

# Program Design For Embedded Systems

Ingo Sander  
ingo@imit.kth.se



## Challenges

- Embedded programs
  - meet system deadlines
  - handle memory resources efficiently
  - meet power consumption requirements
- Advances in design techniques and tools help to use more abstract languages than assembler in order to allow a higher efficiency

September 17, 2004

2B1447 Embedded Systems

2

## Program design and analysis

- Design patterns
- Representations of programs
- Assembly and linking



September 17, 2004

2B1447 Embedded Systems

3

## Design patterns

- A design pattern is a generalized and design solution for a given class of problems
- Design patterns encapsulate knowledge of previous designers and are typically efficient solutions to a specific problem
- A designer can use a particular design pattern and modify it for his particular needs



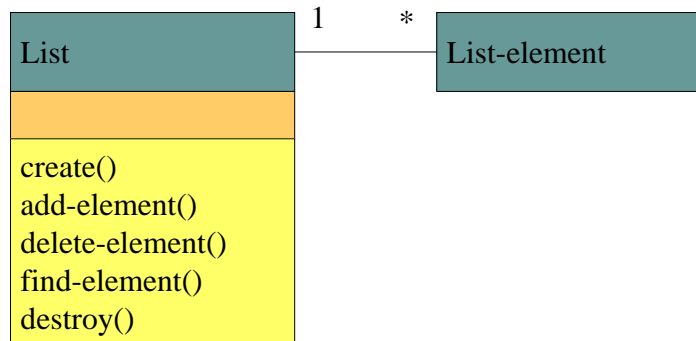
September 17, 2004

2B1447 Embedded Systems

4



## List design pattern (UML)



© 2000 Wolf (Morgan Kaufman)



## Design pattern elements

- Design patterns can have many different elements
  - Class diagram
  - State diagrams
  - Sequence diagrams
  - etc.



## State machine

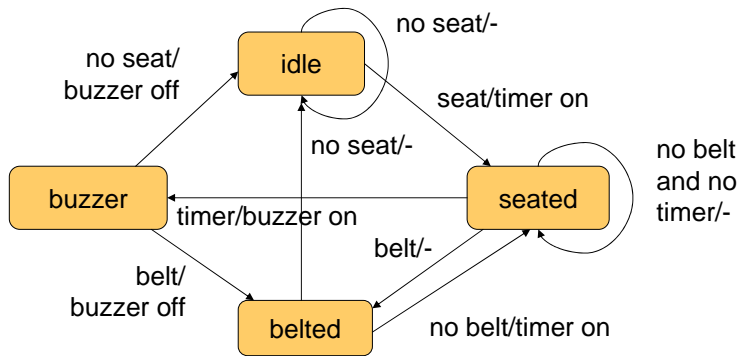
- State machines are a key component in embedded system design!
- They are used for
  - Asynchronous communication (handshake)
  - Protocols
  - Control parts in an embedded system application
  - Etc...



## State machine example: Seat belt controller

- The controller turns on a buzzer, if a person sits in a car and does not fasten the seat belt within amount of time
- Inputs: seat, timer, belt
- Outputs: timer\_on, buzzer
- The state machine is executed in an infinite loop!

# Seat belt controller State machine



© 2000 Wolf (Morgan Kaufman)

# C implementation



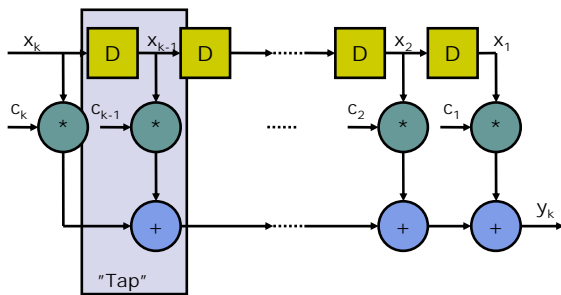
```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
    case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
               break;
    case SEATED: if (belt) state = BELTED;
                 else if (timer) state = BUZZER;
                 break;
    ...
}
```

