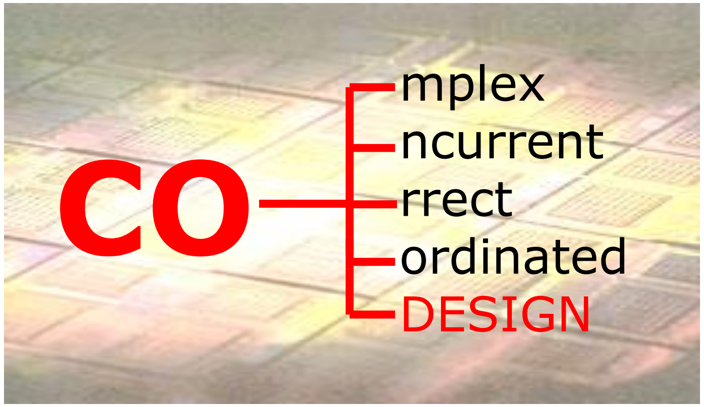
Contents

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

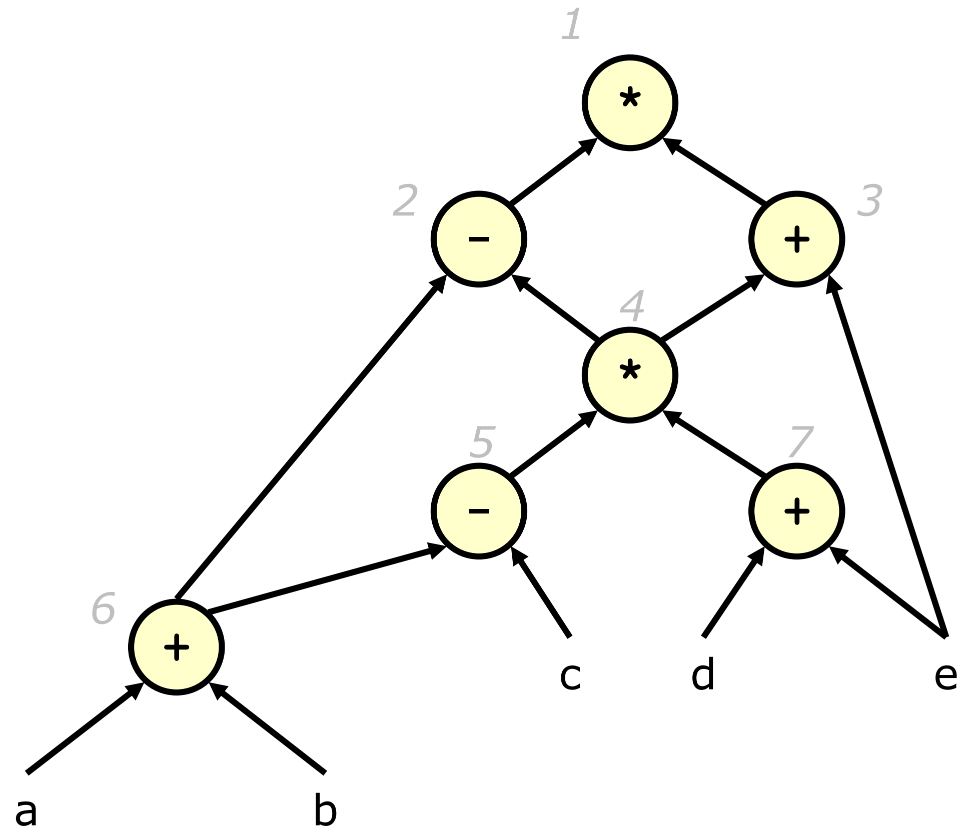


Code Generation for DAGs (I)

- node computation order of a DAG has great impact on the number of required instruction

- **heuristic:** node computation order
  1. schedule the root
  2. select a node n with already scheduled parents and schedule it
  3. as long as the left-most child node m of n has no unscheduled parents and is no leaf
     1. schedule m
     2. n ← m
     3. goto 3
  4. goto 2

Example



scheduling order: **1 2 3 4 5 6 7**

Code Generation for DAGs (II)

- if a DAG with *n* nodes **builds a tree**, there exists an algorithm that generates ______________________________
  (dynamic programming)

- DAG is not a tree, if common sub expressions exists

  1. split the DAG into trees a nodes which express common sub expressions
  2. generate optimal code for each tree → not necessarily optimal for the DAG

Dynamic Programming (I)

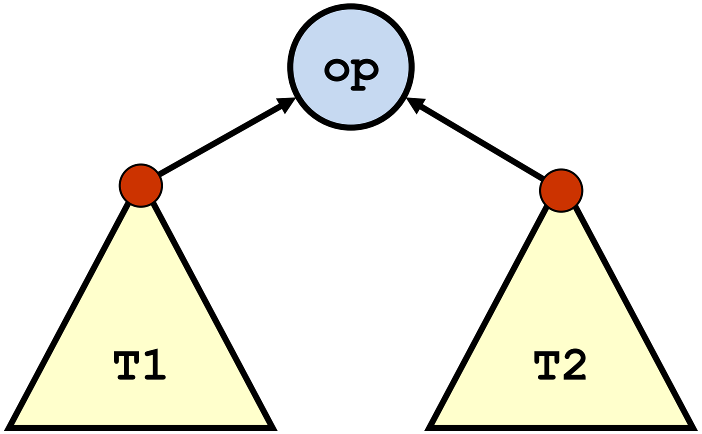
- machine model extended to complex instructions
  - set of n registers $\{R_0, R_1, \ldots, R_{n-1}\}$
  - instruction `Ri := E`

