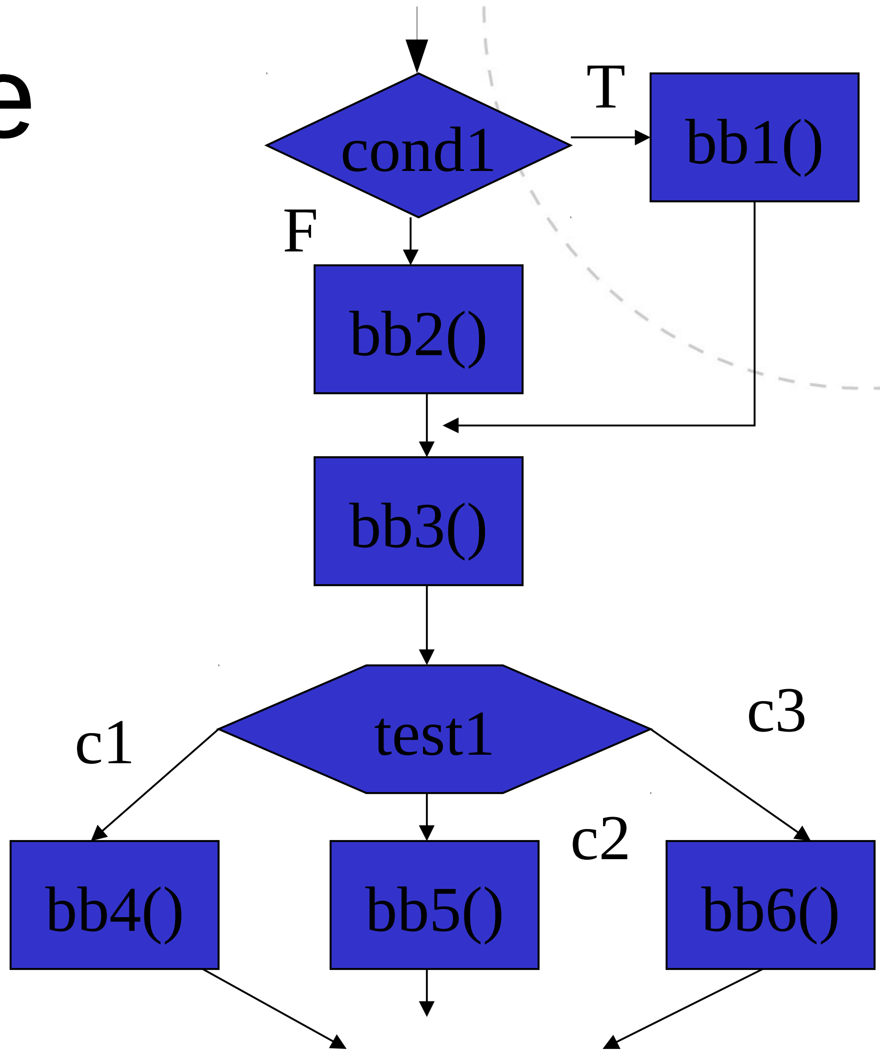
- CDFG: Represents both control and data
- Uses DFG as components
- Two node types:
  - Decision nodes
  - Data flow nodes



basic blocks

CDFG Example

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



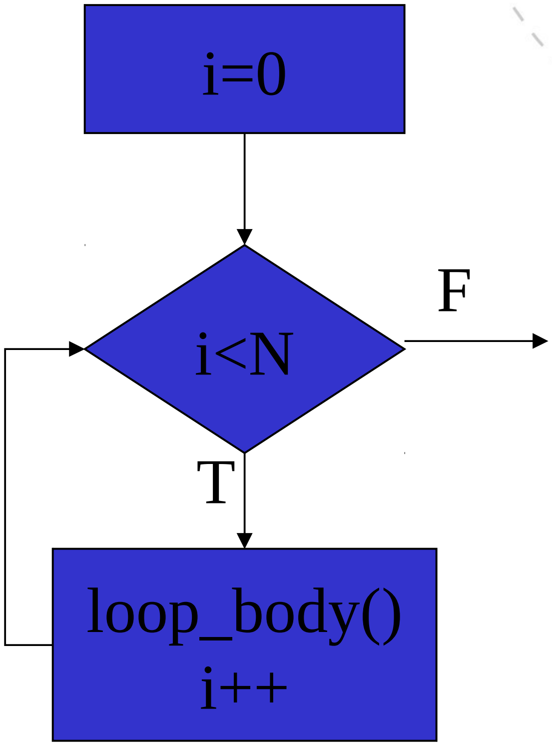
Loops

for-loop:

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

Equivalent while-loop:

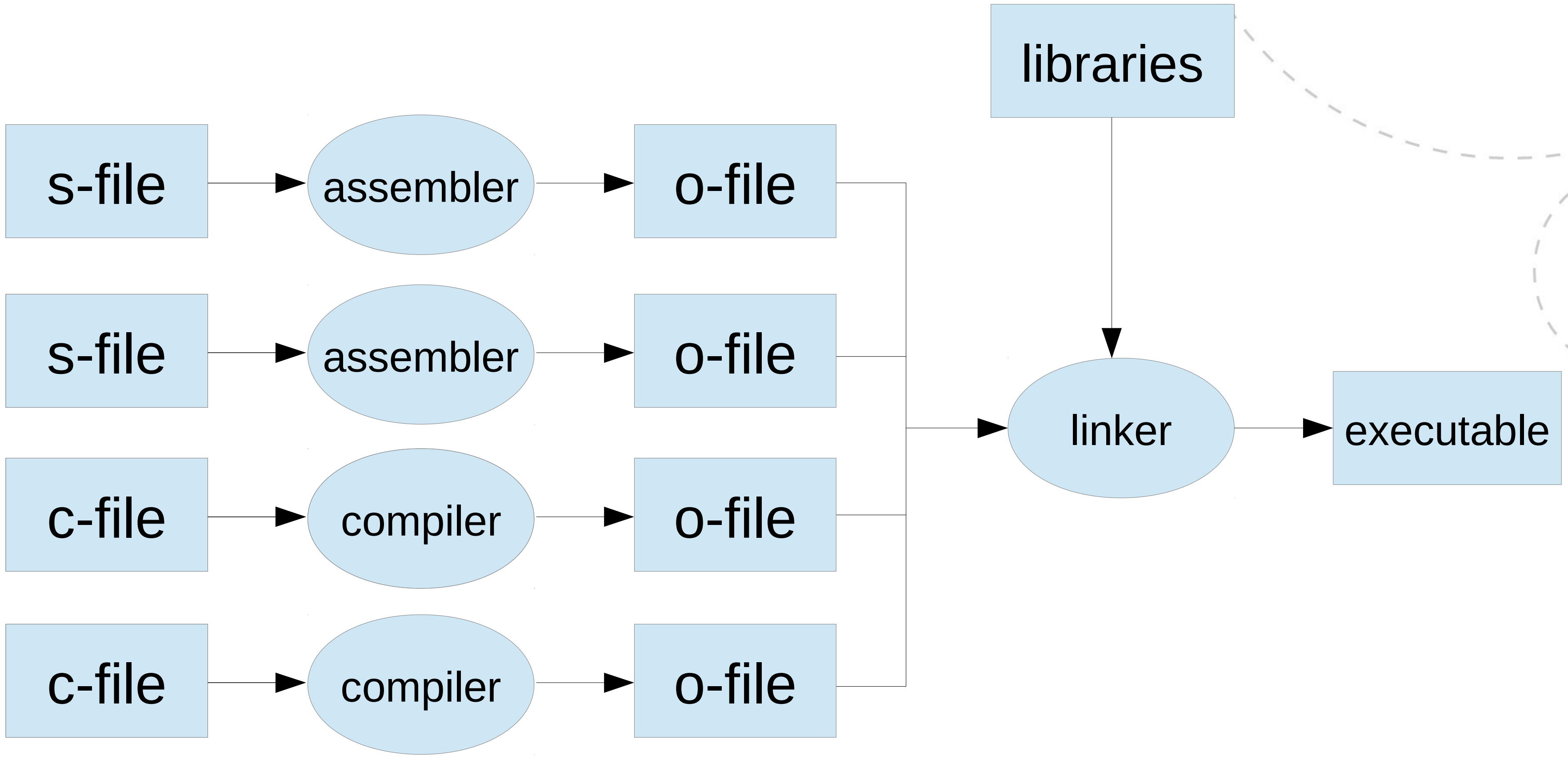
```
i=0;
while (i<N) {
  loop_body();
  i++;
}
```



