Reminders

- Homework assignment 3 will be due on Thursday 11/4 before class

- Lab 4 will be due on 11/5

Recap from last class

- Real-time scheduling
  - Ensuring that all tasks can meet their deadlines
  - Responsiveness > throughput
    - Worst-case performance
- Schedulability
  - Schedulable utilization bound
- Optimal scheduling algorithms
  - RMS vs. EDF
- Priority inversion
- End-to-end scheduling framework
  - Dependency constraints among subtasks
  - Multi-processor scheduling

ECE 1175
Embedded Systems Design

Program Optimization I

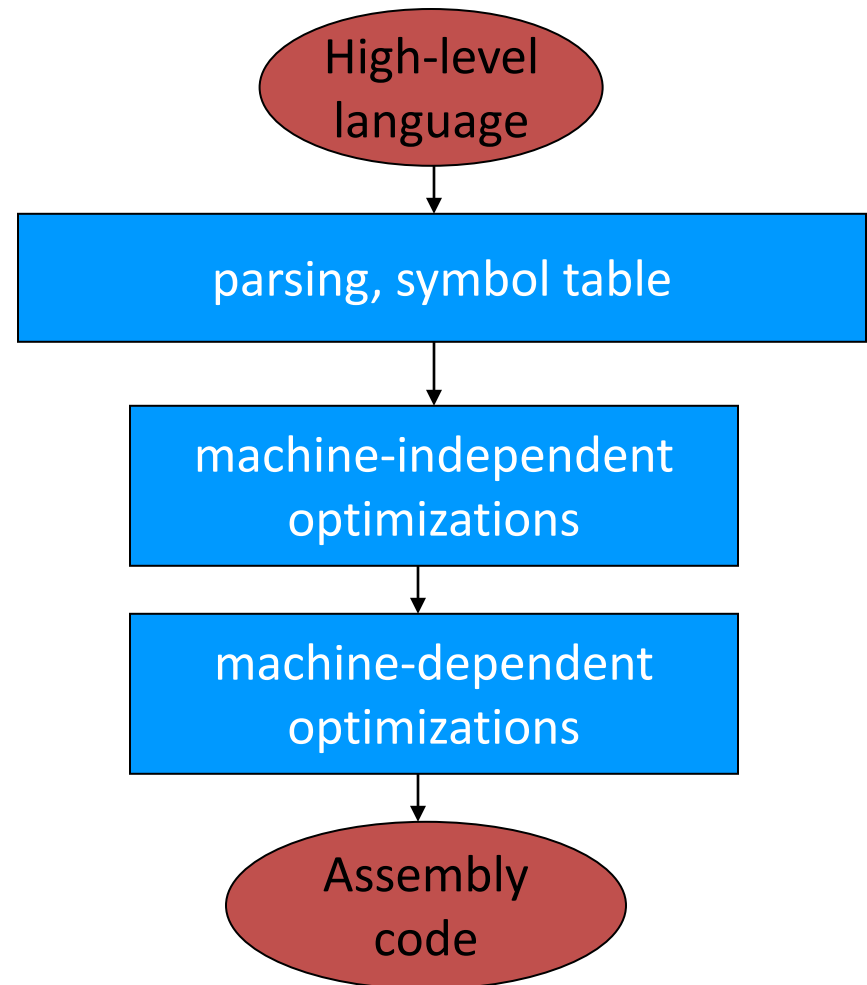## Wei Gao

Program Transformation

- How high-level program is transformed to executable binary code?
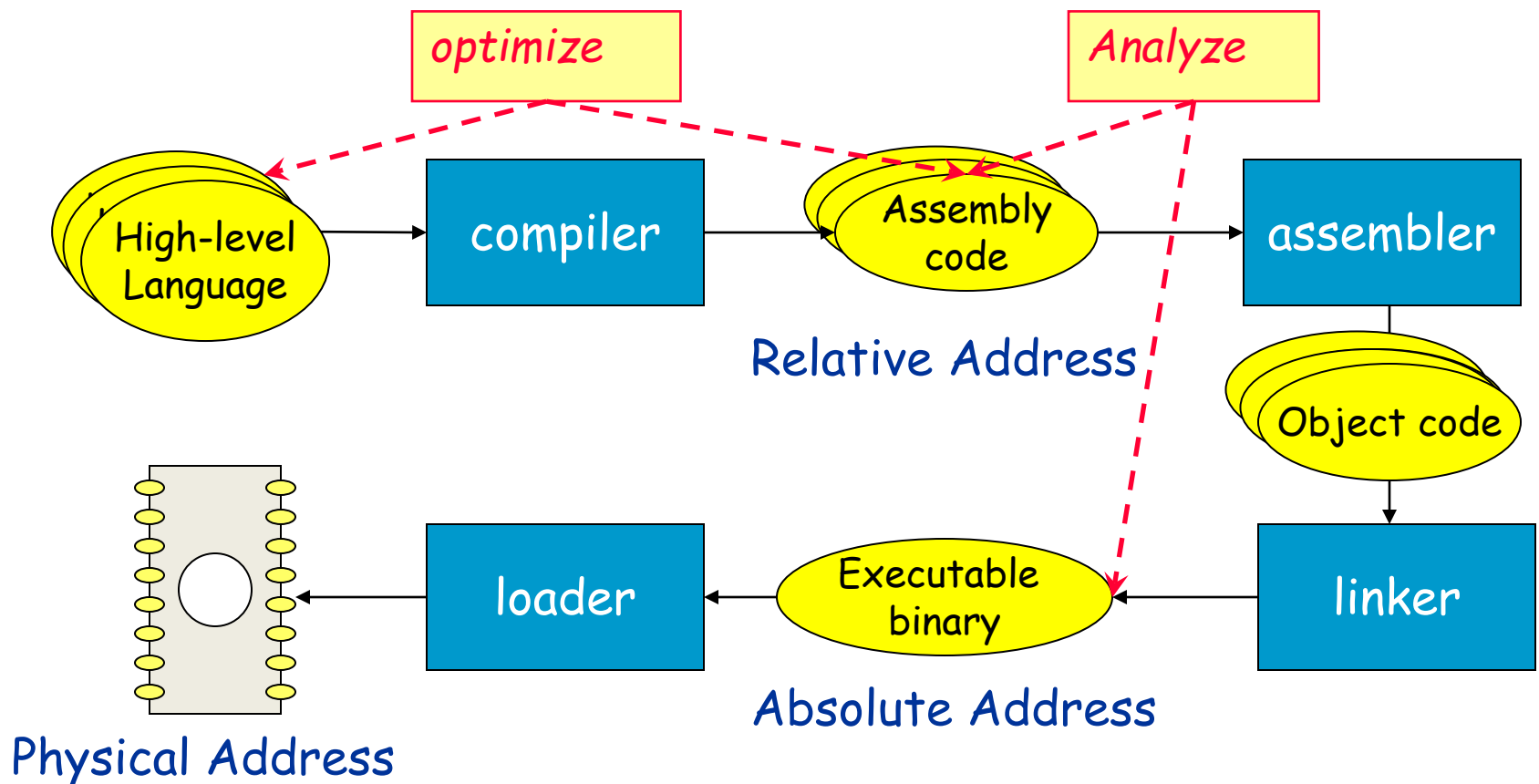
Compilation

- Compilation strategy (by Wirth):
  - compilation = translation + optimization

- Compiler determines quality of code:
  - use of CPU resources;
  - memory access scheduling;
  - code size.