© 2000 Wolf (Morgan Kaufman)

# The need for efficient buffers

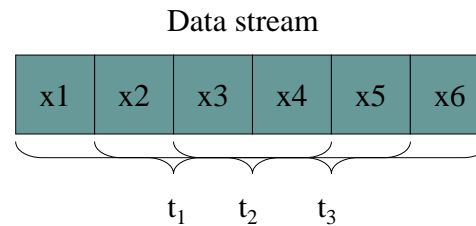


- In DSP applications data streams come in regularly and must be processed on the fly
- A FIR-Filter needs a buffer that holds the last  $k$  values

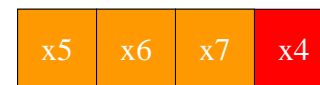


$$y_k = c_k x_k + c_{k-1} x_{k-1} + \dots + c_1 x_1$$

# Circular buffer



A circular buffer uses memory efficiently!



Circular buffer

© 2000 Wolf (Morgan Kaufman)



## Circular buffer implementation: FIR filter

```
int circ_buffer[N], circ_buffer_head = 0;
int c[N]; /* coefficients */

...
int ibuf, ic;
for (f=0, ibuff=circ_buff_head, ic=0;
     ic<N; ibuff=(ibuff==N-1?0:ibuff++), ic++)
    f = f + c[ic]*circ_buffer[ibuff];
```

© 2000 Wolf (Morgan Kaufman)



## Models of programs

- Compilers (and synthesis tools) work not directly on the source code
  - Source code is verbose
  - Several languages are based on the same model (e.g. Imperative languages like C, Pascal, Fortran)
- The intermediate model is easier to transform than the source code, which leads to better optimization results



## 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.

© 2000 Wolf (Morgan Kaufman)



## Single assignment form

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

original basic block

single assignment form

© 2000 Wolf (Morgan Kaufman)

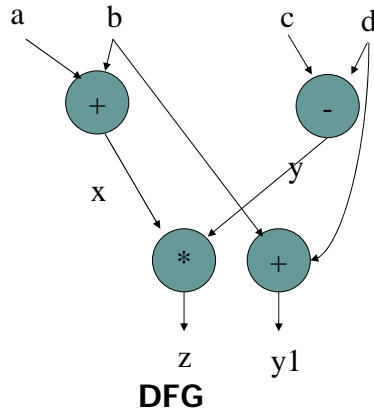


## Data flow graph

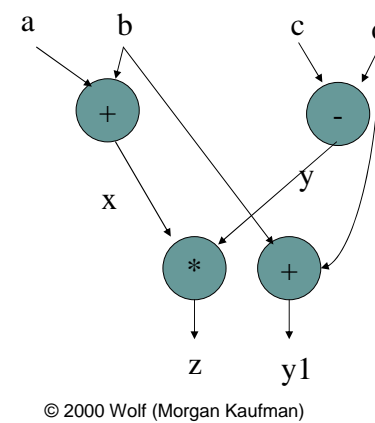
$x = a + b;$   
 $y = c - d;$   
 $z = x * y;$   
 $y1 = b + d;$

single assignment form

© 2000 Wolf (Morgan Kaufman)



## DFGs and partial orders



- Partial order:
  1.  $a+b, c-d$
  2.  $b+d, x*y$
- Can do pairs of operations in any order.
- Easy to find tasks, which can be implemented in parallel



## Control-data flow graph

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

© 2000 Wolf (Morgan Kaufman)



## Data flow node

Encapsulates a data flow graph:

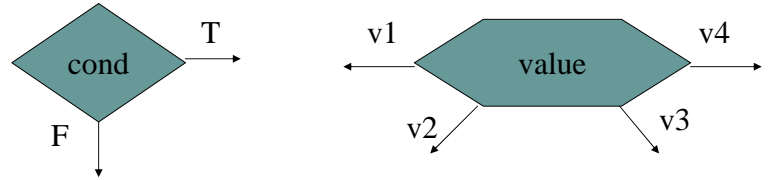
$x = a + b;$   
 $y = c + d$

Write operations in basic block form for simplicity.

