

Professur Technische Informatik
Prof. Dr. Wolfram Hardt

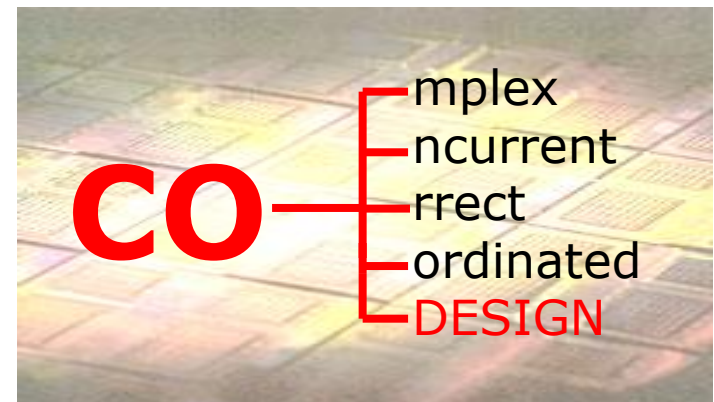
Hardware /Software Codesign I

Compiler and Code Generation

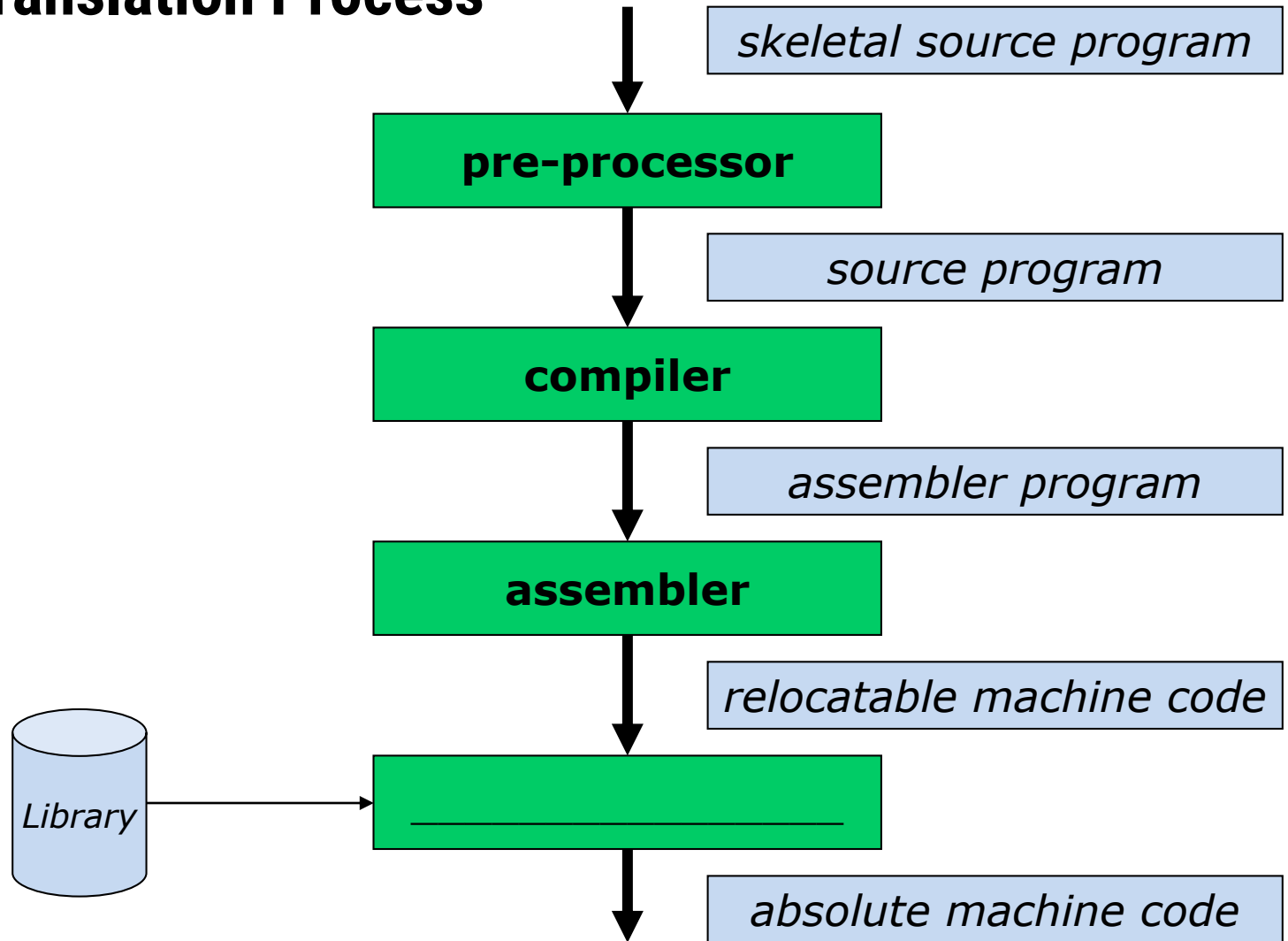
Prof. Dr. Wolfram Hardt
Dipl.-Inf. Michael Nagler

Contents

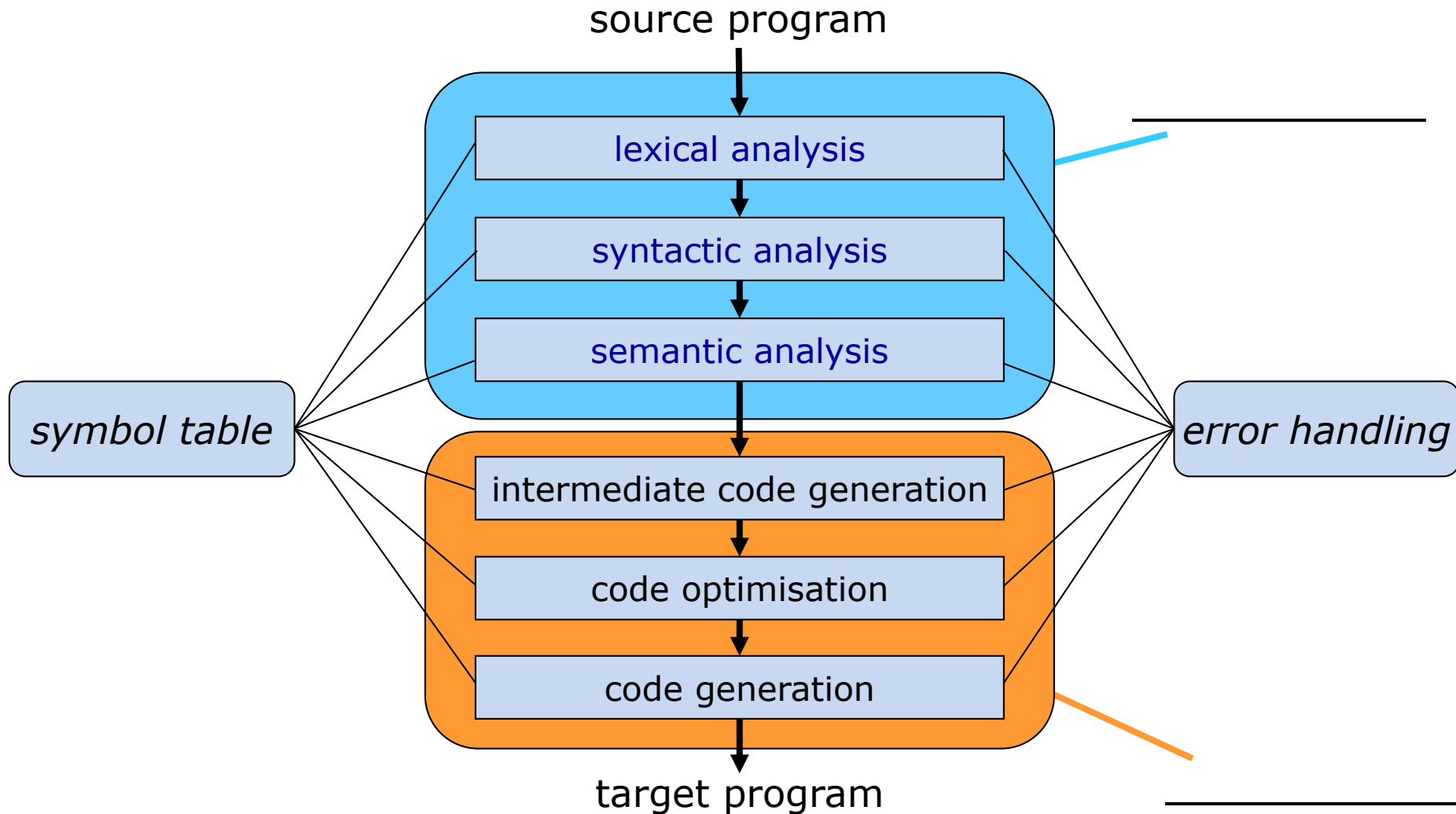
- Compiler Structure
- Code Generation
- Code Optimisation
- Code Generation for Specialised Processors
- Retargetable Compilers