    E is an expression with arbitrarily many registers and memory addresses
  - if E contains more than one register, R_i has to be one of it
  - load: `Ri := M`, store: `M := Ri`, copy: `Ri := Rj`
  - simplification: all instructions (all addressing modes) have cost `'1'`
- examples

machine model	→	***extended machine model***
`ADD R0, R1`	→	`R1 := R1 + R0`
`ADD *R0, R1`	→	`R1 := R1 + ind R0`
`SUB a, R0`	→	`R0 := R0 - a`

Dynamic Programming (II)

- principle of dynamic programming applied to code generation for DAGs



optimal code for
`E := T1 op T2`

1. optimal code for **`T1`** and **`T2`**
2. optimal code for **`E`**:
   - calculate **`T1`**, than **`T2`**, execute **`T1 op T2`**
   - calculate **`T2`**, than **`T1`**, execute **`T1 op T2`**

Dynamic Programming (III)

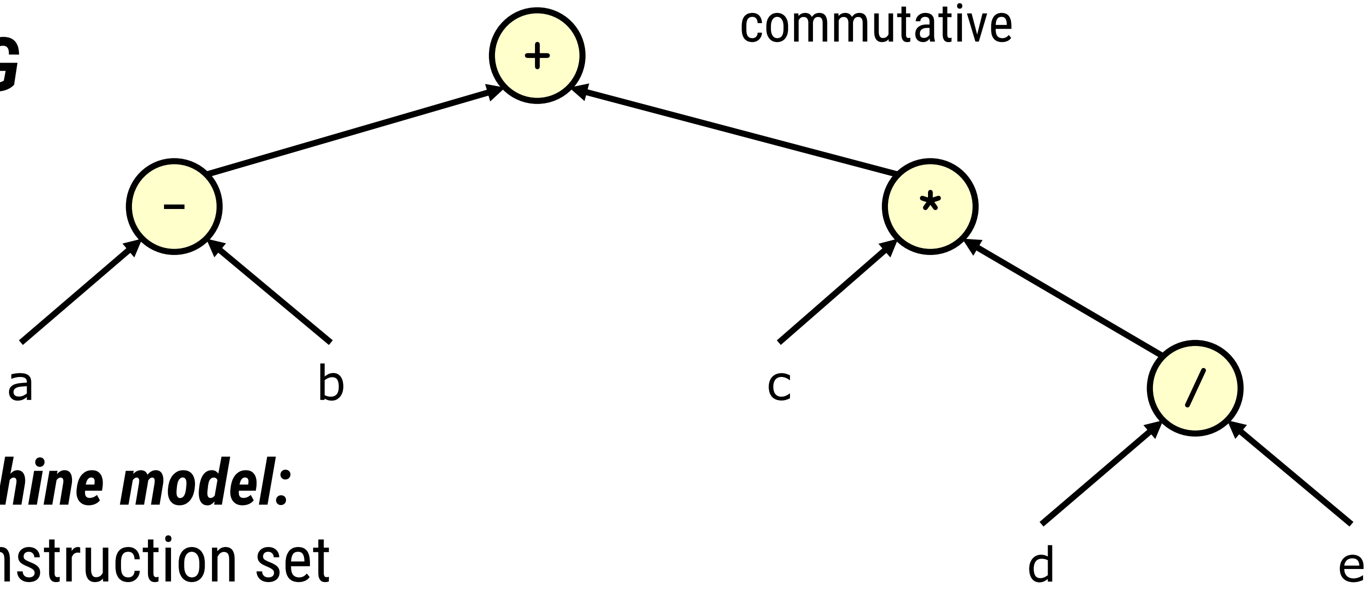
- method has 3 phases
  - computation of ____________________
  - determination of the ____________________
  - generation of ____________________

- computation of cost vectors for each node n (bottom up):
  - $C[i]$ optimal cost for computing n with i registers
  - $C[0]$ optimal cost for computing n, if the result is stored in memory

Example (I)