© 2000 Wolf (Morgan Kaufman)



# Control



© 2000 Wolf (Morgan Kaufman)

## Equivalent forms

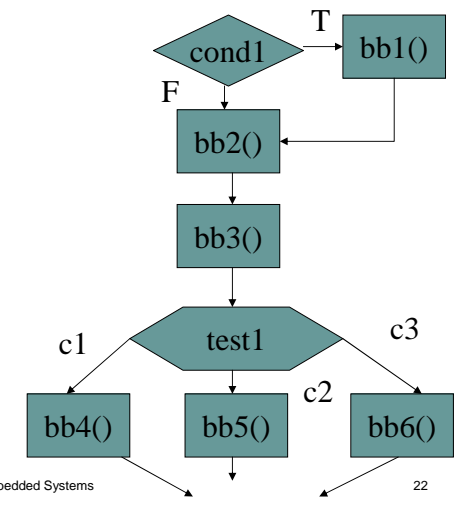
# CDFG example

```

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

```

© 2000 Wolf (Morgan Kaufman)



# for and while loop

```

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

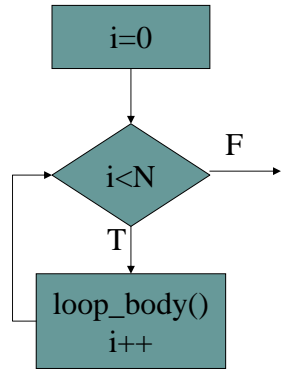
```

```

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

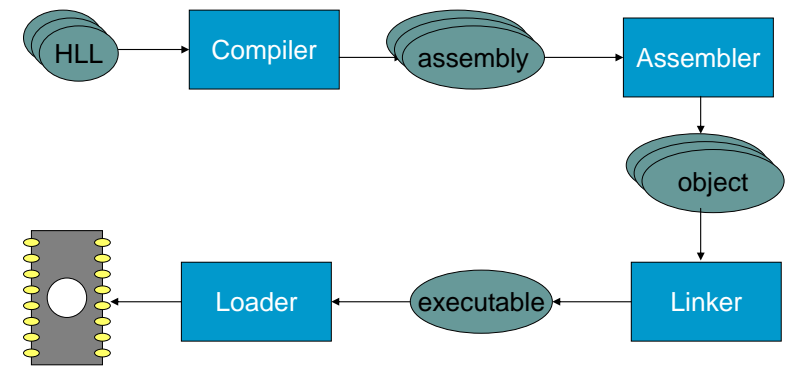
```

© 2000 Wolf (Morgan Kaufman)



# Assembly and linking

- Last steps in compilation:



© 2000 Wolf (Morgan Kaufman)



## 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.

© 2000 Wolf (Morgan Kaufman)



## 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      ;(Does not work for ARM, done by
                                ;memory map for linker)
label1  ADR r4,c
      