Translation Process



Compiler Phases



Analysis

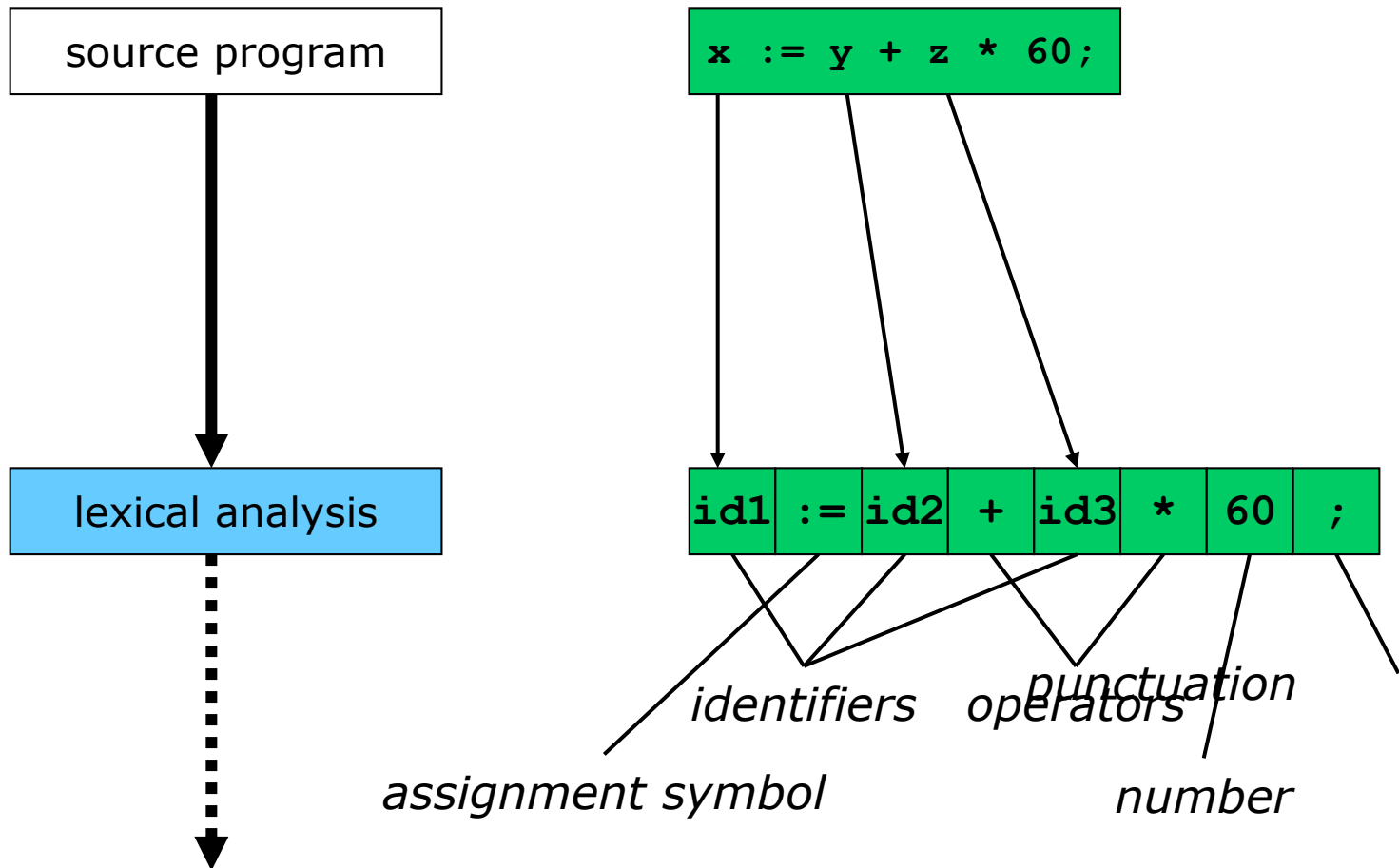
- lexical analysis
 - scanning of source program and splitting into symbols
 - recognition of regular expressions by _____
- syntactic analysis
 - parsing symbol sequences and construction of sentences
 - sentences are described by a context-free grammar
$$A \rightarrow \text{Identifier} := E$$
$$E \rightarrow E + E \mid E * E \mid \text{Identifier} \mid \text{Number}$$
- semantic analysis
 - ensure that program parts fit together
 - e. g. type casts: `int a = 4 * 3.14;` 

Synthesis

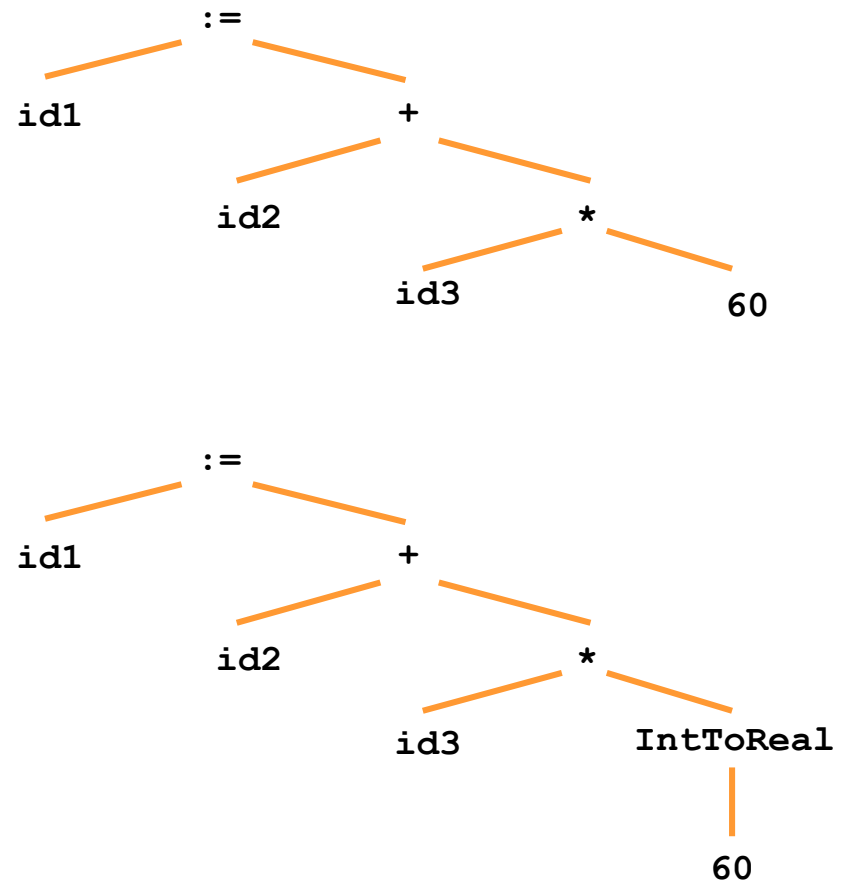
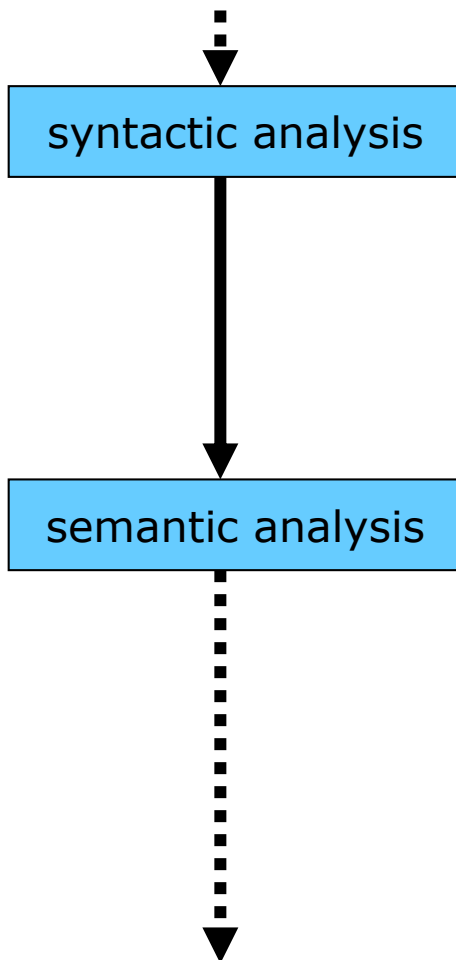
- intermediate code generation
 - _____ \rightarrow simplified retargeting
 - should be easy to generate
 - should be easily translatable into target program
- code optimisation
 - trade-off:

GP processors	$fast\ code \leftrightarrow fast\ translation$
specialised proc.	$fast\ code \leftrightarrow short\ (low\ mem.)\ code \leftrightarrow low\ power$
 - intermediate AND target code can be optimised
- code generation

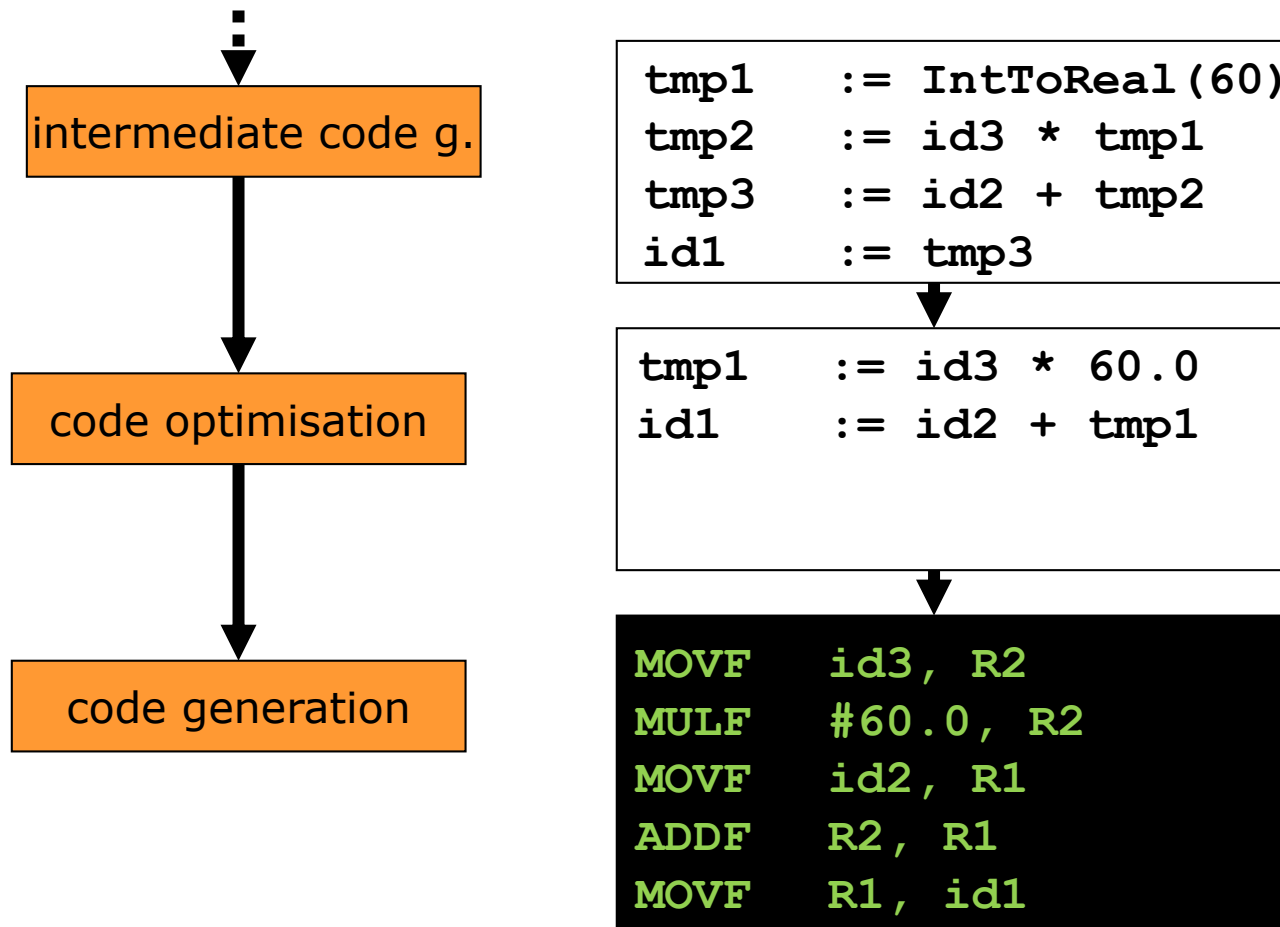
Example (I)



Example (II)