Basic compilation phases

```
┌─────────────────────┐
│  High-level language │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  parsing, symbol table │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  machine-independent │
│     optimizations    │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  machine-dependent   │
│     optimizations    │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│    Assembly code     │
└─────────────────────┘
```

Basic Compilation Optimization

- 1. Expression simplification

- 2. Dead code elimination

- 3. Function inlining

- 4. Loop optimizations

- 5. Register allocation

1. Expression Simplification

- Algebraic:
  - `a*b + a*c ⇨ a*(b+c)`
  - Three operations change to two operations
- Constant folding:
  - `for (i=0; i<8+1; i++)`
  - ⇨ `for (i=0; i<9; i++)`
- Strength reduction of constant multiplication
  - `a*2⇨a<<1; a*3⇨a*(2+1)⇨a<<1+a;`
  - `a*1000⇨a*(1024-16-8)⇨a<<10 - a<<4 - a<<3`
  - SHIFT, ADD, SUB take one cycle but multiplication takes about 10 cycles

2. Dead Code Elimination

- Dead code:

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

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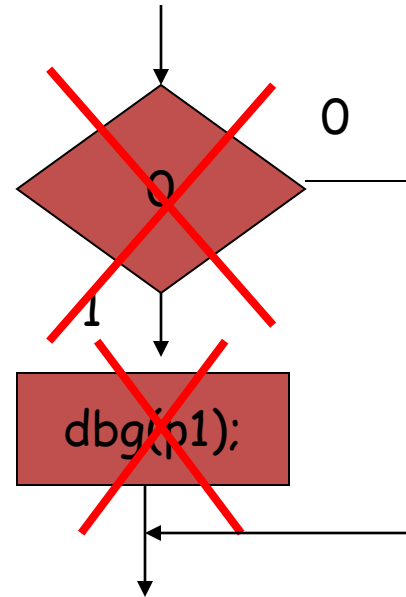
3. Function inlining

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

z = w + x - y;
```

- An inline function's body is inserted directly (like a substitution) in the compiled code at the point where the function is called.

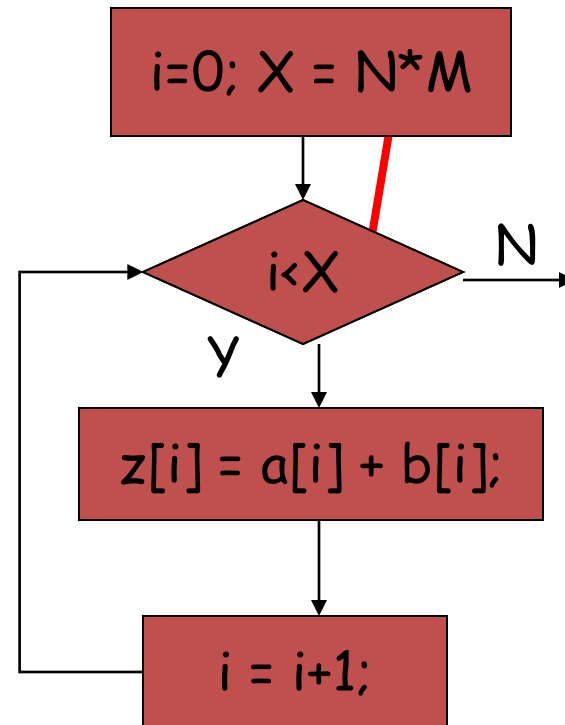- Improve performance by reducing function call overhead

4. Loop Optimizations

- Loops are good targets for optimization.

- Basic loop optimizations:

  - Code motion;

  - Reduce loop overhead;

  - Increase opportunities for pipelining and parallelism;

Code Motion

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

- Move computation inside a loop to the outside to avoid doing it repeatedly.
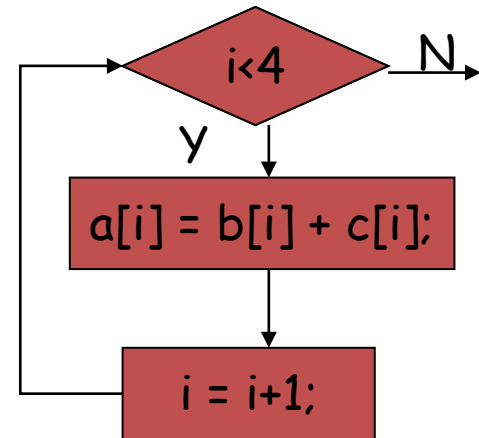
Loop Unrolling

- Reduces loop overhead