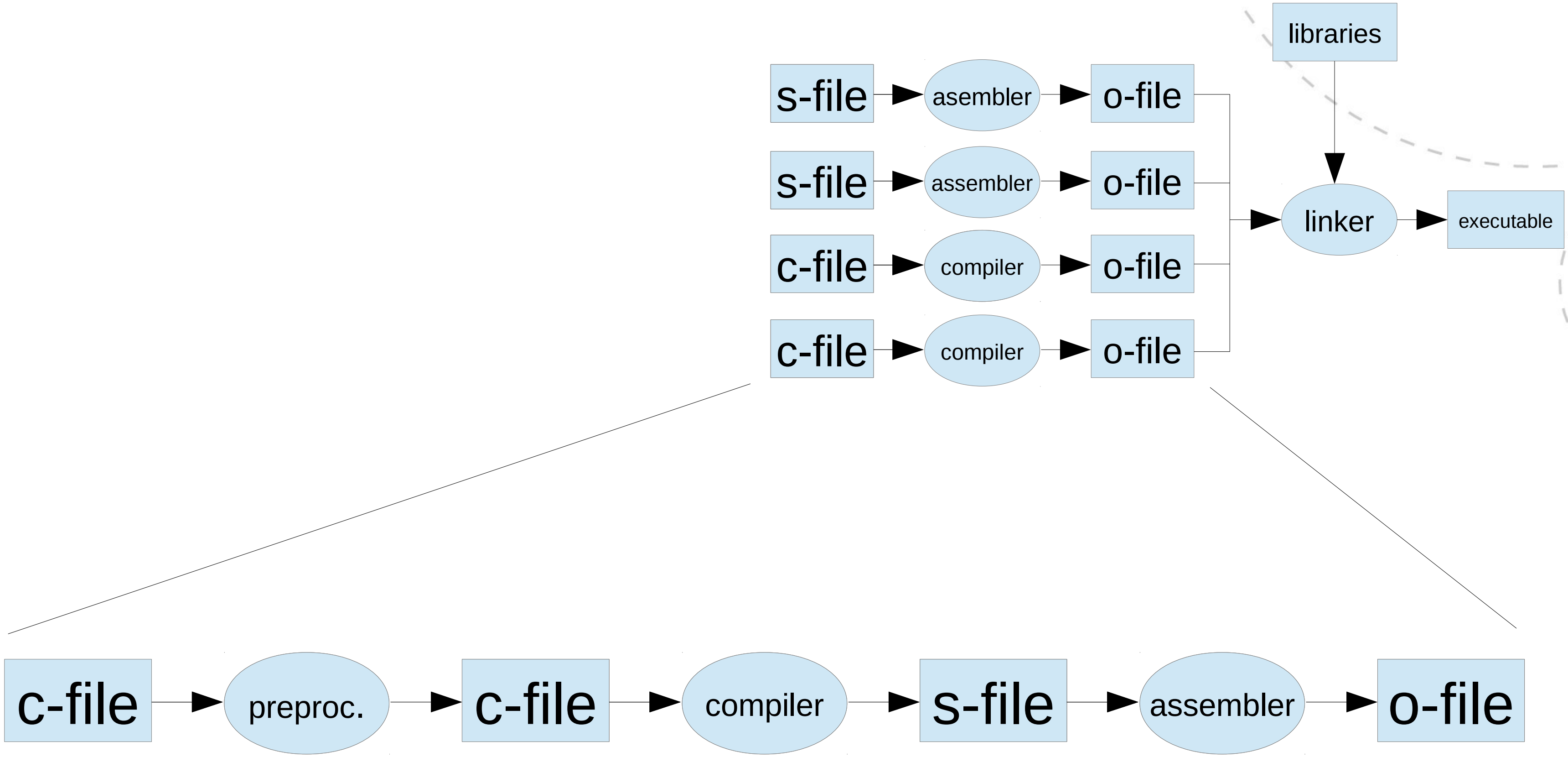
Lecture overview

- Models of programs
- Compiler toolchain
  - Precompiler
  - Compiler and compilation techniques
  - Assembler
  - Linker
- Optimizing execution time
- Energy aware programming