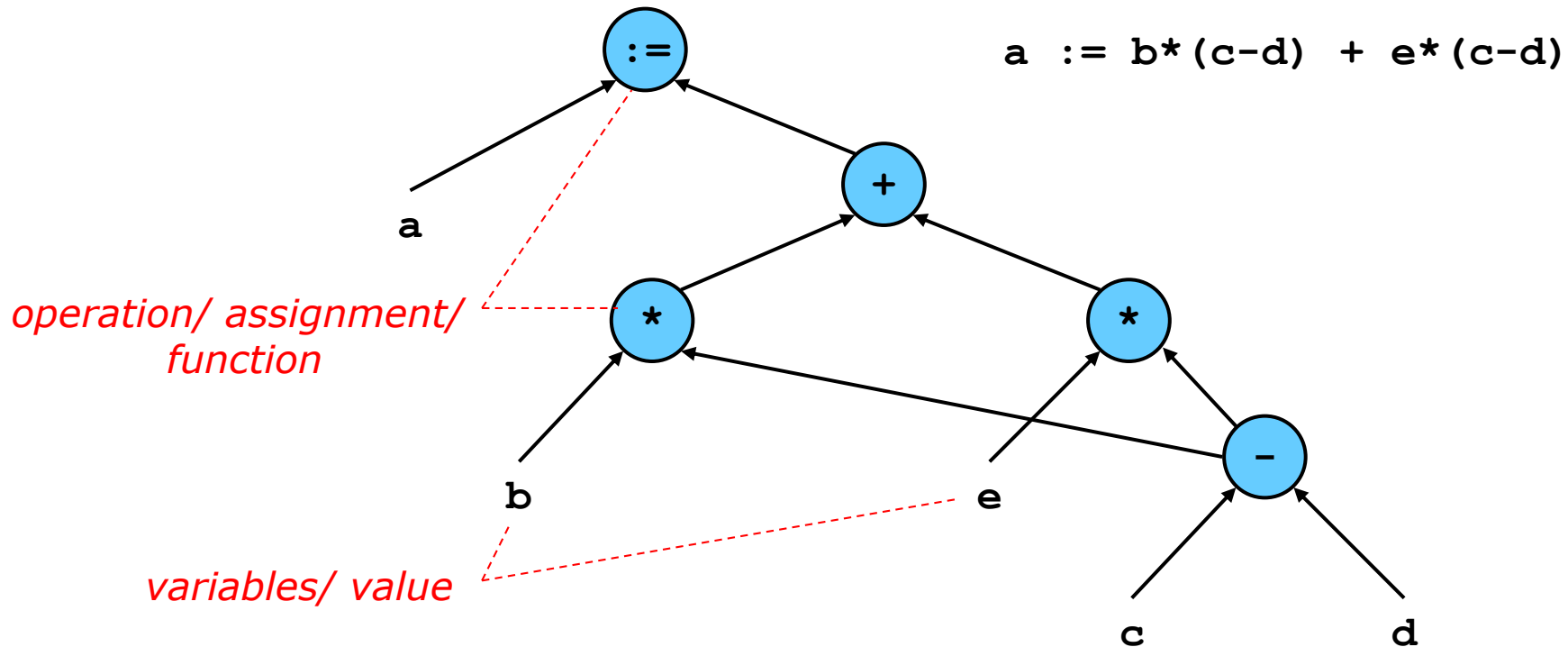


Example (III)



Representations (II)

•



- three-address code

Three-Address Code (I)

- three-address instructions
 - maximal 3 addresses (2 operands, 1 result)
 - maximal 1 operation and 2 operands

assignment instructions

```
x := y op z  
x := op y  
x := y
```

```
x := y[i]  
x[i] := y
```

```
x := &y  
y := *x  
*x := y
```

control flow instructions

```
goto L  
if x relop y goto L
```

subroutines

```
param x  
call p,n  
return y
```

Three-Address Code (II)

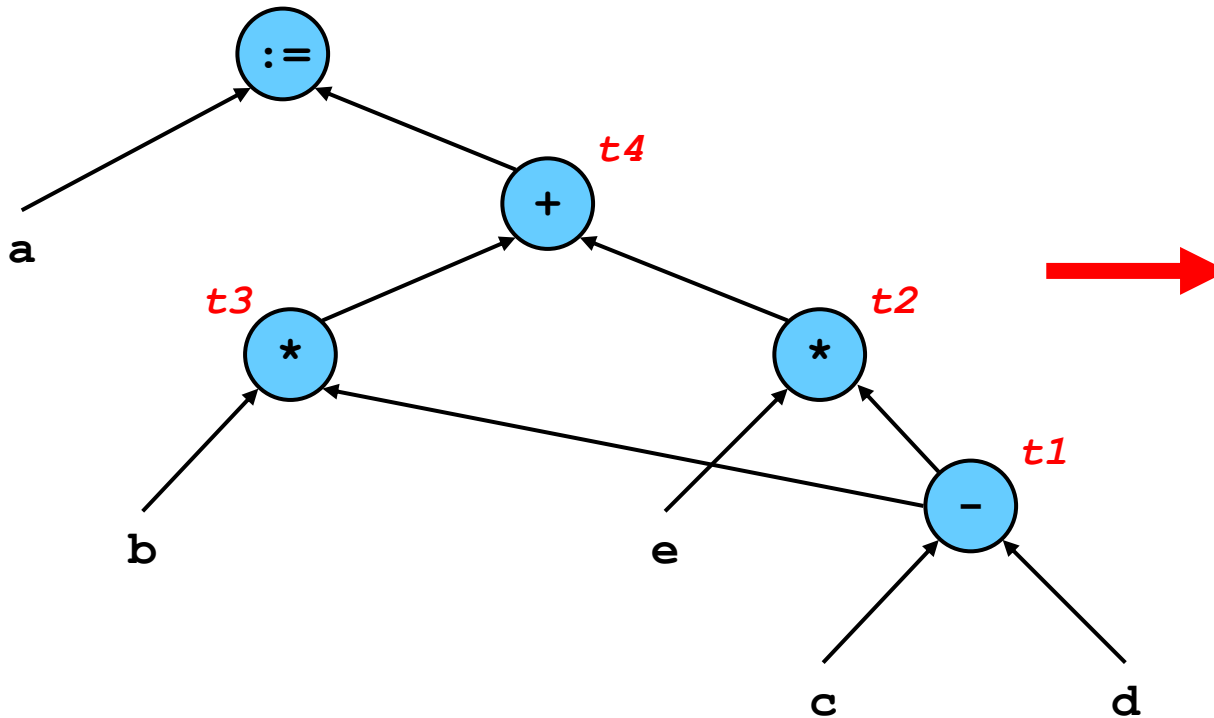
- advantages
 - dissection of long arithmetic expressions
 - temporary names facilitate reordering of instructions
 - forms a *valid* schedule
- **definition**

A three-address instruction $x := y \text{ op } z$

_____	x
_____	$y \text{ and } z$

Example

- three-address code generation



t1 := c - d
t2 := e * t1
t3 := b * t1
t4 := t3 + t2
a := t4

Basic Blocks (I)

- **definition**

A basic block (BB) is a _____
of instructions where the control flow enters at the beginning and
exits at the end, without stopping in-between or branching (except at
the end).

- **example**

```
if t1 = 3 goto K  
t1 := c - d  
t2 := e * t1  
t3 := t2 + t3  
if t4 < 10 goto L  
t2 := 3
```

BB

branching at the end of BB

Basic Blocks (II)

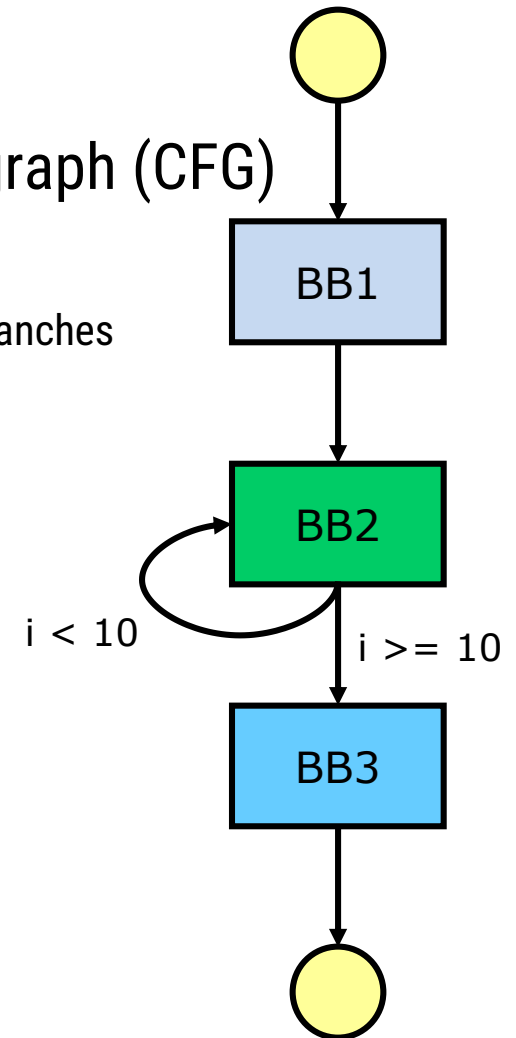
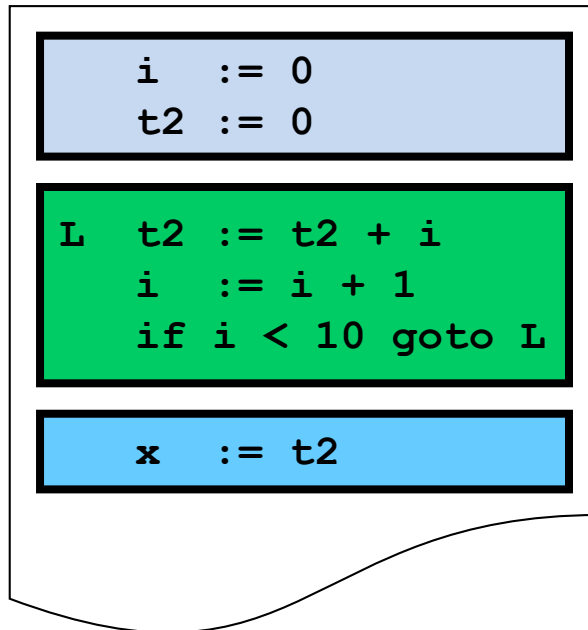
- a sequence of three-address instructions = set of basic blocks
 1. determine the block beginnings:
 - the _____ is a block beginning
 - targets of (un)conditional jumps are block beginnings
 - instructions that follow (un)conditional jumps are block beginnings
 2. determine the basic blocks:
 - every block beginning creates a basic block
 - the basic block consists of the instructions from block beginning to the next block beginning (exclusive) or to the end of program

Control Flow Graph

- " _____ " control flow graph (CFG)

- nodes are basic blocks

- no differentiation between instructions and branches



Directed Acyclic Graph of a BB

- **definition**

A DAG of a basic block is a directed acyclic graph with following node markings:

- _____ marked with a variable or constant name, variables with initial values are assigned the index 0
- inner nodes marked with operator symbol, operator decides whether value or address of associated variables is used
- optionally, node can be marked with a sequence of variable names, which are assigned the computed value

Example (I)

```
int i, prod, a[20], b[20];  
[...]  
  
prod = 0;  
i = 0;  
do  
{  
    prod = prod + a[i] * b[i];  
    i++;  
} while (i < 20)
```