```

© 2000 Wolf (Morgan Kaufman)



## Symbol table

|    |              |    |     |
|----|--------------|----|-----|
|    | ADD r0,r1,r2 | xx | 0x4 |
| xx | ADD r3,r4,r5 | yy | 0xc |
|    | CMP r0,r3    |    |     |
| yy | SUB r5,r6,r7 |    |     |

assembly code

symbol table

© 2000 Wolf (Morgan Kaufman)



## 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.

© 2000 Wolf (Morgan Kaufman)



## Symbol table example

|                |    |              |       |
|----------------|----|--------------|-------|
| <b>PLC=0x0</b> |    | ADD r0,r1,r2 | xx0x4 |
| <b>PLC=0x4</b> | xx | ADD r3,r4,r5 | yy0xc |
| <b>PLC=0x8</b> |    | CMP r0,r3    |       |
| <b>PLC=0xc</b> | yy | SUB r5,r6,r7 |       |

→

© 2000 Wolf (Morgan Kaufman)



## 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 (not available in ARM SDT).
  - **EQU** generates symbol table entry without advancing PLC.
  - **Data statements** define data blocks.

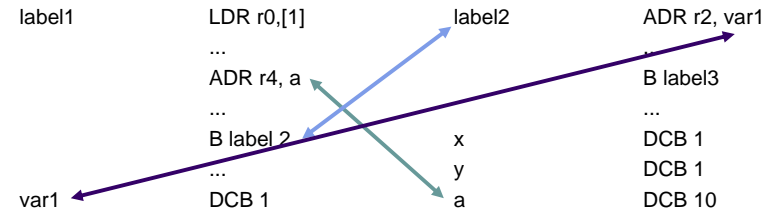


# Linking

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



# External references and entry points



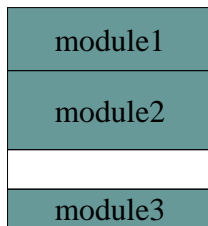
| External Ref. | Entry Points |
|---------------|--------------|
| a             | label1       |
| label2        | var1         |

| External Ref. | Entry Points |
|---------------|--------------|
| var1          | label2       |
| label3        | x            |
|               | y            |
|               | a            |



# Module ordering

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



© 2000 Wolf (Morgan Kaufman)



# 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.

© 2000 Wolf (Morgan Kaufman)



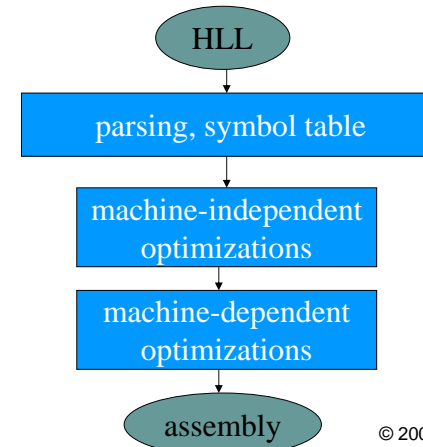
# Compilation

- Compilation strategy (Wirth):
  - compilation = translation + optimization
- Compiler determines quality of code:
  - use of CPU resources;
  - memory access scheduling;
  - code size.

© 2000 Wolf (Morgan Kaufman)



# Basic compilation phases



© 2000 Wolf (Morgan Kaufman)



# Statement translation and optimization

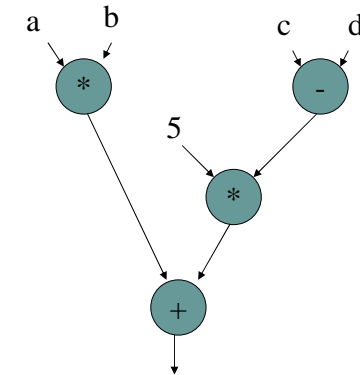
- Source code is translated into intermediate form such as CDFG.
- CDFG is transformed/optimized.
- CDFG is translated into instructions with optimization decisions.
- Instructions are further optimized.

© 2000 Wolf (Morgan Kaufman)



# Arithmetic expressions

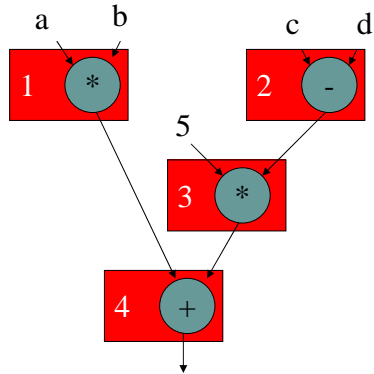
$a*b + 5*(c-d)$   
expression



DFG

© 2000 Wolf (Morgan Kaufman)

# Arithmetic expressions, cont'd.