Assumption:
All operations / operators are **NOT** commutative

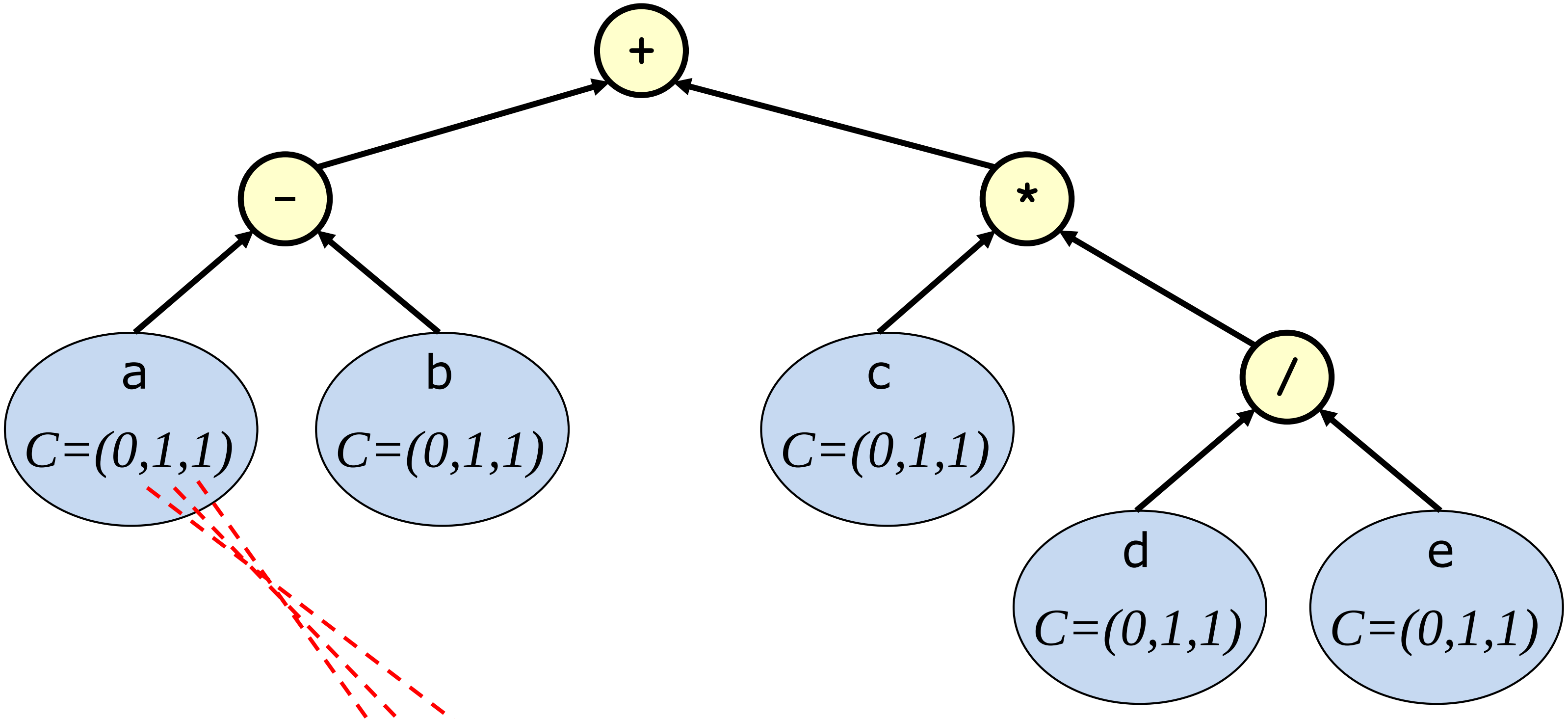
DAG



machine model:

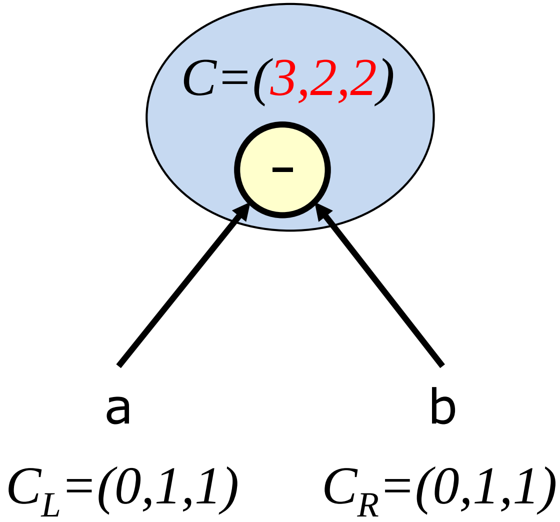
- instruction set
  - `Ri := Mj`
  - `Ri := Ri op Rj`
  - `Ri := Ri op Mj`
  - `Ri := Rj`
  - `Mj := Ri`
- two registers available

Example (I)



- save in memory $C[0] = 0$ (a is already in memory)
- usage of 1 register $C[1] = 1$ (`Ri := M`)
- usage of 2 registers $C[2] = 1$ (see $C[1]$)

Example (III)



- usage of 1 register: **`Ri := Ri – M`**
  - left sub tree with 1 register
  - right sub tree into memory

  → $C = C_L[1] + C_R[0] + 1 = 2 = C[1]$

  costs for operation '-'

- usage of 2 registers:
  - **`Ri := Ri – M`**

  → $C = C[1] = 2 = C[2]$

  - **`Ri := Ri – Rj`**

  → left sub tree with 2 registers
  right sub tree with 1 register:
  $C = C_L[2] + C_R[1] + 1 = 3$