C program

length of integer = 4 byte

```
(1)  prod := 0  
(2)  i := 0  
(3)  t1 := 4 * i  
(4)  t2 := a[t1]  
(5)  t3 := 4 * i  
(6)  t4 := b[t3]  
(7)  t5 := t4 * t2  
(8)  t6 := prod + t5  
(9)  prod := t6  
(10) t7 := i + 1  
(11) i := t7  
(12) if i < 20 goto (3)
```

three-address code

Example (II)

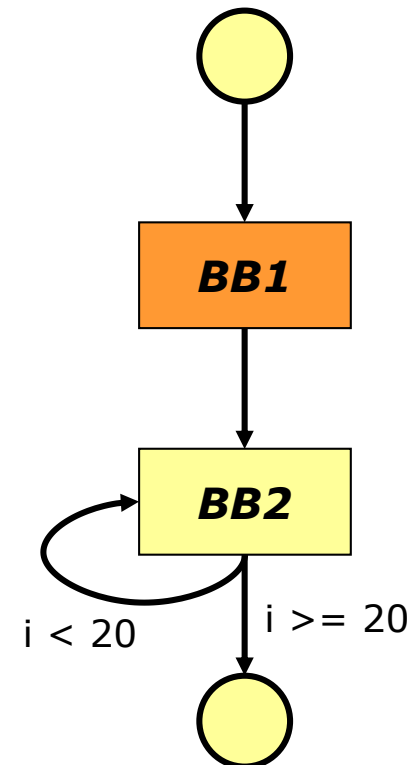
```
(1)  prod := 0
(2)  i := 0
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t4 * t2
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

three-address code

```
(1)  prod := 0      BB1
(2)  i := 0
```

```
(3)  t1 := 4 * i      BB2
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t4 * t2
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

basic blocks



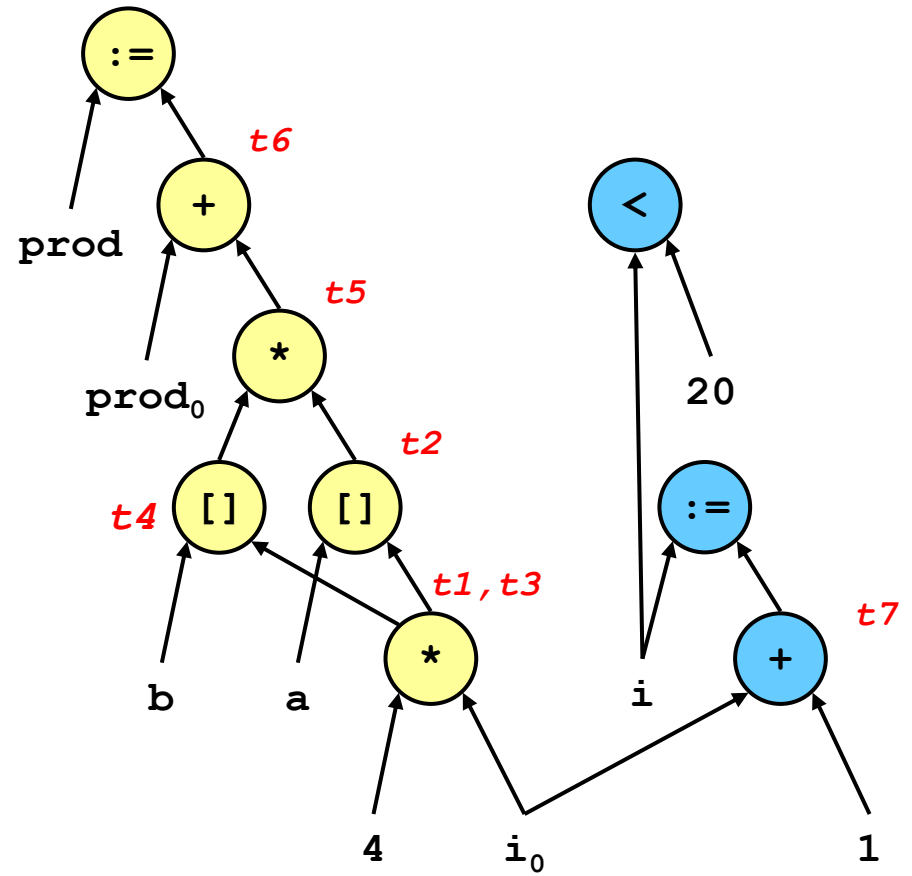
CFG

Example (III)

```

(3)  t1 := 4 * i    BB2
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t4 * t2
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
  
```

basic block 2



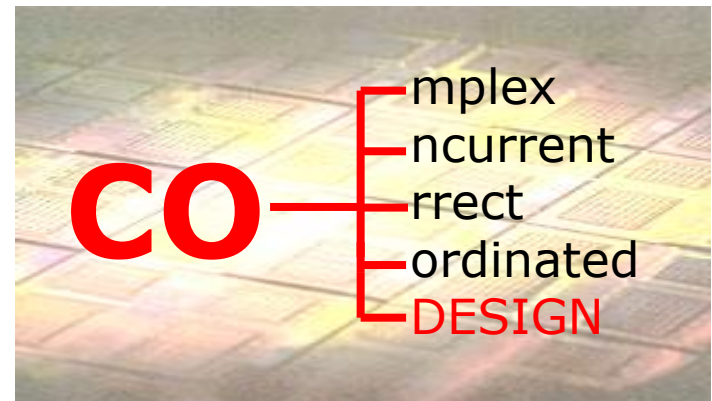
DAG for BB2

Contents

- Compiler Structure

- Code Generation → **Basics**

- Code Optimisation
- Code Generation for
Specialised Processors
- Retargetable Compilers



Machine Model (I)

- set of n registers $\{R_0, R_1, \dots, R_{n-1}\}$
- byte addressable, word = 4 byte
- instruction format

– e.g. **MOV R0, a**
 ADD R1, R0

- each instruction has cost of 1
- addresses of operands follow the instructions (in the code)

Machine Model (II)

- addressing modes

mode	form	address	added cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + <i>contents</i> (R)	1
indirect register	*R	<i>contents</i> (R)	0
indirect indexed	*c(R)	<i>contents</i> (c+ <i>contents</i> (R))	1
immediate	#c	c	1

Code Generation

- code generation = _____
 - allocation:
 - mostly the components are fixed (e. g. target CPU)
 - binding:
 - register binding (allocation/ assignment)
 - instruction selection
 - scheduling:
 - instruction sequencing
- requirements
 - correct code
 - efficient code
 - efficient generation

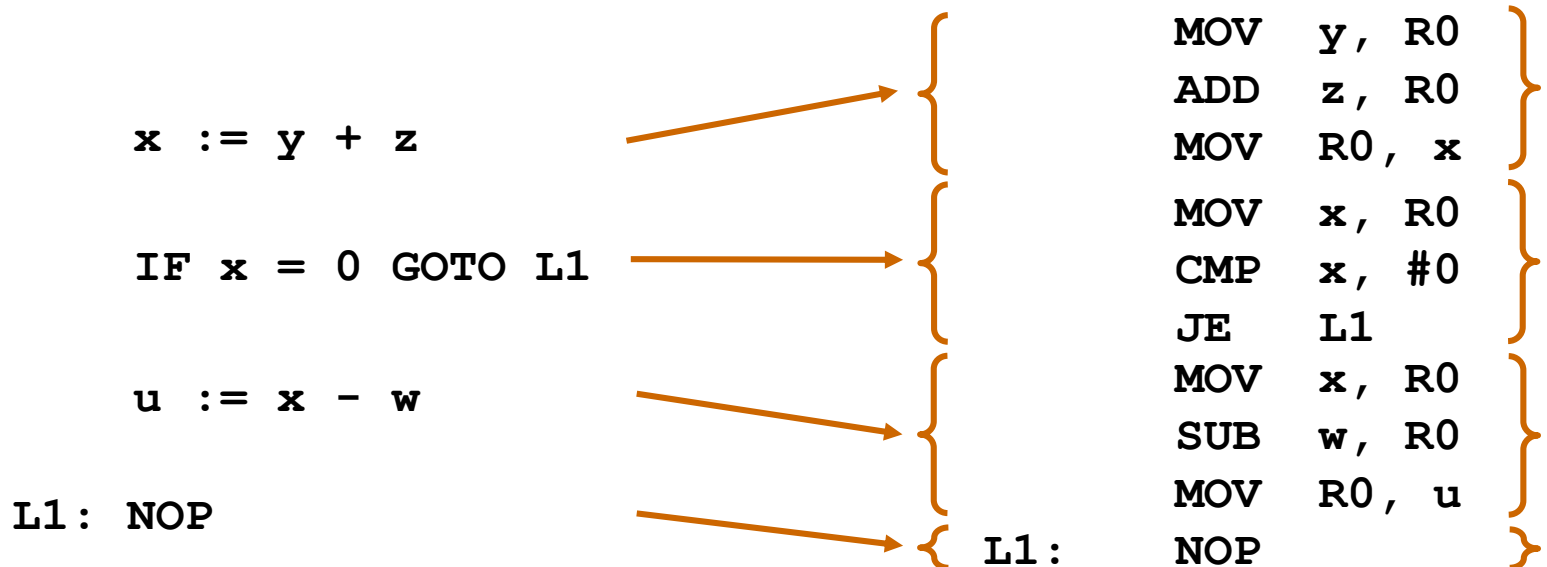
Register Binding

- goal: efficient usage of available registers
 - instructions with register operands are generally shorter and faster than instructions with memory operands (important with CISC)
 - minimize number of LOAD/STORE instr. (important with RISC)
- ---

 - determine the set of variables that should be held in registers for each time
- register assignment
 - assign these set of variables to the available registers
- **optimal** register binding
 - NP-hard problem
 - consider additional restrictions for register use by the processor architecture, compiler or operating system

Instruction Selection

- code patterns for each three-address instruction

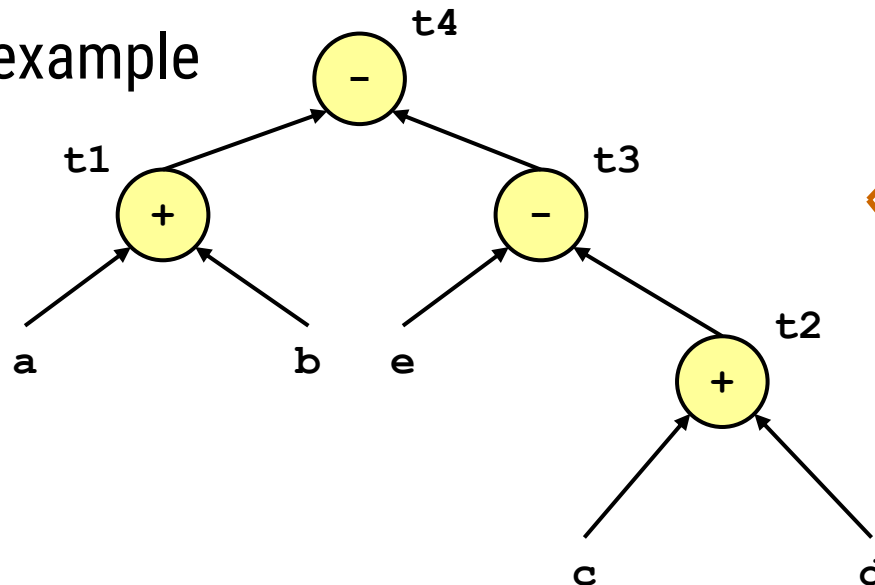


- problems:
 - often inefficient code generated → *code optimisation*
 - several matching target instructions
 - some instructions only with _____
 - exploitation of special processor features (MMX, ...)