```
for (i=0; i<4; i++)
  a[i] = b[i] * c[i];
⇨
for (i=0; i<4; i+=2) {
  a[i] = b[i] * c[i];
  a[i+1] = b[i+1] * c[i+1];
}
```


- May allow to be executed in CPU's pipeline
- May increase the code size

Loop Fusion

- Combines multiple loops into one loop to reduce loop overhead

```
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];
}
```

- Necessary conditions:
  - Loops share the same index
  - No dependencies between two loops

5. Register Allocation

- Processor registers
    - A very small amount of very fast computer memory
    - Used to speed the execution of computer programs
    - Provides quick access to most commonly used values
    - Memory hierarchy: register – cache – main memory – disk
- Reduce the number of used registers
    - Fit more frequently used variables in registers
    - Load once, use many times
- → Reduce number of cache/memory access
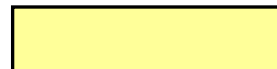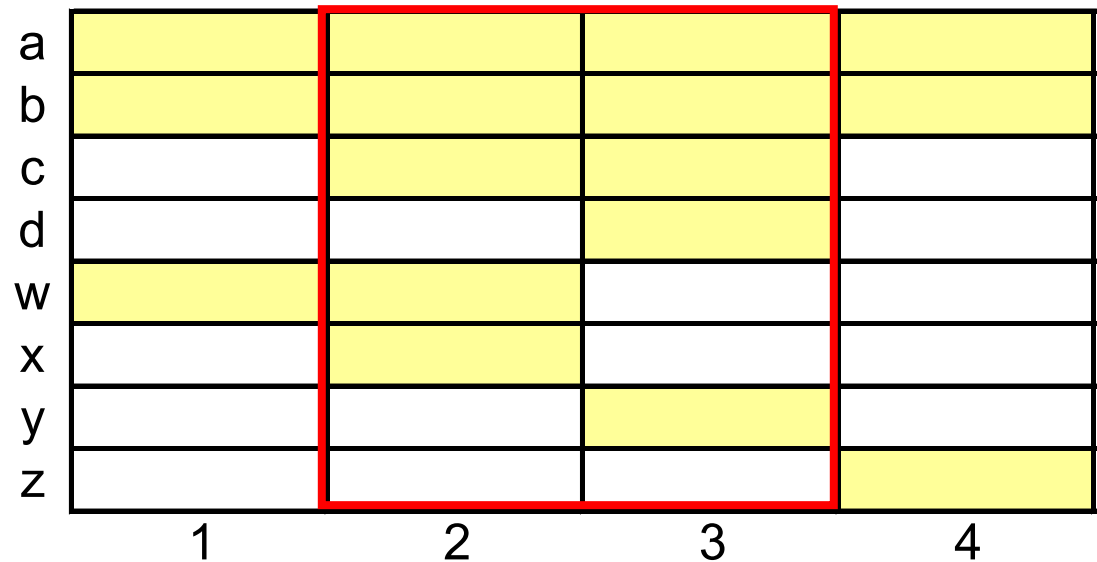- → Reduce energy consumption

Register Lifetime Graph

1. `w = a + b;`
2. `x = c + w;`
3. `y = c + d;`
4. `z = a - b;`

	1	2	3	4
a	▓	▓	▓	▓
b	▓	▓	▓	▓
c		▓	▓	
d			▓	
w	▓	▓		
x		▓		
y			▓	
z				▓

� means this variable should be loaded to register

After Rescheduling

no. of needed registers = 4

```
1. w = a + b;
2. z = a - b;
3. x = c + w;
4. y = c + d;
```

	1	2	3	4
a	▢	▢		
b	▢	▢		
c			▢	▢
d				▢
w	▢	▢	▢	
x			▢	
y				▢
z		▢		

Cannot change dependencies between instructions!

Summary of Compilation Optimization

- Use registers efficiently.

- Optimize loops.

- Optimize function calls.

- Optimize cache behavior:

  - instruction conflicts can be handled by rewriting code, rescheduling;