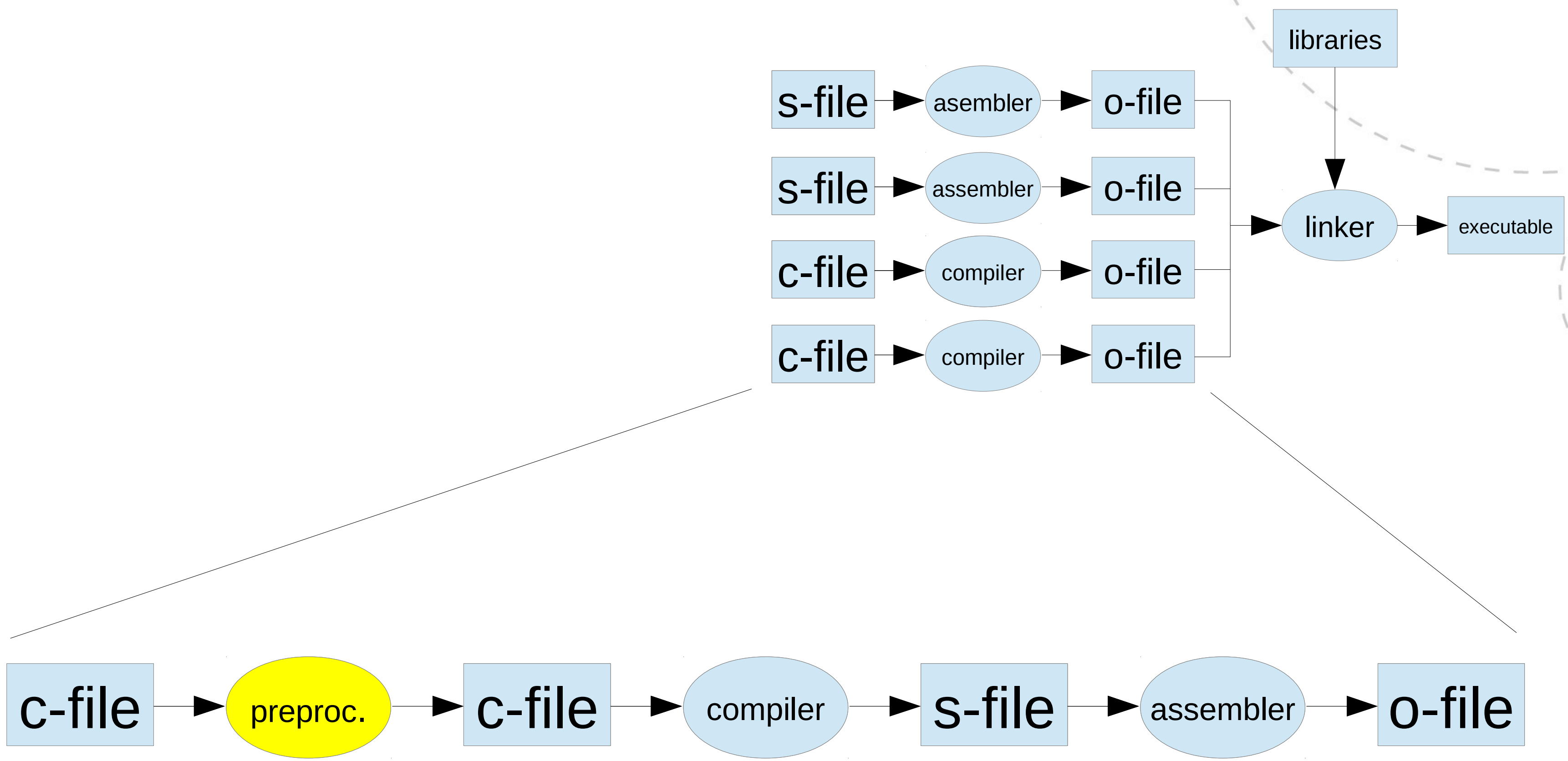
Compiler tool chain



Detailed C tool flow



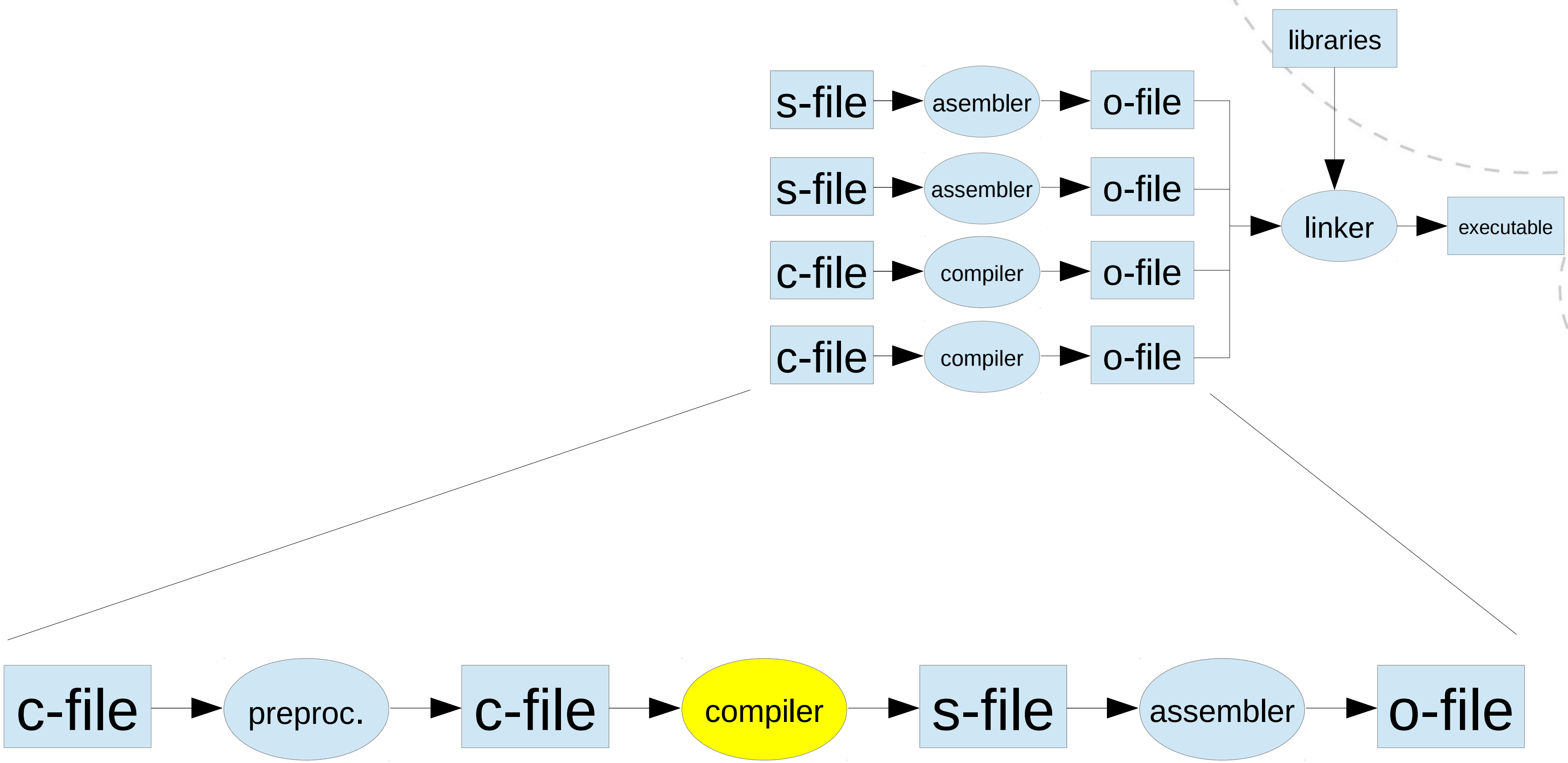
C preprocessor



C preprocessor

- Transforms C source file with macro substitutions
  - Result is another C source file
- Example: #define CONST 5
  - All occurances of CONST in source will be substituted with 5
    - Ny type checking or anything, just a direct textual replacement
  - Different from final in java, the following is not equivalent:
    - #define CONST 5+3
    - #define CONST (5+3)
- Example: #include “file.h”
  - The entire file.h will be inserted in the C source file
- To examine output from the GCC preprocessor:
  - gcc -E file.c -o output.c

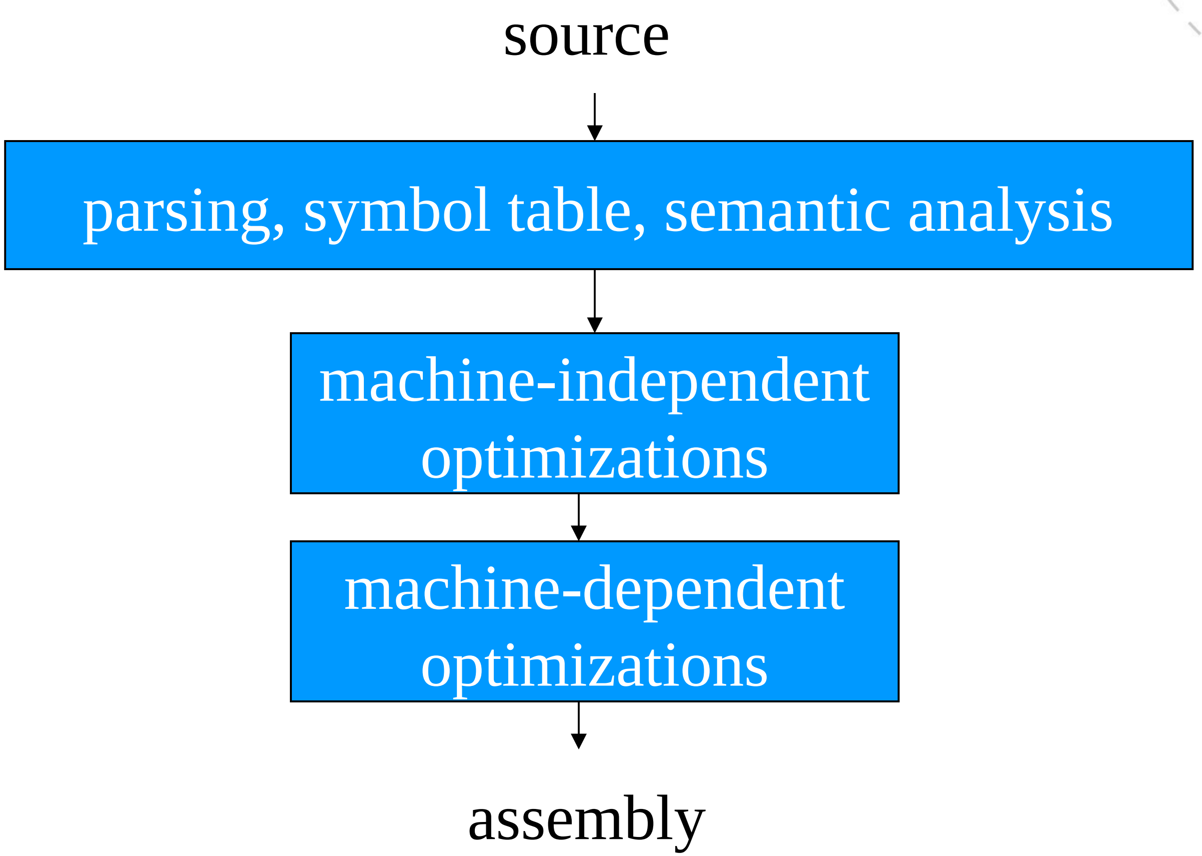
Compilation



Compilation

- Important to understand, especially for embedded systems
  - Intimately related to HW
  - Fewer abstractions (sometimes no OS)
  - Performance, power, energy, real time requirements
  - Which problems can arise during compilations?
  - When is it necessary to write assembly code manually?
- Compiler strategy:
  - Compilation = translation + optimization
- Compiler has a large impact on the resulting program
  - Which and how many instructions are used?
  - How efficient are registers and cache utilized?
  - Are the available resources used?
  - How are memory and I/O operations scheduled?
  - Code size

Compiler phases



Translation and optimization of expressions

- Source code is translated to a temporary CDFG representation
- The CDFG is transformed and optimized
- The CDFG is translated to instructions
- The instructions are further optimized for the target platform