Scheduling (I)

- goal: optimal instruction sequence
 - _____ of instructions for the given number of registers
 - NP-hard problem

- example



$t2 := c + d$
 $t3 := e - t2$
 $t1 := a + b$
 $t4 := t1 - t3$
code 1

$t1 := a + b$
 $t2 := c + d$
 $t3 := e - t2$
 $t4 := t1 - t3$
code 2

Scheduling (II)

(2 registers available)

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

↓ **code 1**

```
(1) MOV  c, R0
(2) ADD  d, R0
(3) MOV  e, R1
(4) SUB  R0, R1
(5) MOV  a, R0
(6) ADD  b, R0
(7) SUB  R1, R0
(8) MOV  R0, t4
```

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

↓ **code 2**

```
(1) MOV  a, R0
(2) ADD  b, R0
(3) MOV  c, R1
(4) ADD  d, R1
(5) MOV  R0, t1
(6) MOV  e, R0
(7) SUB  R1, R0
(8) MOV  t1, R1
(9) SUB  R0, R1
(10) MOV R1, t4
```

Life Time of Variables (I)

- **definition**

$S = \{S_1, \dots, S_n\}$... set of three address instructions

$G = (V, E)$... control flow graph

S_i writes a variable x , S_j reads x

if a path (S_i, S_j) from S_i to S_j exists in G without rewriting x
 $\rightarrow x$ is **alive** in every instruction line S_k of path (S_i, S_j)

life time of variable x = set of all nodes where x is alive

- **definition**

A name (variable) is **active** at a certain point inside a basic block if its value is used afterward – possibly in another basic block.

Life Time of Variables (II)

- states of a variable x in a instruction line i :
 - _____ : x is read (used) after line i
 - _____ : x is not read after line i
 - _____ : line number of next usage of x
- determining life times
 1. go to the end of the BB and find the variables which must be active at the block exit
 2. go back to the block entry instruction by instruction
for each instruction (I) $\mathbf{x} := \mathbf{y} \text{ op } \mathbf{z}$
 - if \mathbf{x} is **INACTIVE**, remove the instruction (I)
 - else set \mathbf{x} to **INACTIVE** and **NEXT-USE** to *none*
 - set \mathbf{y} , \mathbf{z} to **ACTIVE** and **NEXT-USE** to (I)

Simple Code Generator (I)

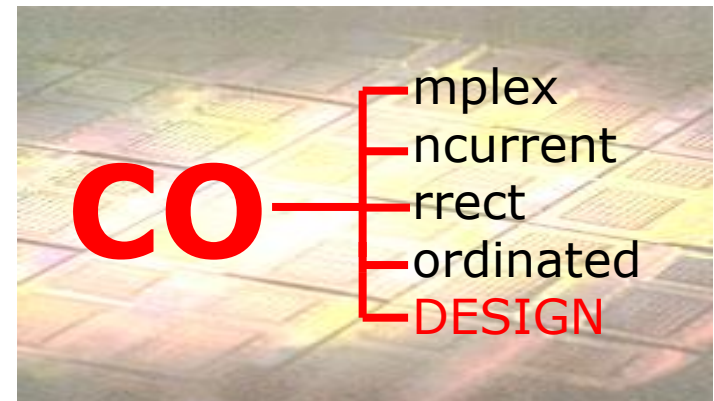
- for each three-address instruction $x := y \text{ op } z$
 1. call `getreg(x)` to determine the location L , where the result x will be written
 2. determine y' – the current location of y
 3. if $y' \neq L$, generate **MOV y' , L**
 4. determine z' – the current location of z
 5. generate **op z' , L**
- current location of variable t
 - _____
 - if t is in memory **and** register, prefer the register

Simple Code Generator (II)

- **L = getreg(x)** for the instruction **x := y op z**
 1. if **y** is stored in a register **R**, which is not used by another variable, and **y** has no **NEXT-USE**: **return(R)**
 2. if an empty register **R** exists: **return(R)**
 3. if **x** has a **NEXT-USE** in the BB or **op** is an operator which needs a register (e.g. indirect addressing), find a used register **R**, store the register content to memory using **MOV R,M** and **return(R)**
 4. if **x** has no further usage in the basic block or no feasible register could be found: **return(M_x)**

Contents

- Compiler Structure
- Code Generation → Register Binding
- Code Optimisation
- Code Generation for Specialised Processors
- Retargetable Compilers



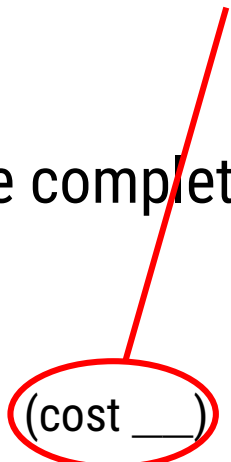
Register Allocation

- global register allocation
 - determine number of _____
 - for global variables
 - for loop variables
 - for variables in basic blocks
 - user-defined register allocation
 - e. g. in programming language C: `register int i;`
- algorithms for optimised register allocation
 - usage counters (loops)
 - graph colouring (inside basic blocks)

Allocation by Usage Counters (I)

- a loop **L** consists of several basic blocks
- if a variable **a** is kept in a register during the complete execution of **L**, costs can be saved:
 - ___ cost unit for each use of **a**
 - **ADD R0, R1** (cost ___) instead of **ADD a, R1** (cost ___)
 - ___ cost units at the end of the basic block, if **a** has been defined in the basic block and is active afterwards
 - no save instruction **MOV R0, a** (cost ___)

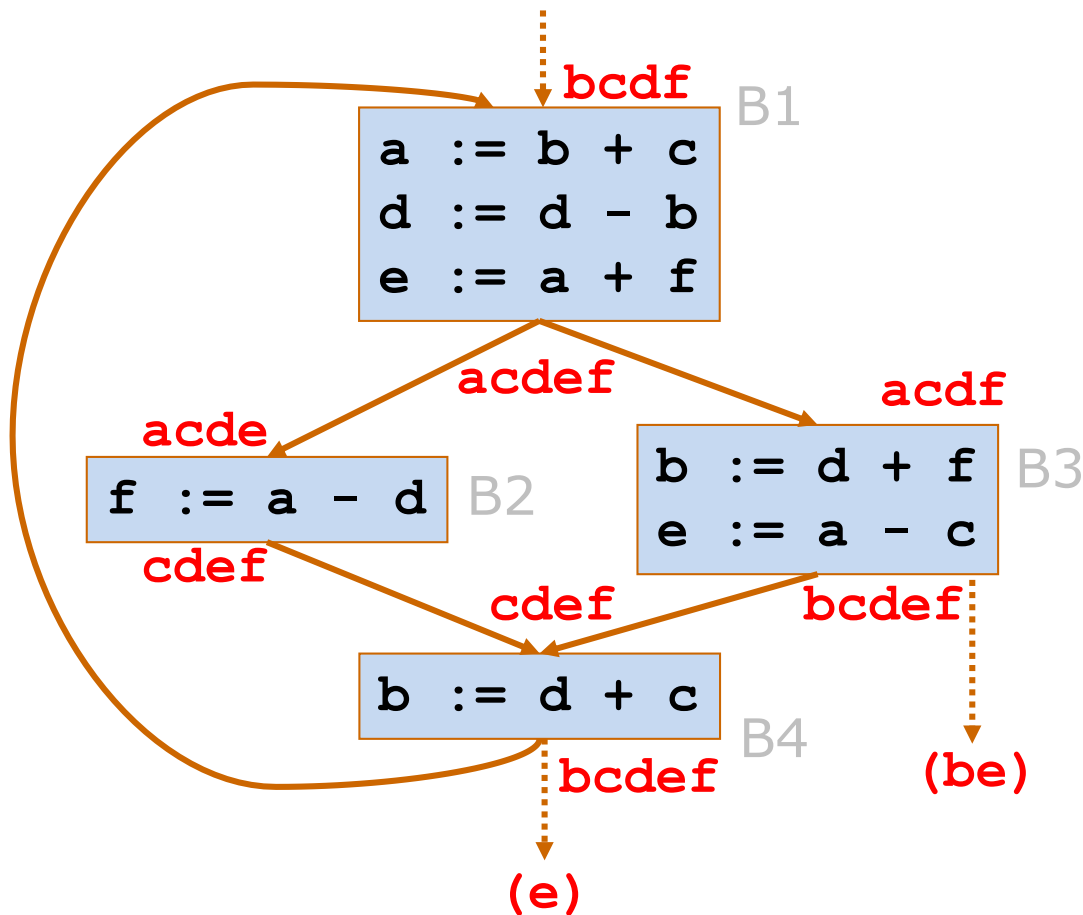
*see slide 24
(added cost)*



Allocation by Usage Counters (II)

- calculate cost savings for the loop
-
- ***uses*** (a, B) counts the number of usages of a in basic block B **before** a is defined
 - ***active*** (a, B) is '1', if a has been **defined in basic block** B and is active at the end of the basic block, otherwise '0'
- approximation, because of assumption
 - all basic blocks in L are executed equally often
 - L is executed for many times

Example



<code>active(a, B1)</code>	<code>= 1 (*2)</code>
<code>uses(a, B1)</code>	<code>= 0</code>
<code>uses(a, B2)</code>	<code>= 1</code>
<code>uses(a, B3)</code>	<code>= 1</code>
<code>uses(a, B4)</code>	<code>= 0</code>
<i>cost red.</i>	<i>= 4</i>

cost reduction:

a: 4
 b: 6
 c: 3
 d: 6
 e: 4
 f: 4

3 registers available:

b → R0
 d → R1
 a → R2

Allocation by Graph Colouring

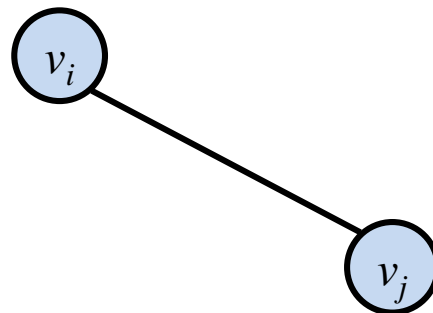
- algorithm
 1. code generation with **unlimited number** of registers (symbolic registers) → each variable name is a symbolic register
 2. **life time** determination of the variables (symbolic registers)
 3. construction of the **register conflict graph**
 4. mapping of the symbolic registers to physical registers by **graph colouring**

Register Conflict Graph

- **definition**

A register conflict graph $G=(V,E)$ is an undirected graph, where the nodes V represent the variables (symbolic registers) and the edges E represent the conflicts between the variables.