```
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
```

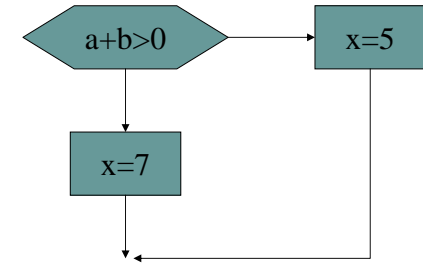
© 2000 Wolf (Morgan Kaufman)

DFG

code

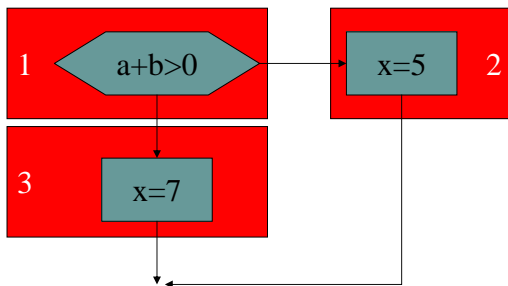
# Control code generation

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



© 2000 Wolf (Morgan Kaufman)

# 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
label3 LDR r3,#7
ADR r5,x
STR r3,[r5]
```

stmtent ...

© 2000 Wolf (Morgan Kaufman)

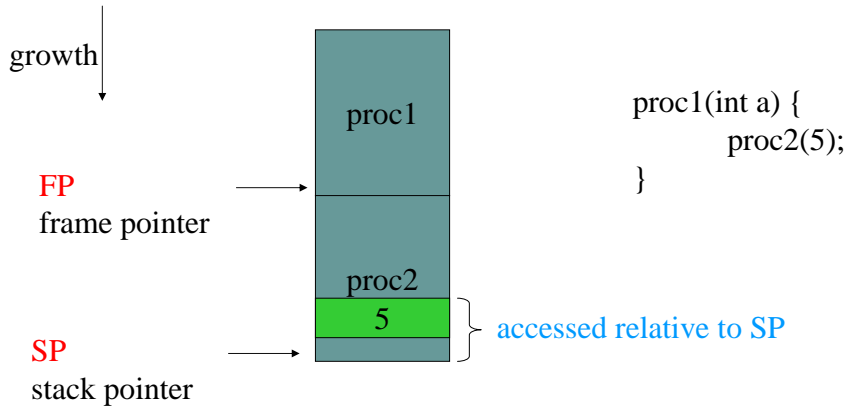
# Procedure linkage

- Need code to:
  - call and return;
  - pass parameters and results.
- Parameters and returns are passed on stack.
  - Procedures with few parameters may use registers.

© 2000 Wolf (Morgan Kaufman)



# Procedure stacks



© 2000 Wolf (Morgan Kaufman)



# ARM procedure linkage

- APCS (ARM Procedure Call Standard):
  - r0-r3 (a1-a4) pass parameters into procedure. Extra parameters are put on stack frame.
  - r0 (a1) holds return value.
  - r4-r8 (v1-v5) hold register values.
  - r11 (fp) is frame pointer, r13 (sp) is stack pointer.
  - r10 (sl) holds limiting address on stack size to check for stack overflows.

© 2000 Wolf (Morgan Kaufman)



# Data structures

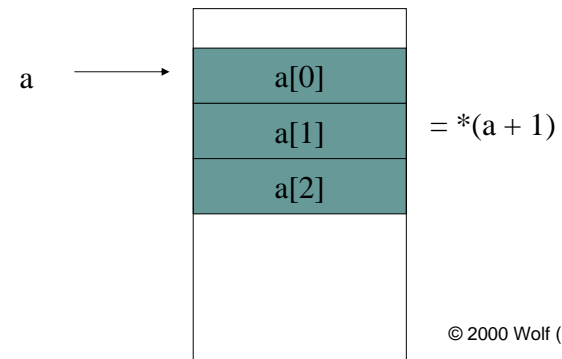
- Different types of data structures use different data layouts.
- Some offsets into data structure can be computed at compile time, others must be computed at run time.