Procedure calls and control flow

- Procedure calls
  - Realized using the processor subroutine call convention
  - Lecture 2 explained this in detail for ARM processors
    - Use the stack to hold local variables and state (saved registers)
    - Pass arguments and return values either through registers or the stack
- Lecture 2 also showed HLL to assembly examples for control flow and expressions
  - Not repeated here

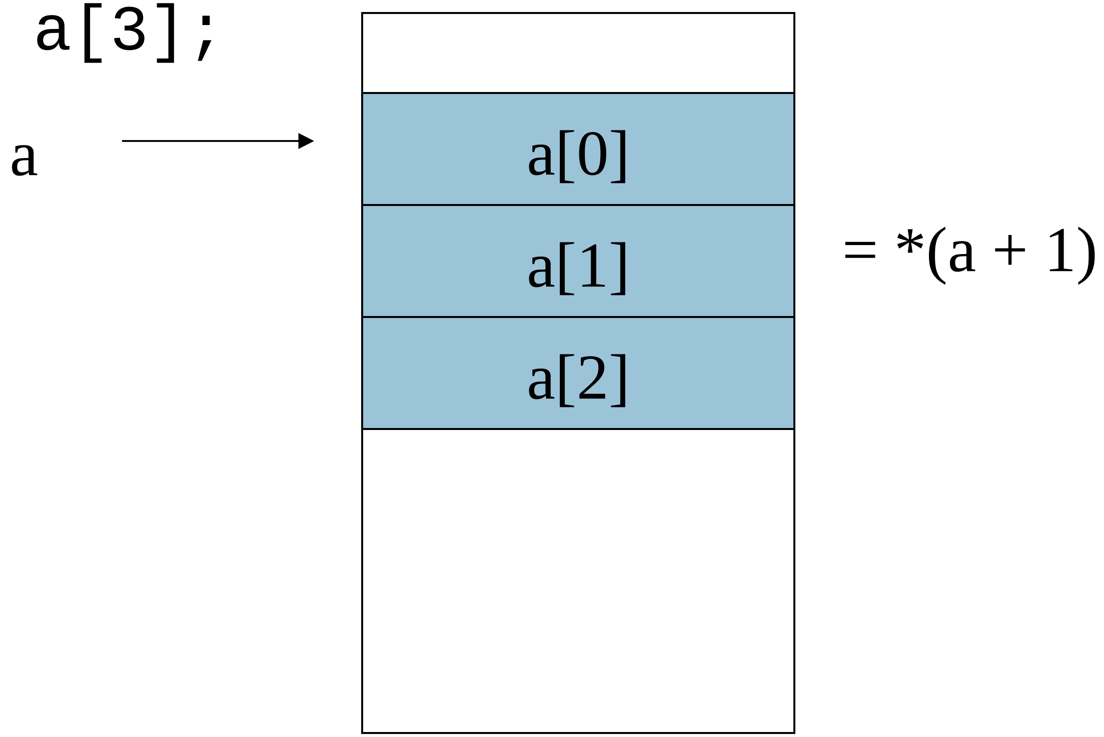
Data structures

- HLL data structures must be realized in memory
  - Integers, floating point, pointers, tables, arrays, trees, graphs, objects, etc
- Typically realized as a base address pointing to a structure with properties known at compile time
- Basic datatypes in C:
  - int, float, double, ...
  - Undefined length
    - Use C99 and include stdint.h and stdbool.h to get proper datatypes
      - bool, uint32_t, int16_t, ...

Array

- A C array is a pointer to the first element

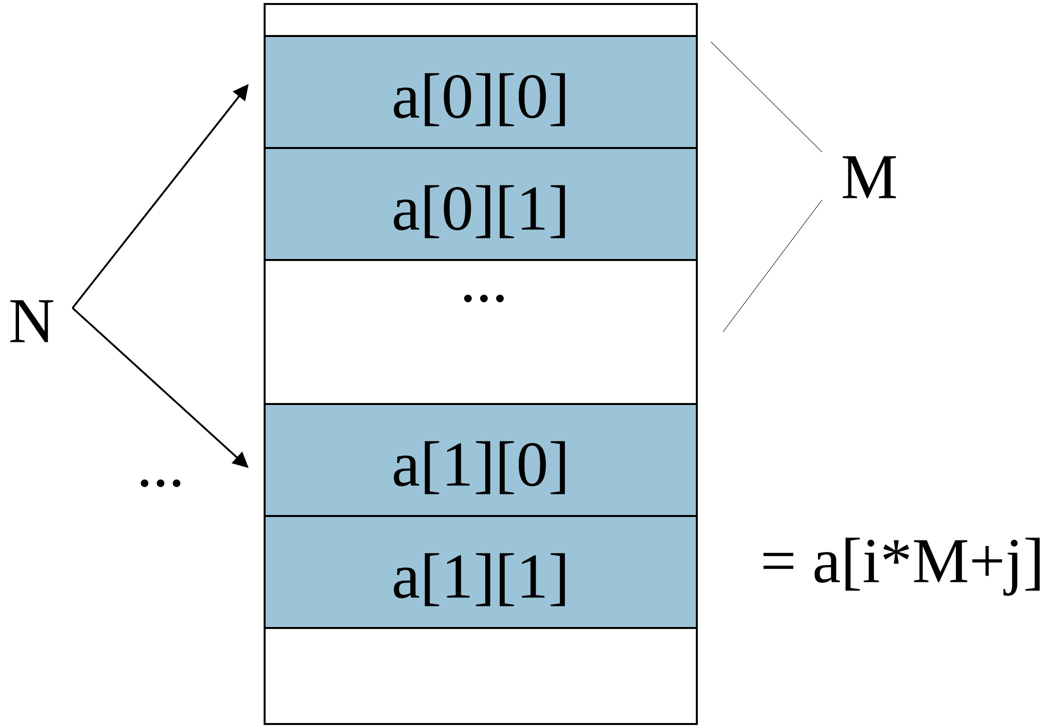
```
int a[3];
```



2D array

- Row-major layout in C
- You need to know the size of each row (M) to compute the offset

```
int a[N][M];
```



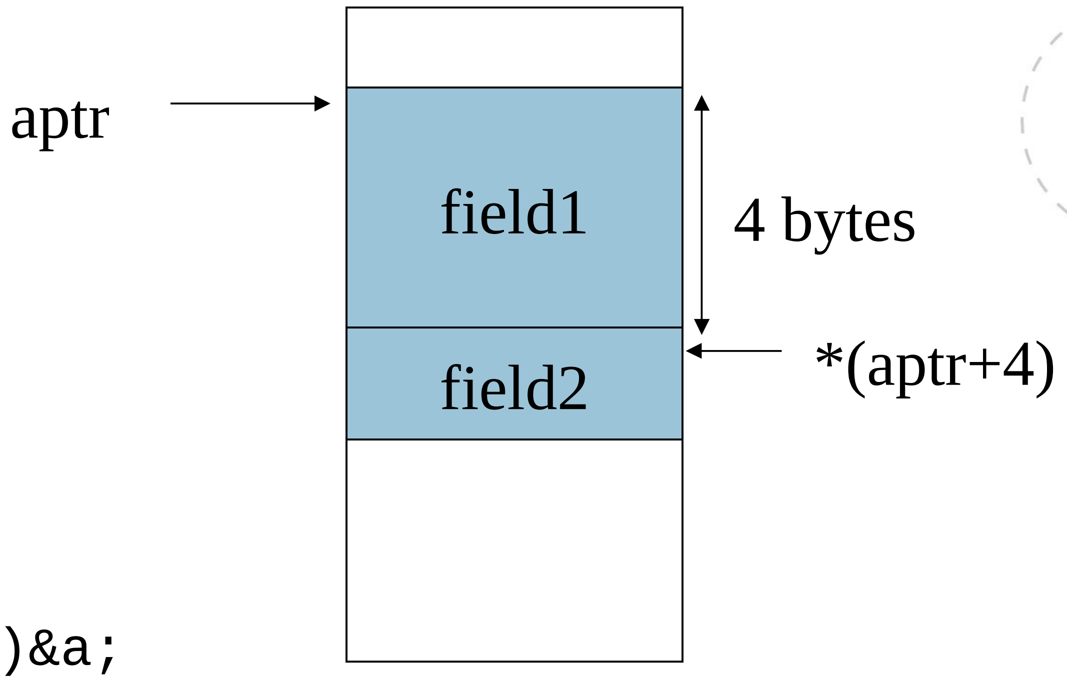
C struct

- The fields in a struct are static offsets

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

struct mystruct a;