An edge between $v_i, v_j \in V$ indicates that the _____
of v_i and v_j overlap. Thus, v_i and v_j **cannot** share a register.

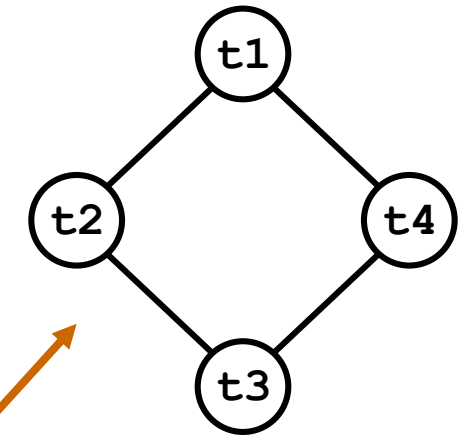
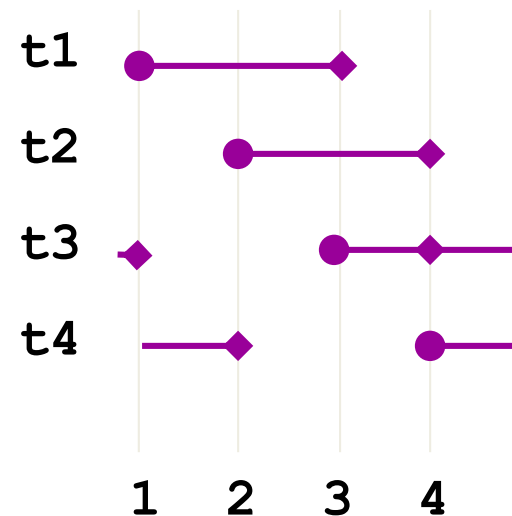


Example

loop body (BB)

```
(1)  t1 := t3 * 10
(2)  t2 := t4 * 20
(3)  t3 := t1 + 5
(4)  t4 := t2 + t3
```

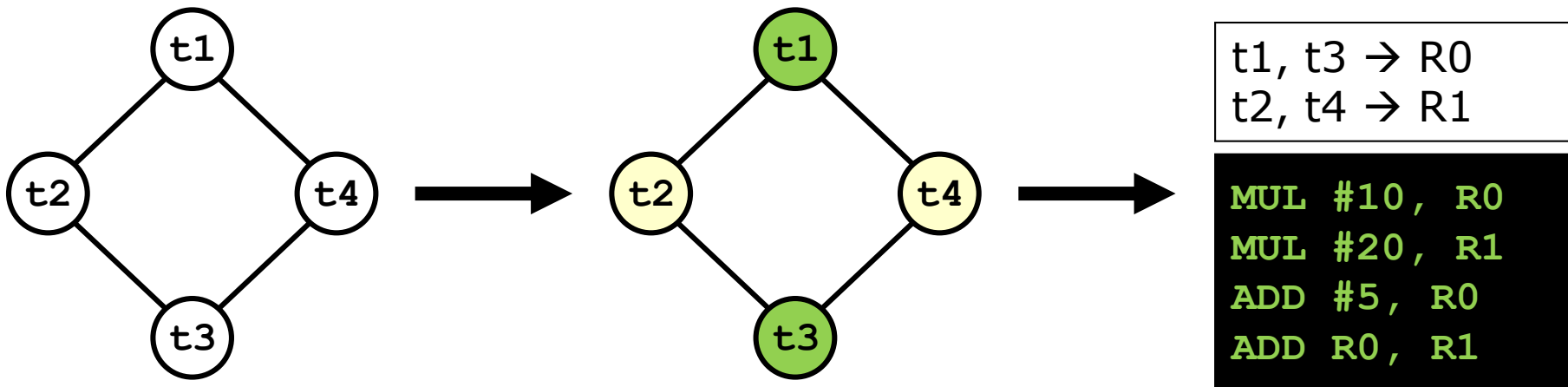
life times



register conflict graph

Graph Colouring (I)

- colour the nodes of G with l colours in a way that no two adjacent nodes get the same colour (NP-complete problem)

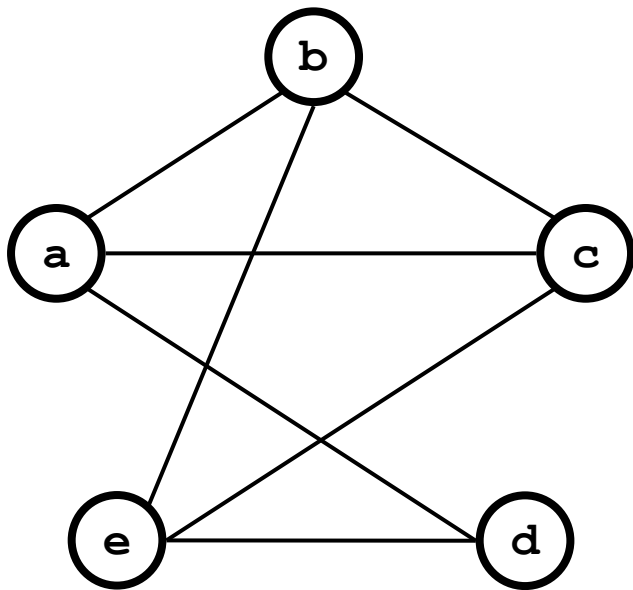


Graph Colouring (II)

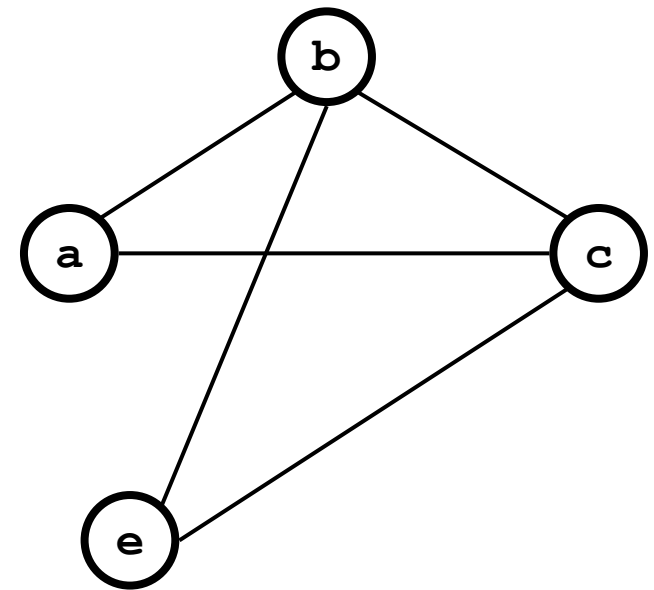
- **heuristic:** can a graph $G = (V, E)$ be coloured with l colours?
 - algorithm
 1. find a node v_i in G with $\text{deg}(v_i) < l$
 2. create $G' = (V', E')$ by removing v_i and all its edges
 3. if _____ then l -colouring is possible, **return true**
else
if all nodes v_i in G' have a $\text{deg}(v_i) \geq l$ then
 l -colouring not possible, **return false**
else $G = G'$, **goto 1**

Example (I)

$k = 3$

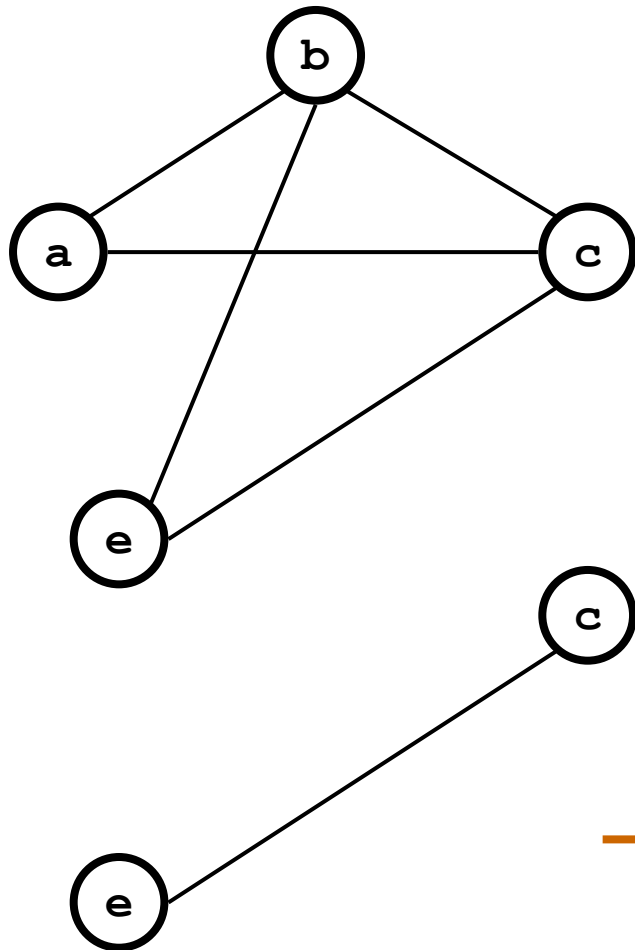


remove d →

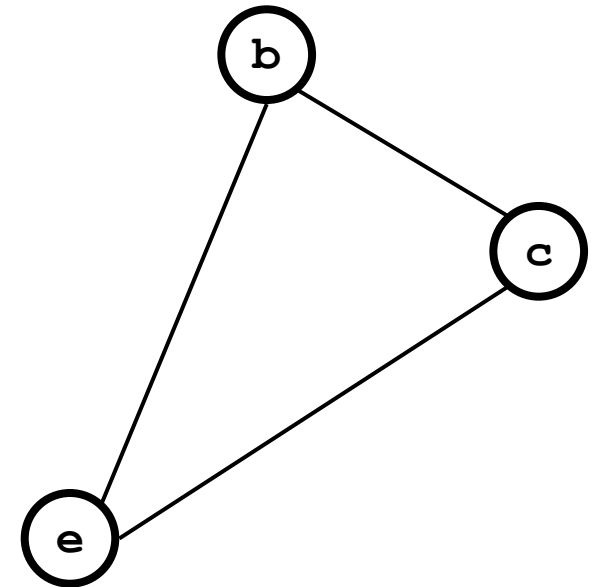


Example (II)