© 2000 Wolf (Morgan Kaufman)



# One-dimensional arrays

- C array name points to 0th element:

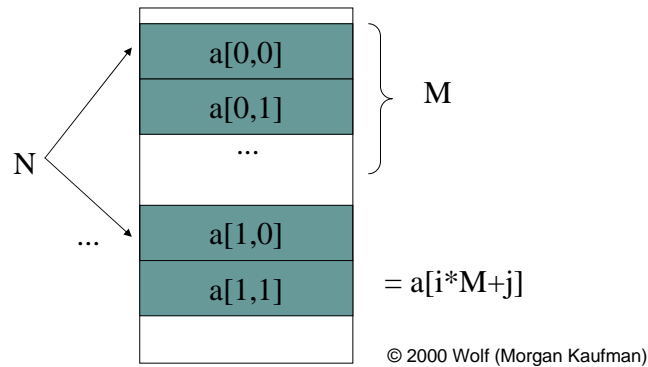


© 2000 Wolf (Morgan Kaufman)



# Two-dimensional arrays

- Column-major layout:

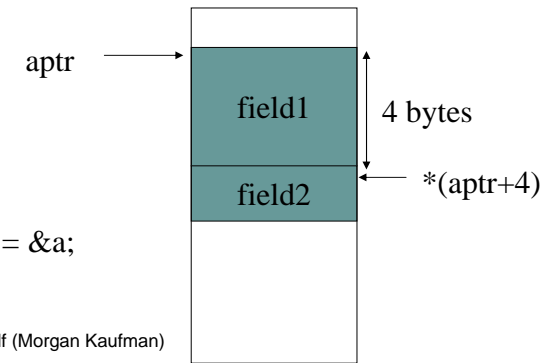


# Structures

- 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:
  - $a*2 = a<<1$

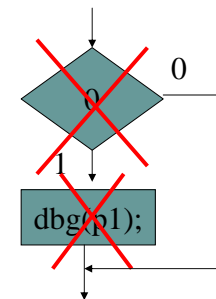
© 2000 Wolf (Morgan Kaufman)



# Dead code elimination

- Dead code:
 

```
#define DEBUG 0
if (DEBUG) dbg(p1);
```
- Can be eliminated by analysis of control flow, constant folding.



© 2000 Wolf (Morgan Kaufman)



## 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.



# Loop tiling

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

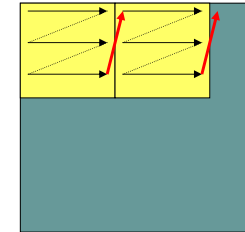
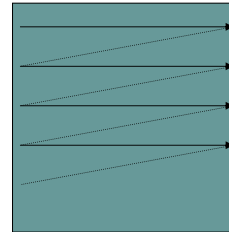
© 2000 Wolf (Morgan Kaufman)



# Loop tiling example

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

```
for (i=0; i<N; i+=2)
  for (j=0; j<N; j+=2)
    for (ii=0; ii<min(i+2,N); ii++)
      for (jj=0; jj<min(j+2,N); jj++)
        c[ii] = a[ii,jj]*b[jj];
```



© 2000 Wolf (Morgan Kaufman)