char *aptr = (char*)&a;
```



Warning: alignment

Simplifying expressions

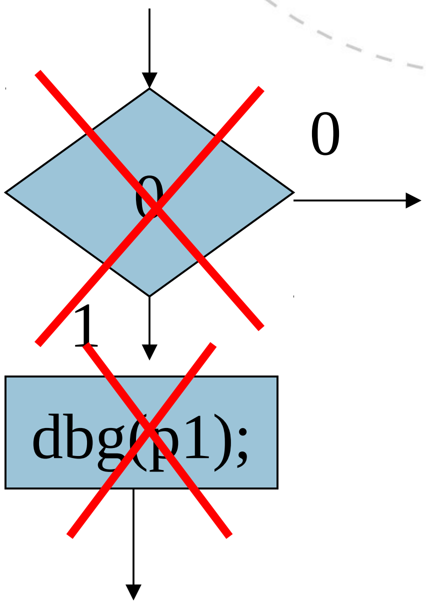
- Constant folding
  - 8 / 2 = 4
  - int portaddr = PORT_A + PORT_OFFSET
- Algebraic
  - a * b + a * c ⟶ a * (b + c)
- Strength reduction:
  - a * 2 ⟶ a << 1;

Eliminating dead code

- Dead code:

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

- Can be eliminated by analyzing the CDFG
  - Reachability analysis
- Reduces code size and may enable additional optimizations



Procedure inlining

- Save procedure call overhead

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

z = w + x – y;
```

- C99 supports keword `inline`
  - Can be rejected
  - Optimizer can inline even without keyword
    - But only if the function code is available when compiling
- Can increase code size
- Can reduce cache hit rate

Loop transformations

- A lot of run time is used in loops
- Goals:
  - Reduce loop overhead
  - Increase possibilities for optimizations for pipelines and parallelism
  - Improved use of the memory system

Loop unrolling

- Reduce number of iterations in a loop
- Reduces loop overhead and can enable other optimizations
  - Efficient branch prediction HW can reduce usefulness unless other optimizations are enabled
    - E.g. reorder instructions from different iterations to eliminate pipeline bubbles

```
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];
}
```

↓

```
a[0] = b[0]*c[0];
a[1] = b[1]*c[1];
a[2] = b[2]*c[2];
a[3] = b[3]*c[3];
```

Loop fusion

- Combine two loops into one
  - Requires similar iteration space and no dependencies between loops

```
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 fission

- Split one loop into two
  - Cache: Principle of locality
  - (Multicore)

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

↓

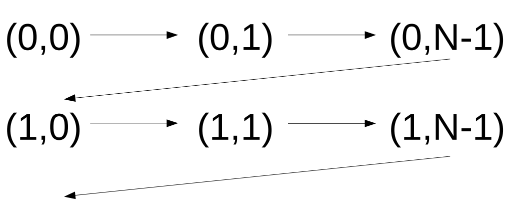
```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
```

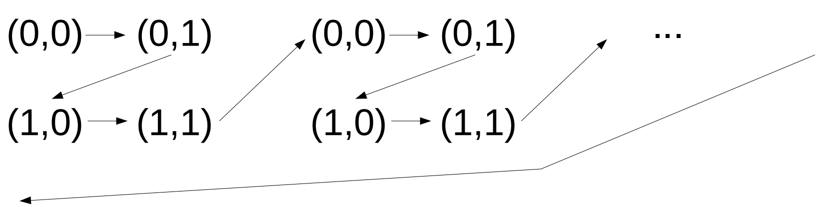
Loop tiling

- Break a loop into a set of nested loops
  - Each inner loop performs operations on a subset of the data
- Changes the memory access pattern
- Can improve cache behaviour

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    f(i,j);
```

```
for(i = 0; i < N; i += 2)
  for(j = 0; j < N; j+= 2)
    for(ii = i; ii < min(i + 2, N); i++)
      for(jj = j; jj < min(j + 2, N); j++)
        f(ii,jj);
```





Array padding and struct alignment

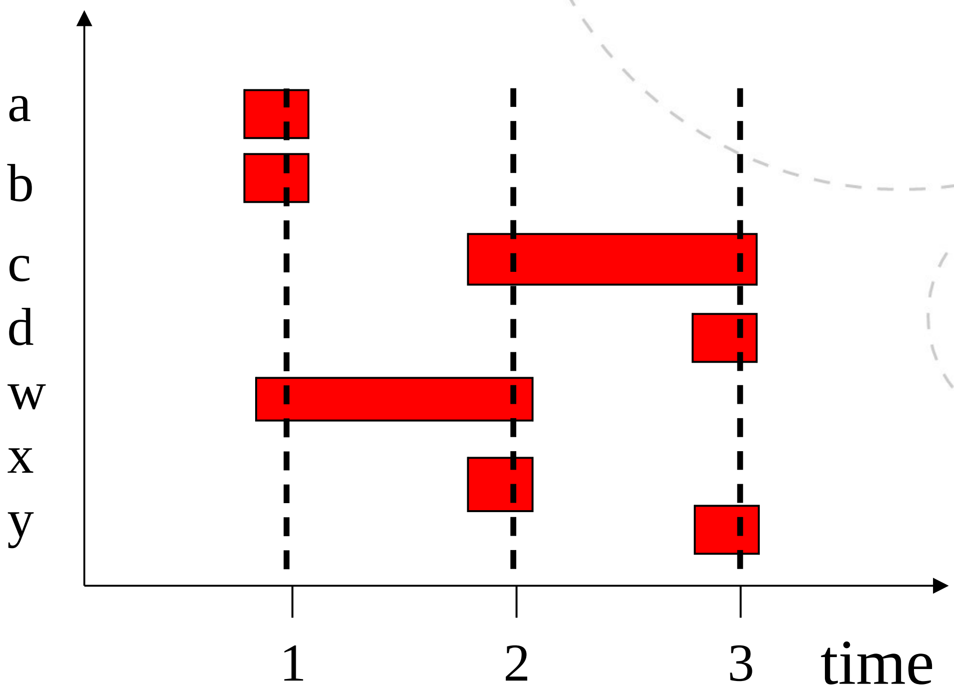
- Arrays can be padded with dummy data to better fit cache lines
- Struct elements can be aligned (or padded) to make access more efficient
  - Can also be a requirement by the CPU architecture
- There exists (non-standard) attributes that instructs compiler to avoid aligning and padding

Register allocation

- We need registers to temporarily hold variables
- Choose registers such that memory accesses are minimized
- Analyze the lifetime of variables
- C supports the `register` keyword
  - Hint from the programmer that a variable should be held in a register
  - Usually just ignored by the compiler, rarely used in modern code
- When a C variable *must* be available in memory
  - C keyword `volatile`
  - Still transferred to register, but immediately written back to memory when changed

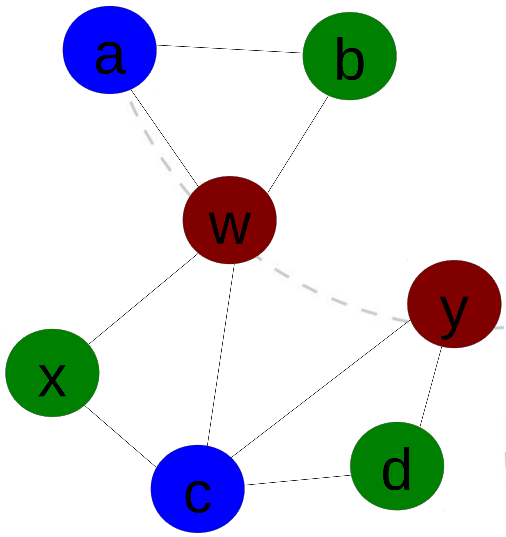
Registers, lifetime graph

w = a + b; t=1
x = c + w; t=2
y = c + d; t=3



Register allocation

- Method: Conflict graph and graph coloring
  - Edge between variables that are alive at the same time
  - Represent each register with a color
  - Color the nodes with as few colors as possible
    - No edge must share a color
  - NP-complete
    - Compilers use heuristics to find a good solution