$k = 3$



remove a



remove b



remove c



remove e



$\{\}$

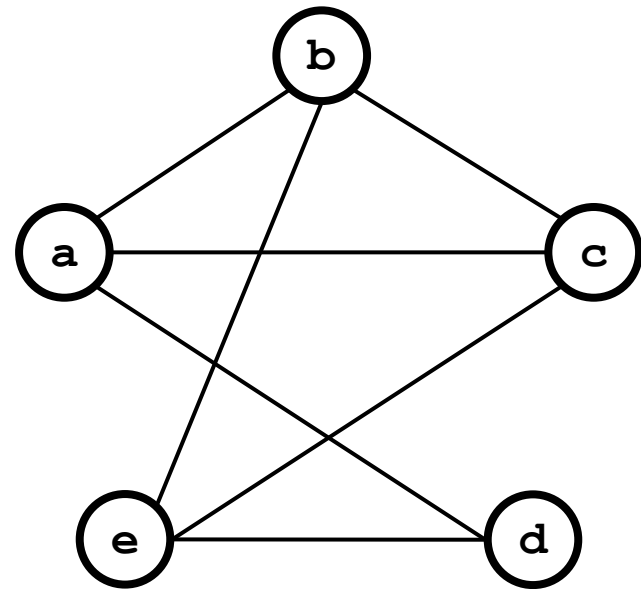
Example (III)

$k = 3$



sequence of removed nodes:

$d \rightarrow a \rightarrow b \rightarrow c \rightarrow e$

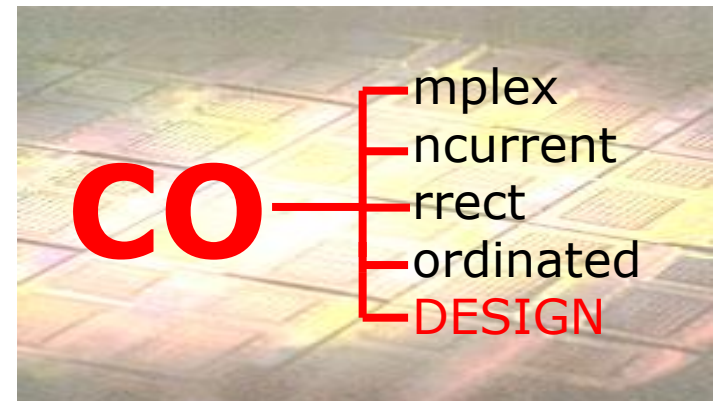


Problem: heuristic is sensitive against _____
→ see exercise

Contents

- Compiler Structure
- Code Generation
- Code Optimisation
- Code Generation for Specialised Processors
- Retargetable Compilers

→ Scheduling



Classification

- static / dynamic:
 - scheduling during design- or runtime?
- _____ :
- tasks interruptible?
- with / without resource restrictions:
 - scheduling influenced by resources?
- aperiodic / periodic (iterative)
 - scheduling calculable?

Scheduling without Resource Constraints

- ASAP (As Soon As Possible)
 - determines earliest possible start times of tasks
 - result: minimal latency
- ALAP (As Late As Possible)
 - determines latest possible start times of tasks for a given latency
- _____ M of a task:
 - difference of start times: $M = time(ALAP) - time(ASAP)$
 - $M = 0 \rightarrow$ task is part of critical path

Example

- equations (one program)

$$x = ((a \cdot b) + (b \cdot c) - d) - (c \cdot d \cdot f)$$

$$y = (d \cdot e) + c$$

$$z = (e + f) \ll 2$$

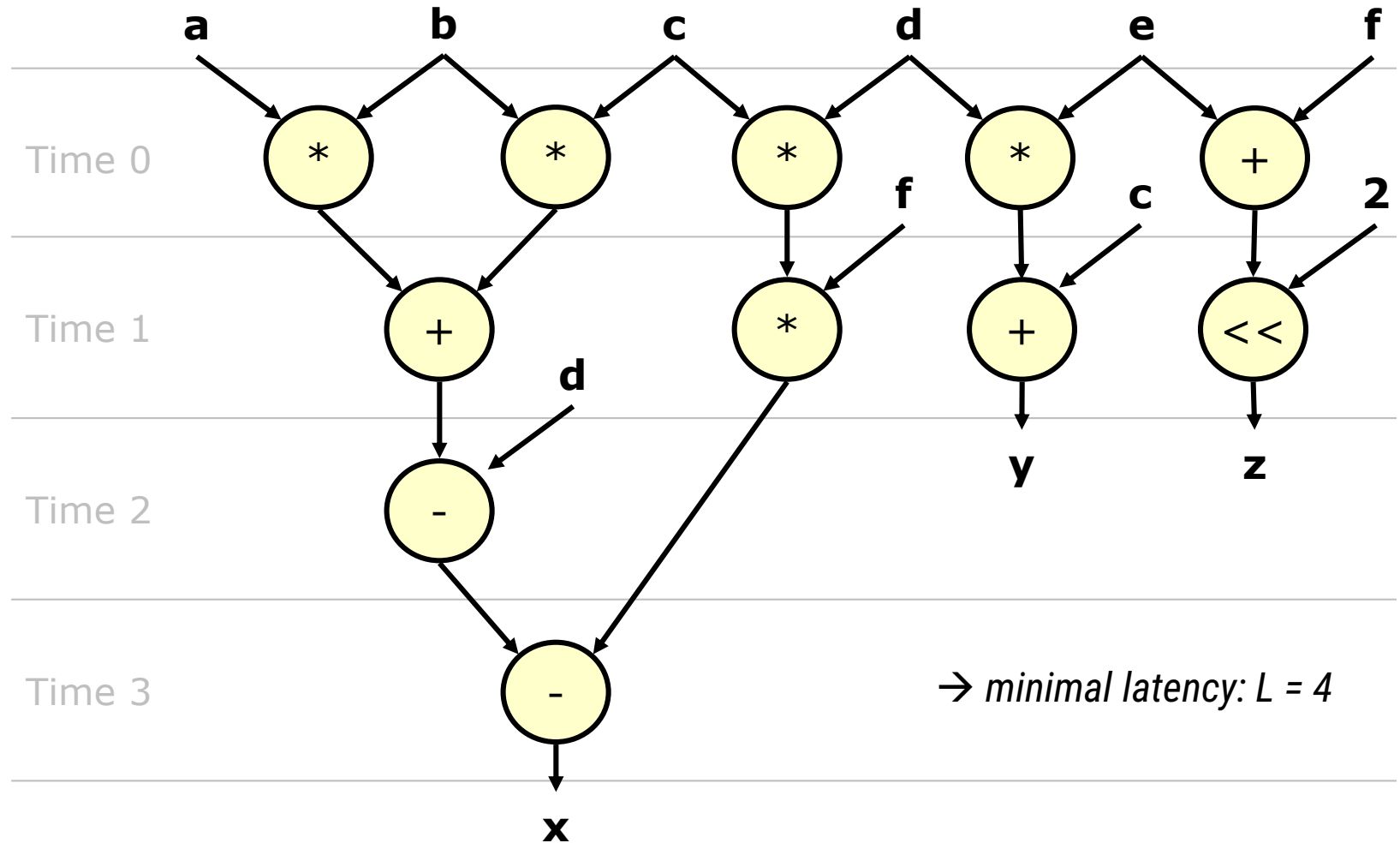
- three-address code

```
(1)  t1  := a  *  b
(2)  t2  := b  *  c
(3)  t3  := t1  +  t2
(4)  t4  := t3  -  d
(5)  t5  := c  *  d
(6)  t6  := t5  *  f
(7)  x   := t4  -  t6
```

```
(8)  u1  := d  *  e
(9)  y   := u1  +  c
```

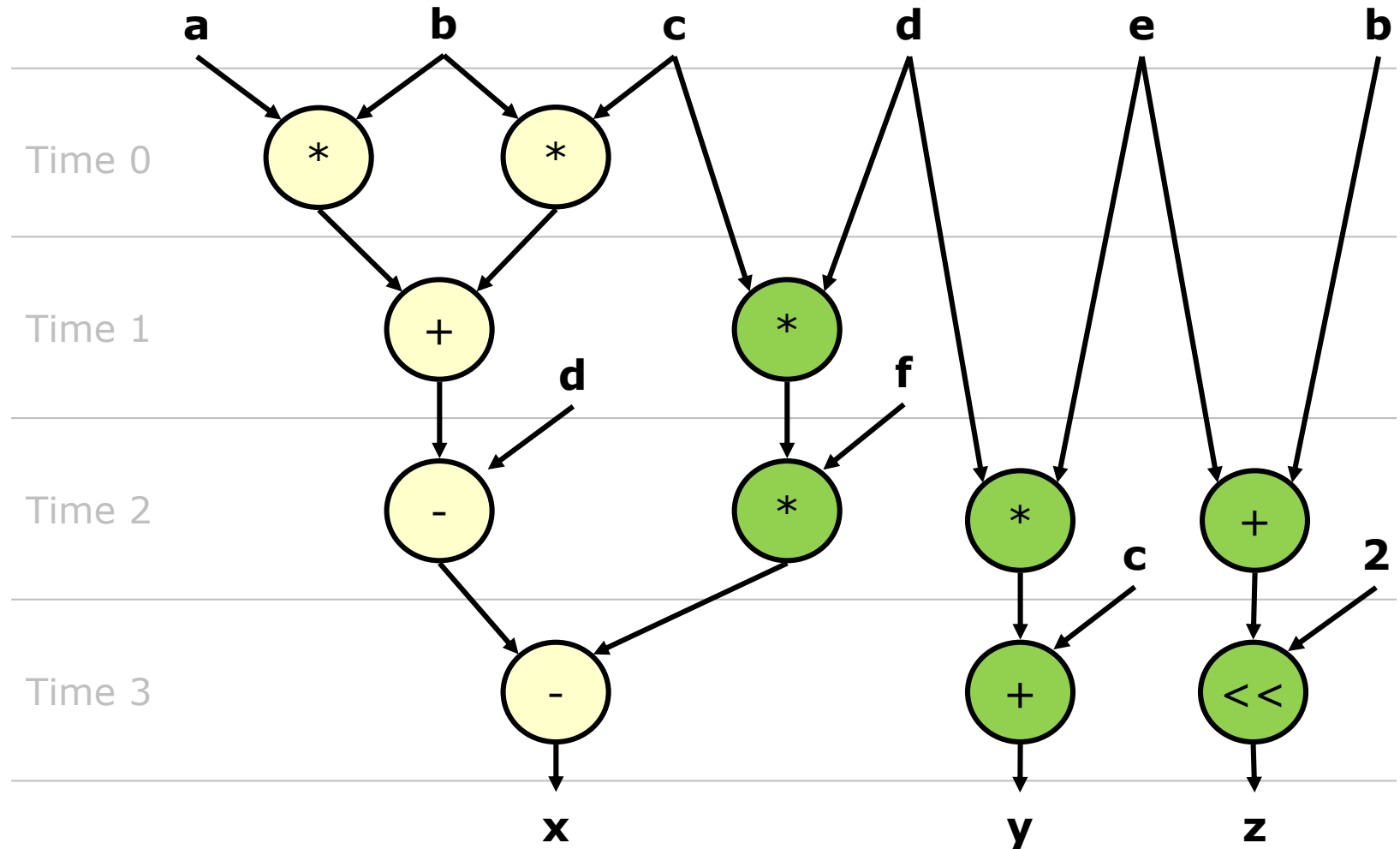
```
(10) v1  := e  +  f
(11) z   := v1  << 2
```