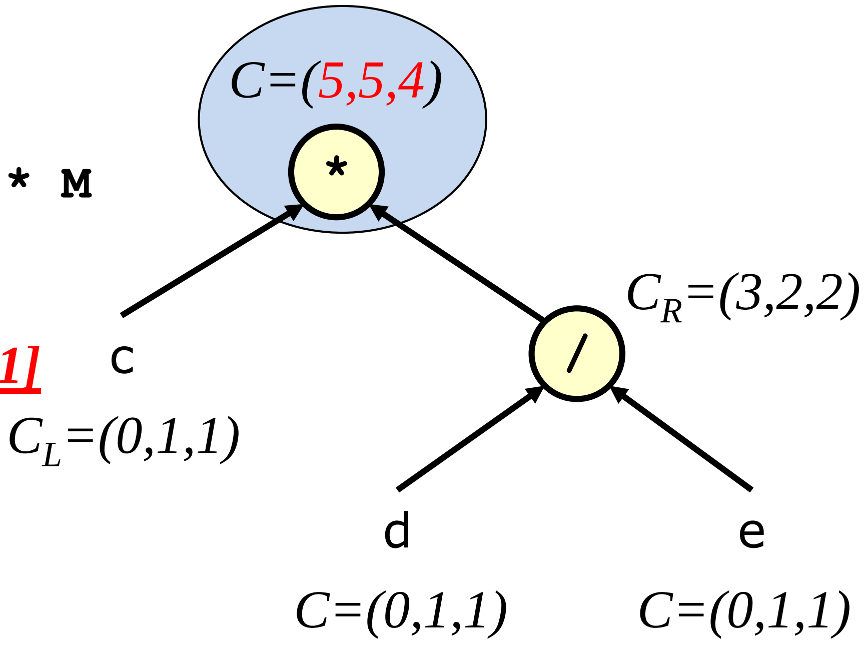
  OR

  right sub tree with 2 registers,
  left sub tree with 1 register:
  $C = C_L[1] + C_R[2] + 1 = 3$

- store in memory:

  → $C = \min(C[1], C[2]) + 1 = 3 = C[0]$

Example (IV)



- usage of 1 register: `Ri := Ri * M`
  - left sub tree with 1 register
  - right sub tree into memory

  → $\underline{C = C_L[1] + C_R[0] + 1 = 5}$ $\underline{= C[1]}$
- usage of 2 registers:
  - `Ri := Ri * M`