3 registers

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

Operator scheduling

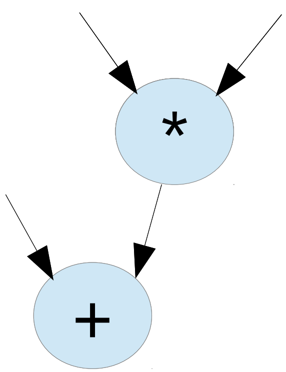
- Can change the order of instructions to reduce number of registers
  - Changes the life time graph

w = a + b;	w = a + b;
x = c + d;	z = a – b;
y = x + e;	x = c + d;
z = a - b;	y = x + e;

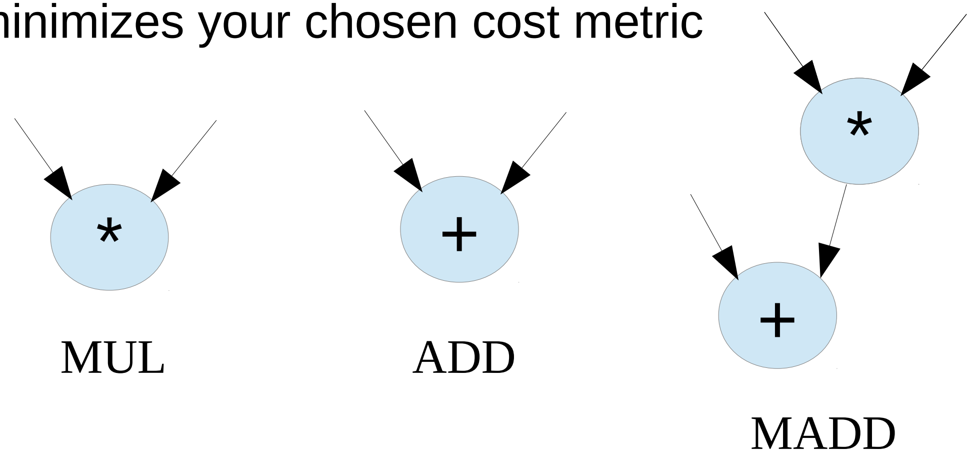
From 5 registers to 3

Choosing instructions

- A number of instruction sequences can implement a piece of HLL code
- The expressions are represented as a graph
  - Find the best template for the expression
  - Find the mapping that minimizes your chosen cost metric

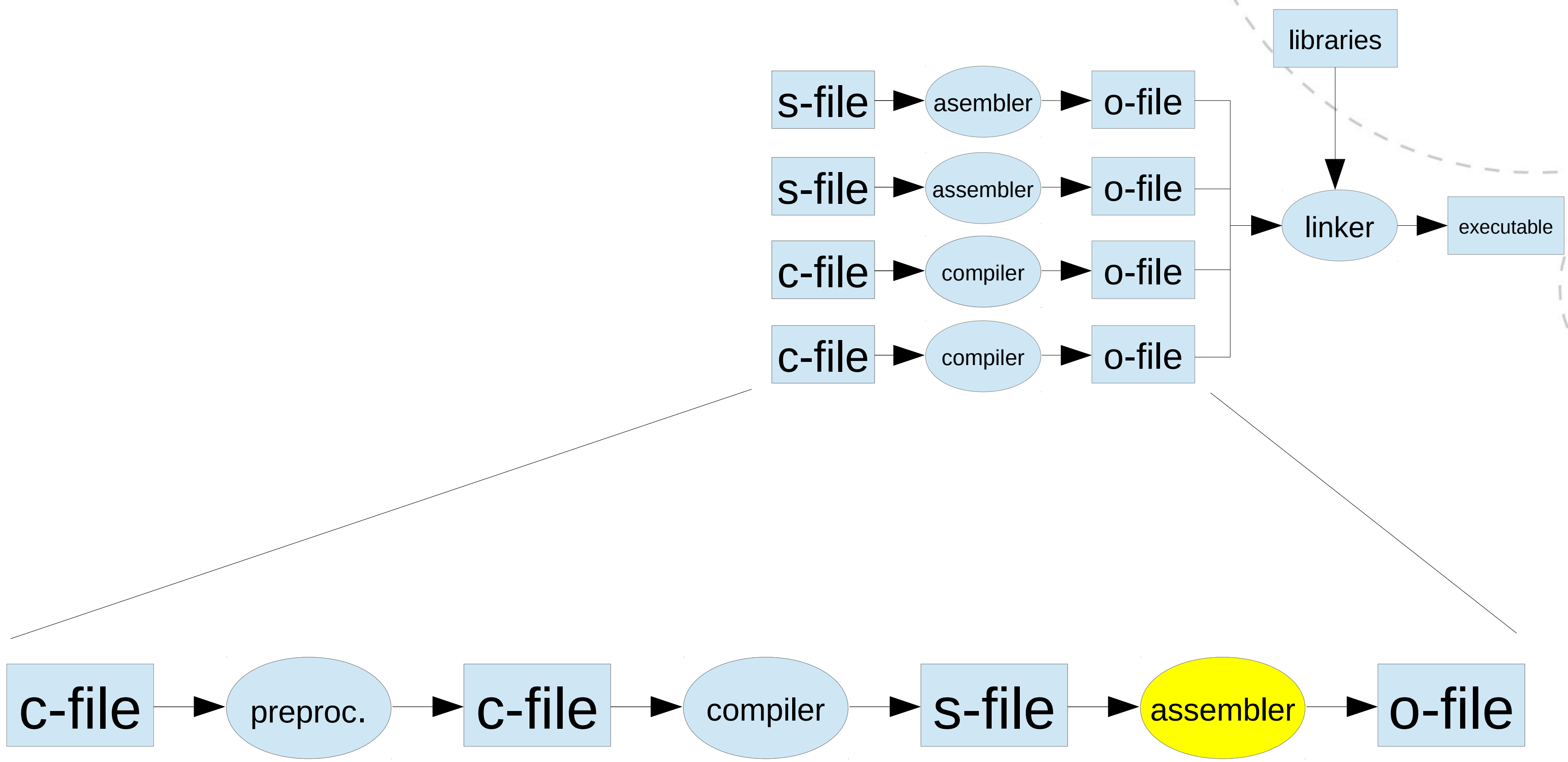


Expression



Templates

Assembly


libraries
s-file
asembler
o-file
s-file
assembler
o-file
c-file
compiler
o-file
c-file
compiler
o-file
linker
executable
c-file
preproc.
c-file
compiler
s-file
assembler
o-file

Assembly

- Lecture 2 introduced assembly for ARM
- Goals:
  - Generate a binary representation of instructions from a symbolic representation

    `ADD R3, R2, R3` → `0xE0823003`
  - Generate symbol table
    - Remember, symbols are:
      - Labels
      - Constants
  - Handle pseudo operations (.word, .data, etc)
- Many assemblers use two passes
  - 1 pass: Generate symbol table
  - 2 pass: Assemble instructions

Symbol table

- Symbols in the assembly file are inserted into the object file *symbol table*.
  - Can then look in the symbol table when assembling instructions refering to symbols
  - Might need to declare in the assembly file which symbols are to be visible to other object files
- Can not always resolve the value of symbols while assembling
  - The start address of the assembly file is often unknown
  - Often necessary to refer to symbols in other object files
- All references to unresolvable symbols are marked as unresolved.
  - Placeholders are inserted where instructions refer to unresolved symbols
  - Must be resolved later, either by the *linker* or the *loader*

Generating the symbol table

```
CONST = 5

        ADD R0,R1,#CONST
label1: ADD R3,R4,R5
        CMP R0,R3
label2: SUB R5,R6,R7
```

Assembly code

CONST = 5
label1 = start + 4
label2 = start + 12

Symbol Table

Program Location Counter: PLC

The object file

- Result of assembly
- Several standards:
  - ELF (unix), COFF (unix, windows)
- The object file includes:
  - Symbol table
  - Binary program code (text segment)
  - Data (data segment)
  - Uninitialized data (bss segment)
  - Information about relocatable parts
  - Debug data (references to source files)