ASAP

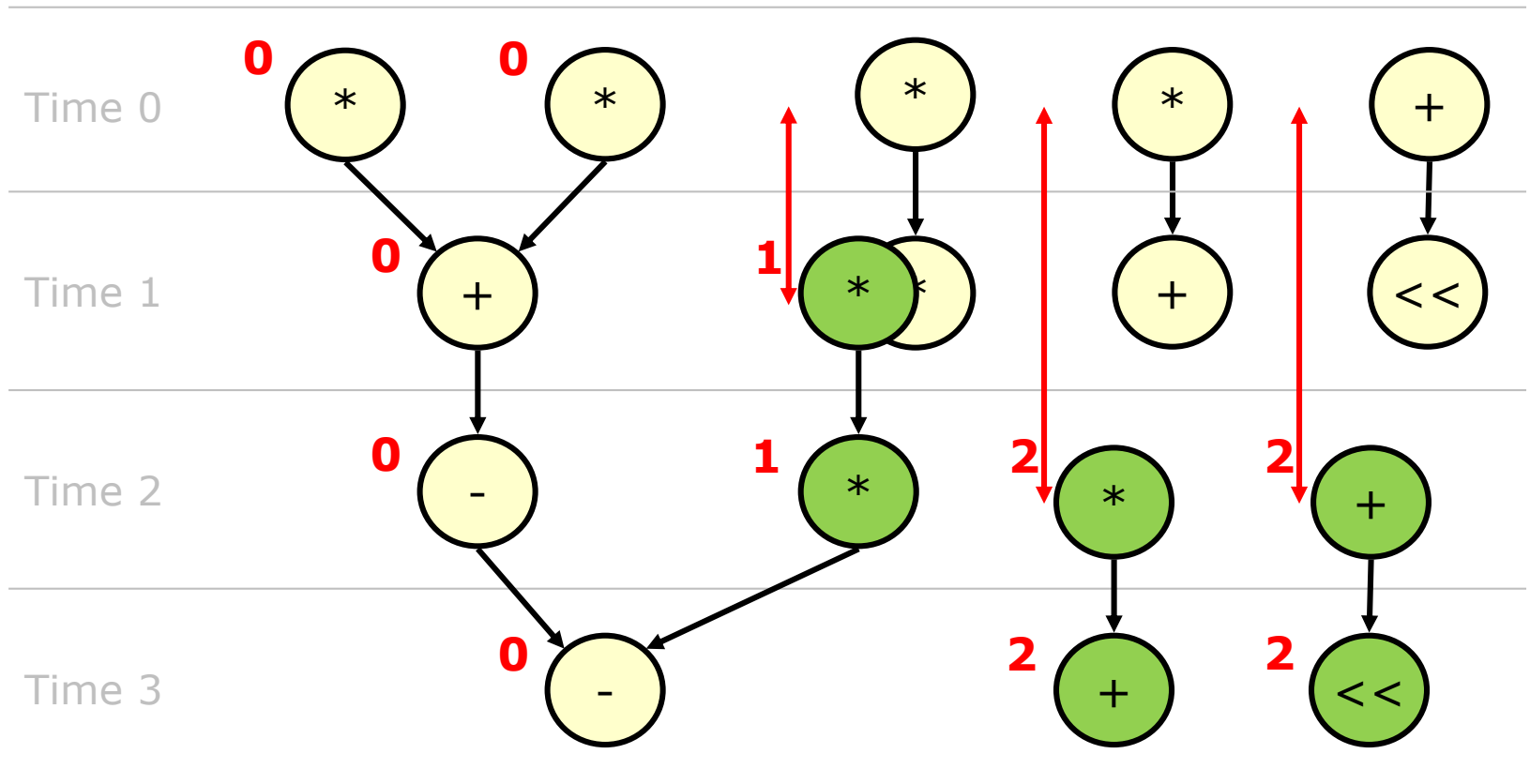


ALAP

→ latency bound: $L = 4$



Mobility

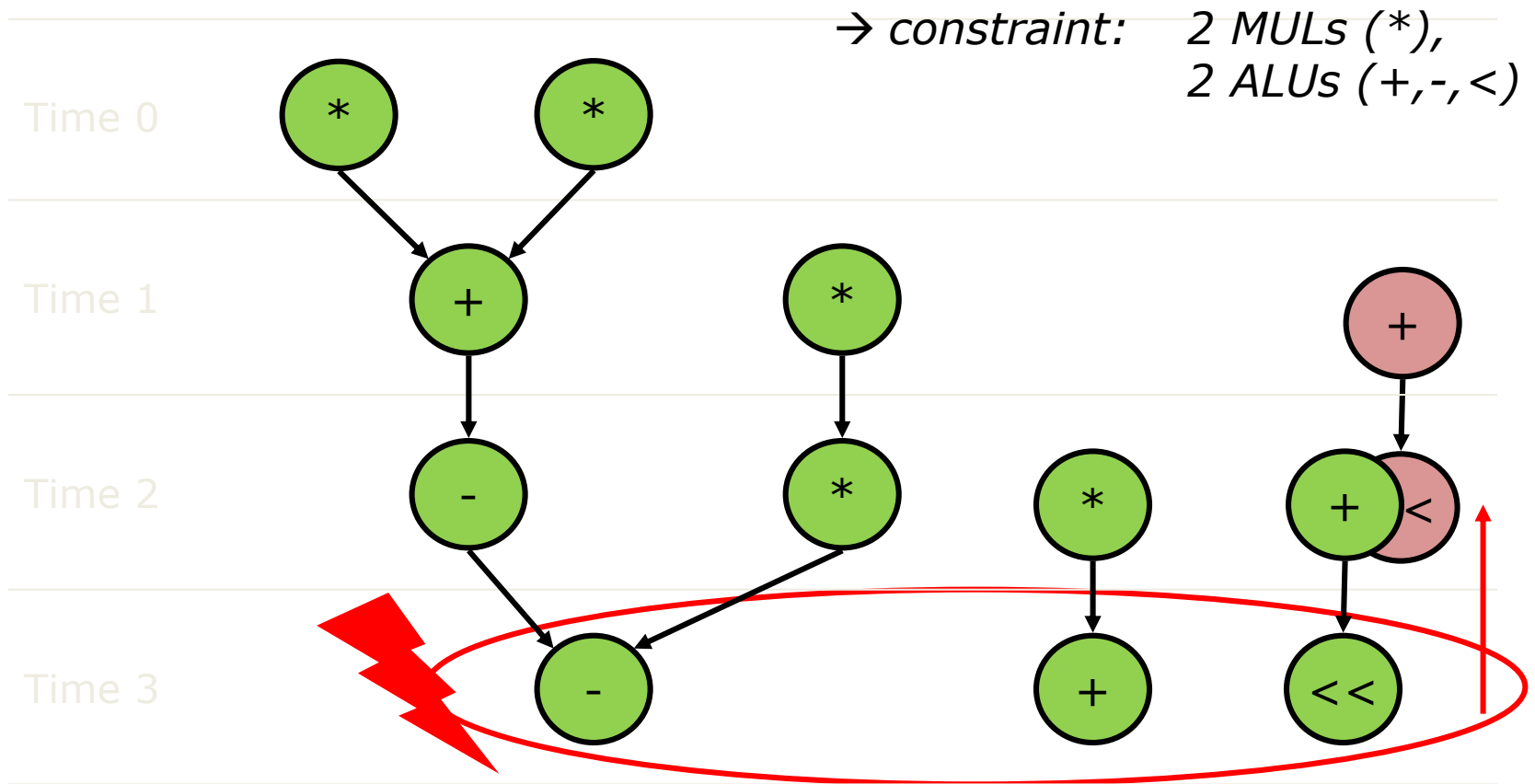


Scheduling under Resource Constraints

- extended ASAP/ ALAP
 - calculate ASAP or ALAP
 - move tasks down (ASAP) or up (ALAP) until resource constraints are satisfied
- ---

 - operations are prioritised depending on a given criteria (e. g. number of children nodes, mobility, ...)
 - assign to each free resource in each time step the task which is executable and has the highest priority

Extended ALAP

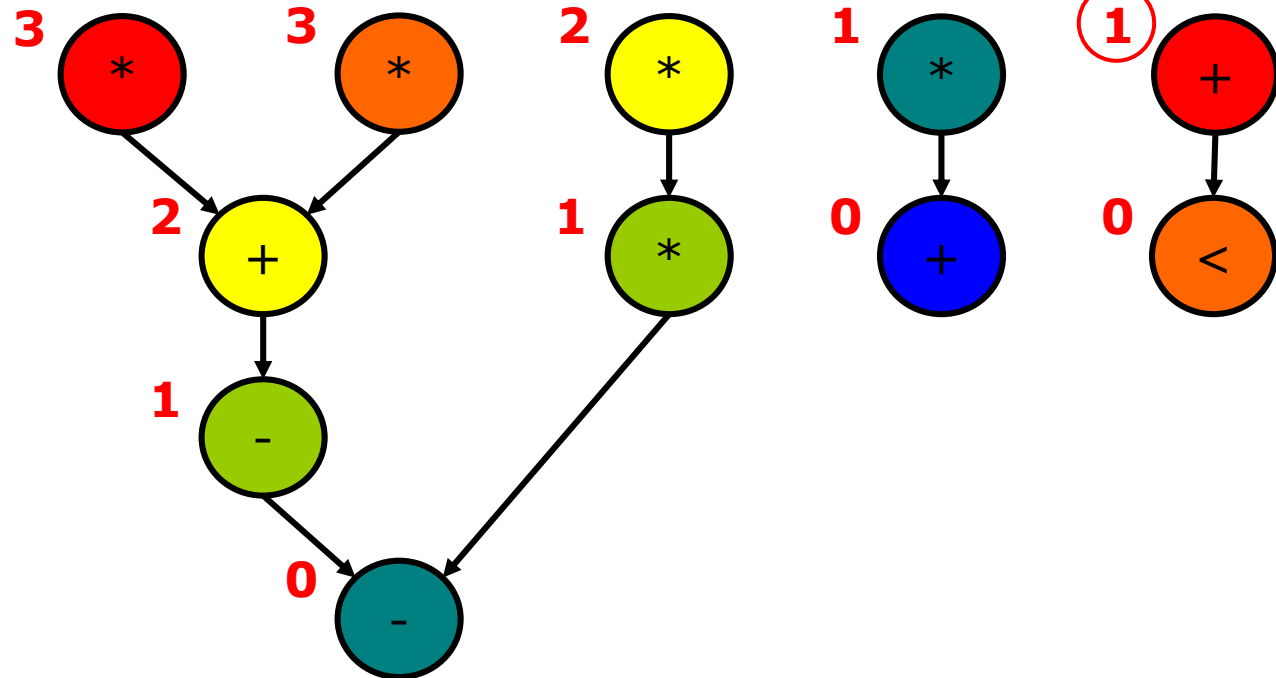


List Scheduling

- example
 - prioritisation criterion: number of children nodes
 - resource constraints: 1 MUL (*), 1 ALU (+,-,<)

exec. time

1
2
3
4
5
6



Periodic Scheduling

- 2 execution units ($r1, r2$)