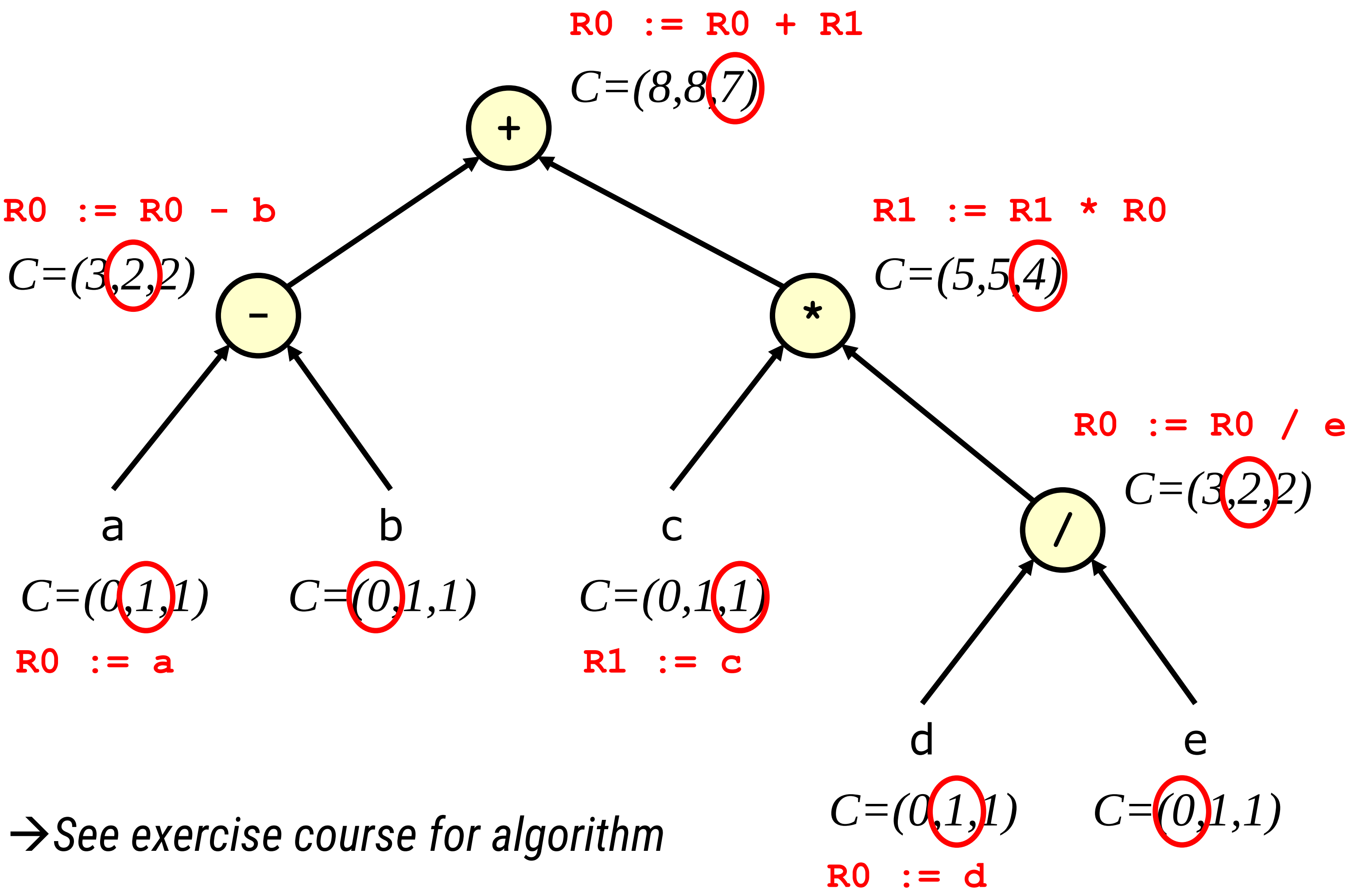
  → $\underline{C = C[1] = 5}$

  - `Ri := Ri * Rj`

  → left sub tree with 2 registers OR right sub tree with 2 registers,
  right sub tree with 1 register: left sub tree with 1 register:
  $\underline{C = C_L[2] + C_R[1] + 1 = 4}$ $\underline{= C[2]}$ $\underline{C = C_L[1] + C_R[2] + 1 = 4}$
- store in memory:

  → $\underline{C = \min(C[1], C[2]) + 1 = 5}$ $\underline{= C[0]}$

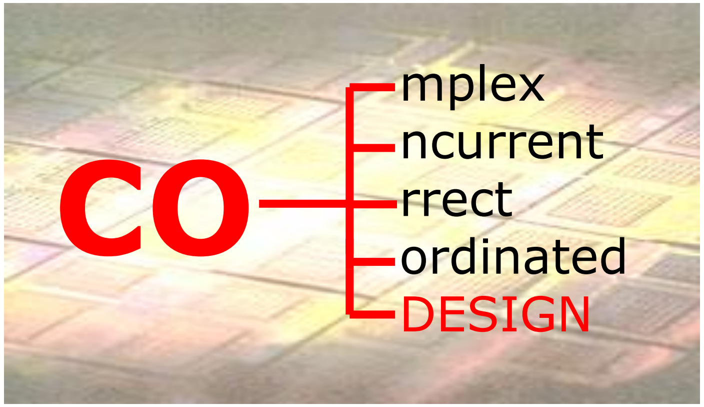
Example (V)



→*See exercise course for algorithm*

Contents

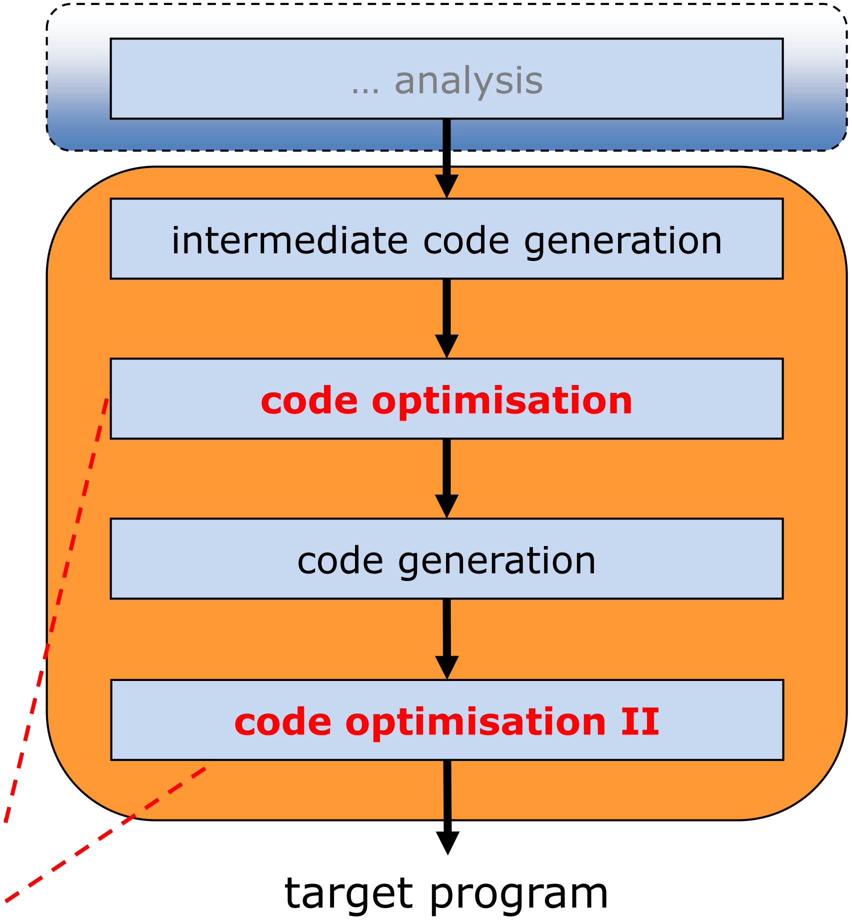
- Compiler Structure
- Code Generation
- Code Optimisation
- Code Generation for Specialised Processors
- Retargetable Compilers