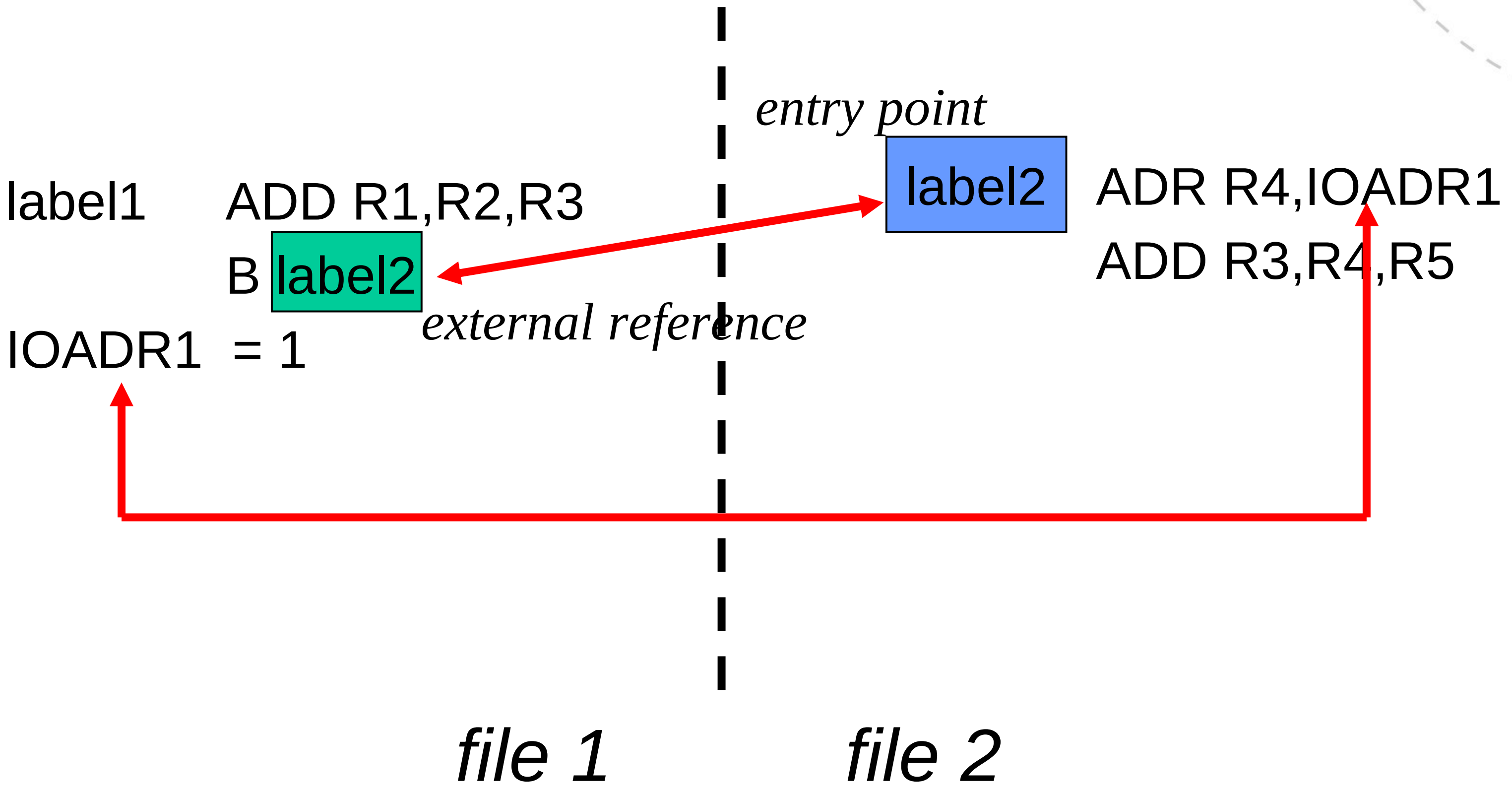
Symbols in C code

- What ends up in the object file symbol table from a C source?

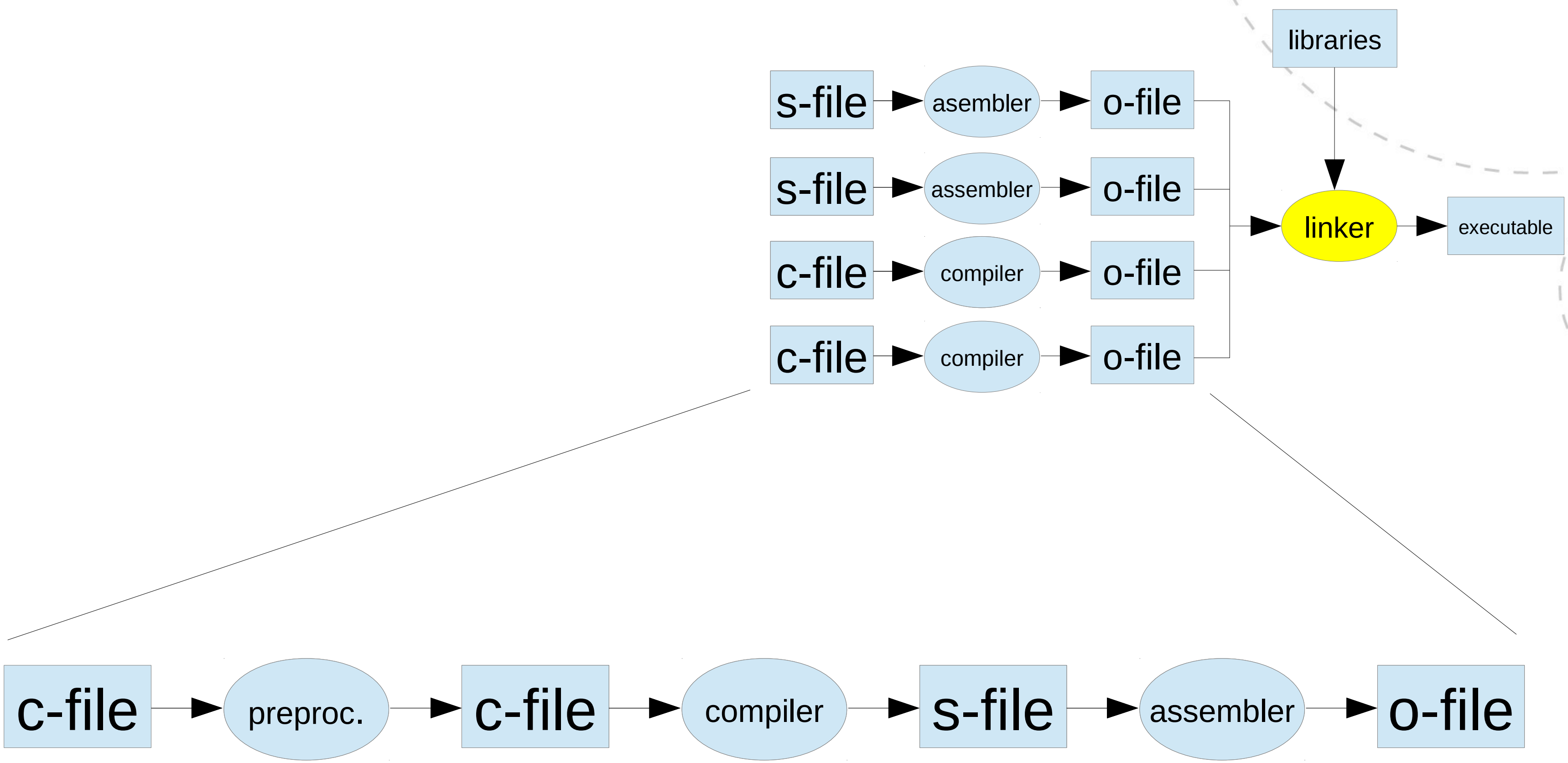
Symbols in C code

- What ends up in the object file symbol table from a C source?
  - Function names
    - But not if declared `static`
  - Global variable names
    - But not if declared `static`
  - Not macros (#defines)
- To access symbols from other object files:
  - Need function prototypes to access external functions
  - Need to declare variable `extern` to access external variables
    - `extern int variable;`
  - Typically done in a header file describing the object file external interface

External references and entry points



Linking

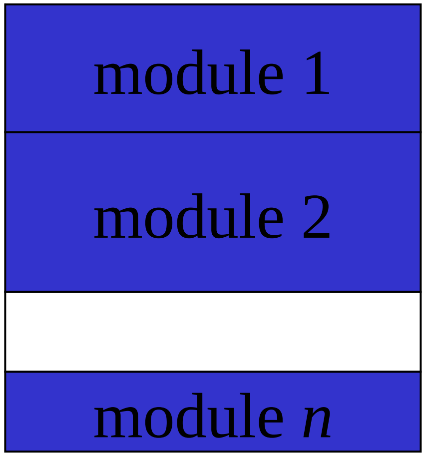

libraries
s-file
asembler
o-file
s-file
assembler
o-file
c-file
compiler
o-file
c-file
compiler
o-file
linker
executable
c-file
preproc.
c-file
compiler
s-file
assembler
o-file

Linking

- Takes multiple object files and libraries, and creates one executable file
  - Combines all the object file segments (text, data, bss)
  - Combines all the symbol tables
  - Determines start addresses for all the modules
  - Resolves all symbols
    - Transforms all relative addresses to absolute addresses
    - Fix all unresolved references
    - Can't find a referenced symbol in the merged table? Error!