# Array padding

- Add array elements to change mapping into cache:

|        |        |        |
|--------|--------|--------|
| a[0,0] | a[0,1] | a[0,2] |
| a[1,0] | a[1,1] | a[1,2] |

|        |        |        |        |
|--------|--------|--------|--------|
| a[0,0] | a[0,1] | a[0,2] | a[0,2] |
| a[1,0] | a[1,1] | a[1,2] | a[1,2] |

before

© 2000 Wolf (Morgan Kaufman)

after



# Register allocation

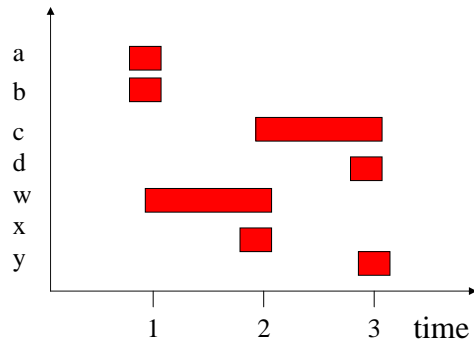
- Goals:
  - choose register to hold each variable;
  - determine lifespan of variable in the register.
- Basic case: within basic block.

© 2000 Wolf (Morgan Kaufman)



## Register lifetime graph

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



© 2000 Wolf (Morgan Kaufman)



## 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.**

© 2000 Wolf (Morgan Kaufman)



## Reservation table

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

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

© 2000 Wolf (Morgan Kaufman)



## Software pipelining

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

© 2000 Wolf (Morgan Kaufman)



## Software pipelining (Harvard)

- Example:  
for (i=0; i<N; i++)  
  sum += a[i]\*b[i];
- Combine three iterations:
  - Fetch array elements a, b for iteration **i**.
  - Multiply a,b for iteration **i-1**.
  - Compute sum for iteration **i-2**.

© 2000 Wolf (Morgan Kaufman)



## Software pipelining (Harvard), cont'd

```

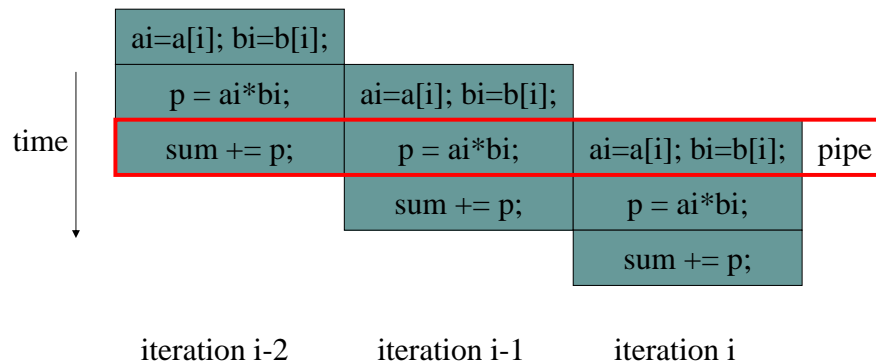
/* first iteration performed outside loop */
ai=a[0]; bi=b[0]; p=ai*bi;
/* initiate loads used in second iteration; remaining loads will be
performed inside the loop */
for (i=2; i<N-2; i++) {
  ai=a[i]; bi=b[i]; /* fetch for next cycle's multiply */
  p = ai*bi; /* multiply for next iteration's sum */
  sum += p; /* make sum using p from last iteration */
}
sum += p; p=ai*bi; sum +=p;

```

© 2000 Wolf (Morgan Kaufman)



## Software pipelining (Harvard) timing

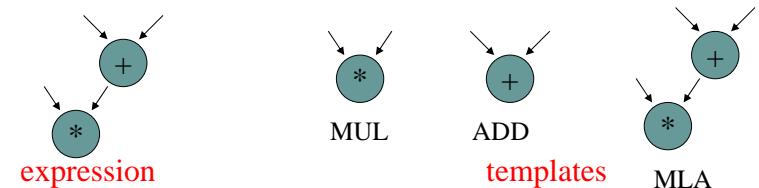


© 2000 Wolf (Morgan Kaufman)



## Instruction selection

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



© 2000 Wolf (Morgan Kaufman)



## 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.

© 2000 Wolf (Morgan Kaufman)



## 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.

© 2000 Wolf (Morgan Kaufman)