Code Optimisation

- *remember:* code generation requirements
  - ______________code
  - ______________code
  - efficient ________________
- **problem:** code generation line by line
  → useless instructions
  → ineffective constructions

→**optimising transformation on the intermediate code and on the target (machine) code**

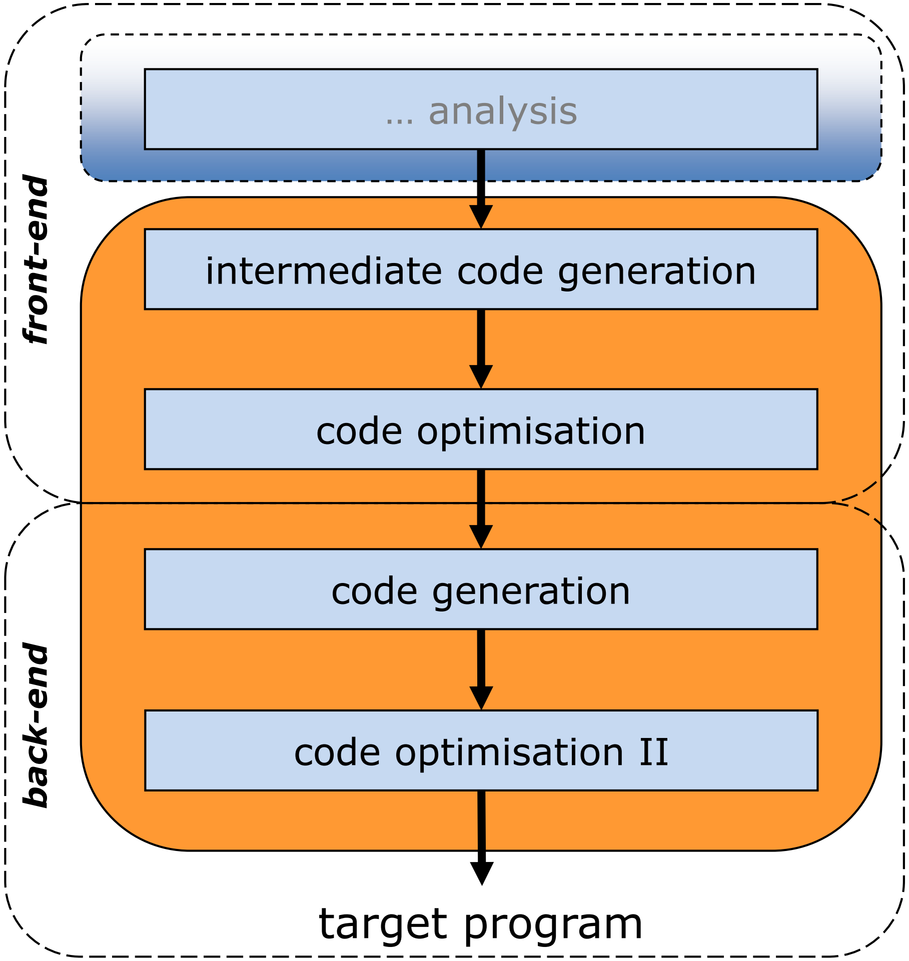


Advantage

- target platform independent
  front-end
  → reusable compiler

- use of knowledge of target platform
  → efficient HW usage

→ ***faster and/or smaller code***



Methods

- optimisation on ___________________ **code**
  - local optimisation
    - transformation inside basic blocks
  - global optimisation
    - transformations across several basic blocks
- optimisation on ________________ **code**
  - peephole optimisation
    - small window (peephole) is "moved" over code
    - iteration, because an optimisation can generate new optimisation opportunities

Local Optimisation (I)

- common sub expression elimination

```
(1)   a := b + c                    (1)   a := b + c
(2)   b := a - d                    (2)   b := a - d
(3)   c := b + c                    (3)   c := b + c
(4)   d := a - d          →         (4)   d := b
```

- instruction interchange

```
(1)   t1 := b + c                   (1)   t2 := x + y
(2)   t2 := x + y         →         (2)   t1 := b + c
```

(e.g. for optimised register usage)

Local Optimisation (II)

- algebraic simplifications

```
(1)  x := y + 0*(z*4/y)  →  (1)  x := y

(2)  y := y * 1          →  (2)  NOP
(3)  z := z + 0          →  (3)  NOP

(4)  a := 4                 (4)  a := 4
(5)  b := a * 5          →  (5)  b := 20
```