Module order

- The modules are put in specific locations by the linker
  - Order typically given by the command line argument order
- The linker merges the symbol tables and creates a new symbol table for the merged module
- Addresses in embedded systems often have specific meanings
  - Different addresses can map to different devices/memories (DRAM, EEPROM, cached, non-cached etc)



Custom linker script

- When non-standard behaviour is needed
- Example: Linking for embedded device, without OS
  - Need to tell linker about the stack, heap and which addresses can be used for data and program

```
SECTIONS
{
 . = 0x0;
 .text : {
 startup.o (.text)
 *(.text)
 }
 .data : { *(.data) }
 .bss : { *(.bss COMMON) }
 . = ALIGN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
}
```

Dynamic linking

- Most operating systems can link modules at load time
  - Shared libraries
  - .so (linux), .dll (windows)
  - Saves storage space
  - Increases load time
- The compile time linker does not include code from the shared library, only creates references to it
- Small embedded systems do not use shared libraries
- More on dynamic linking and loaders in the next lecture

Interpreters and JIT compilers

- Interpreter
  - Translates and runs expression by expression
  - Slow
  - Can debug, test and run in the same environment
- Just in time-compiler (JIT)
  - Compiles code at runtime
    - Enables different kinds of optimizaions
    - Needs warm-up time
    - Stores the translation for use next time
    - Can have large memory requirements

Lecture overview

- Models of programs
- Compiler toolchain
  - Precompiler
  - Compiler and compilation techniques
  - Assembler
  - Linker
- Optimizing execution time
- Energy aware programming

Optimizing execution time

- Embedded systems often react to real time events
  - Need to meet deadlines
  - May need to guarantee that a deadline is always met
- Need to be able to analyze execution time
  - Worst case execution time is often most important
- We need techniques that reduce execution time
- Execution time may vary
  - Input values
  - Cache state
  - Pipeline effects
  - Dynamic interactions of different units in the system

Measuring execution time

- Can be difficult
  - Which input to generate worst case performance?
  - Cache contents
- Software profiler
  - gprof, or similar
  - Changes the code you are profiling
- CPU simulator
  - Different levels of abstraction possible
  - Accuracy vs. simulation speed
  - I/O effects difficult to simulate
- Hardware profiler
  - ARM trace bus
- Logic analyzer

Optimizing execution time

- Make sure you choose the best algorithm
  - No use in optimizing an n^2 algorithm if there exists an n*log n
  - Alg.dat
- Profile and find the parts of the code where time is spent
  - Don't waste your time optimizing if you don't know which parts contribute most to execution time

Reducing program size

- Data footprint
  - Goals
    - Reduce number of main memory accesses
    - Make everything fit inside a small embedded system RAM
  - Reuse constants
  - Reuse variables and data strutures
    - If not used at the same time, storage can be reused
- Code size
  - Goals:
    - Reduce instruction cache misses
    - Fit inside limited program memory
  - Choose a processor with compact instructions
  - Use specialized instructions wherever possible
  - Compiler optimizations
    - gcc -Os

Lecture overview

- Models of programs
- Compiler toolchain
  - Precompiler
  - Compiler and compilation techniques
  - Assembler
  - Linker
- Optimizing execution time
- Energy aware programming

Static SW techniques for energy efficiency

- Overall advice: *High performance == low power*
- Execution time optimization
  - Clear correlation between execution time and energy consumption
  - All techniques for reducing execution time will likely reduce energy consumption rougly as much
- Make use of your available HW
  - Unused resources wastes energy
    - Use available resources to make execution time as short as possible
  - Example: If you have support for vector instructions (ARM NEON), their usage can probably help reduce energy consumption
  - Avoid using processors with capabilities you don't need

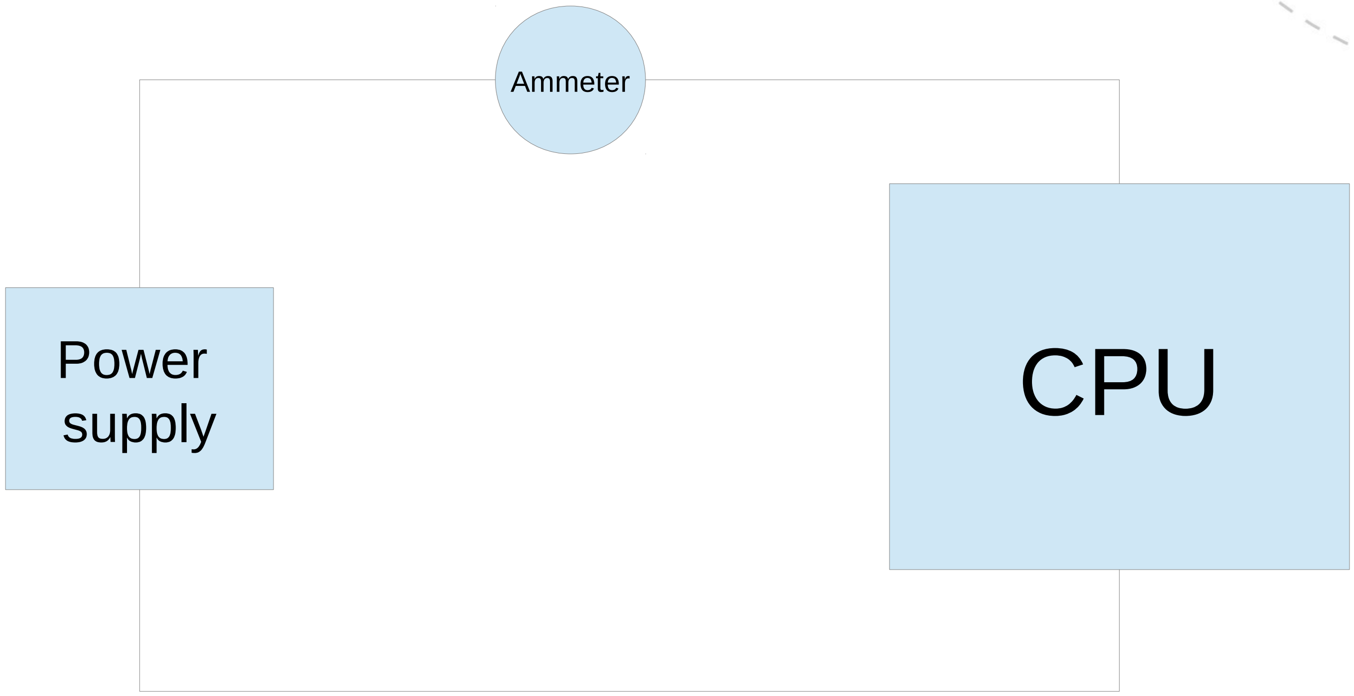
Static SW techniques for energy efficiency

- Memory access optimization
  - Memory transfers consumes a lot of energy
    - Especially if off-chip
    - Tens to hundreds of times more expensive than an ALU operation
  - Make sure registers are allocated properly to reduce loads and stores
  - Tune algorithms and datastructures such that your working set fits in cache
    - Minimize memory footprint
- I/O access optimization
  - Reduce flash/HD usage
  - Reduce network traffic

Static SW techniques for energy efficiency

- Other static techniques
  - Not very effective
  - Problematic because CPU designers don't give enough information in the datasheet
    - Measurements, trial and error
    - Architecture specific
  - Possible things to try:
    - Some instructions might result in less internal switching than others, even if equally fast
      - Use shift instead of multiplication?
      - Use fixed point (integer instructions) instead of floating point?
    - Instruction sequence can have an impact

Measuring energy consumption



Dynamic energy techniques

- Sleep modes, DVFS, ...
- Has a big and important impact on energy consumption
- Belongs to the OS domain
- Next lecture

Next lecture

- Thursday after easter (?)
  - Operating systems
    - Focus on embedded systems
    - ARM Linux as example
  - Reducing energy consumption with run-time techniques