- variable renaming

```
(1)  t := b + c          →  (1)  u := b + c
```

normal form of a BB: each variable is defined only once

Global Optimisation (I)

- passive code elimination

  an instruction that defines **x** can be deleted if **x** is not used afterwards

- copy propagation

```
(1)   x := t1
(2)   a[t2] := t3
(3)   a[t4] := x
(4)   goto L
```

→

```
(1)   x := t1
(2)   a[t2] := t3
(3)   a[t4] := t1
(4)   goto L
```

if **x** *is not used after* **(1)***, line* **(1)** *is passive code*

- code motion

```
while (i < (x*4+2))
{...}
```

→

```
t = x*4+2
while (i < t)
{...}
```

if **x** *is not modified in the loop body*

Global Optimisation (II)

- operator strength reduction

```
(1)                                  (1)  j  := n
(2)  j  := n                         (2)  t4 := 4 * j
(3)  j  := j - 1                     (3)  j  := j - 1
(4)  t4 := 4 * j           →         (4)  t4 := t4 - 4
(5)  t5 := a[t4]                     (5)  t5 := a[t4]
(6)  if t5 > v goto (3)              (6)  if t5 > v goto (3)
```

- loop unrolling

```
for (i=0; i<100; i++) →  for (i=0; i<100; i+=2)
{                        {
  a[i] = a[i] + b[i];      a[i] = a[i] + b[i];
}                          a[i+1] = a[i+1] + b[i+1];
                         }
```

Peephole Optimisation

- deletion of unnecessary instructions

```
(1)   MOV R0, a     →     (1)   MOV R0, a
(2)   MOV a, R0
```

if line (1) *and* (2) *are in the same BB*

- algebraic simplifications (using knowledge about available HW units and features)

```
(1)   MUL R0, 8                  (1)   SHL R0, 3   //(R0 << 3)
(2)   SQR R0   //(R0)²    →      (2)   MUL R0, R0
```

- control flow optimisations

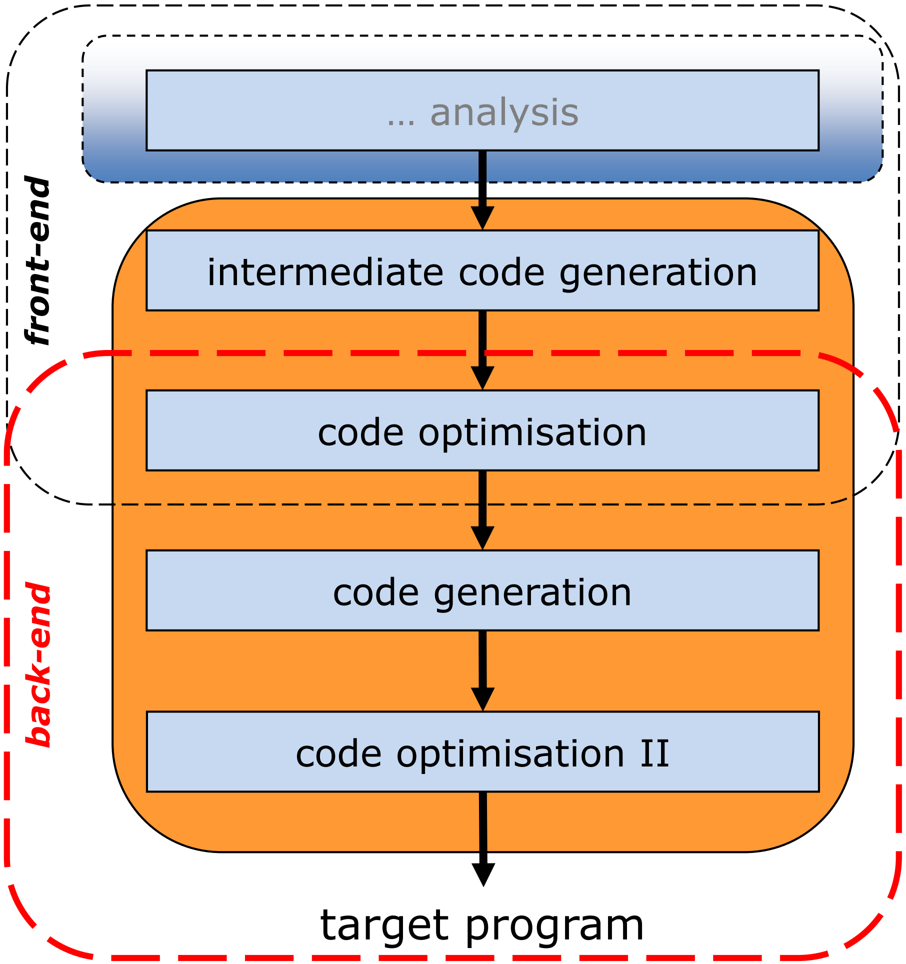
```
(1)   JMP 3       →     (1)   JMP N
(2)   ...               (2)   ...
(3)   JMP N             (3)   JMP N
```

Ideal vs. Real

ideal front-end/back-end

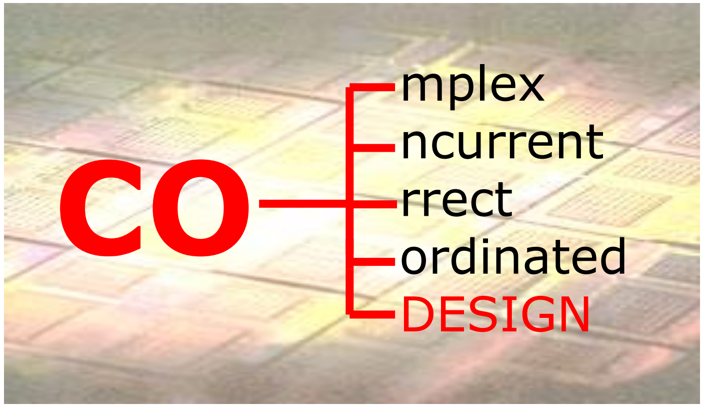
- use some target platform information for intermediate code optimisation
  - register usage
  - algebraic operations
  - …

real front-end/back-end



Contents

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



CO
mplex
ncurrent
rrect
ordinated
DESIGN

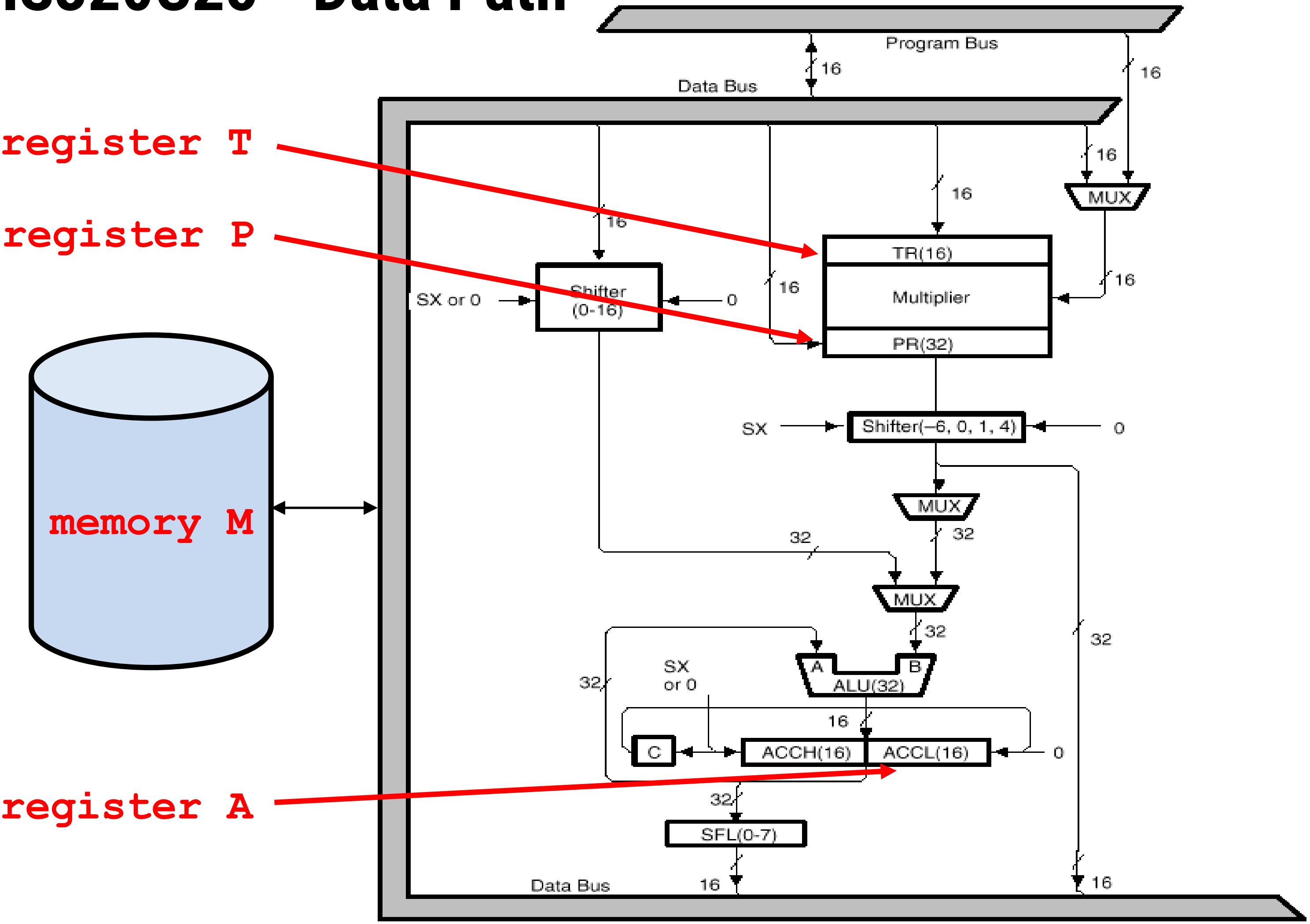
Compilers for Embedded Systems

- software development for embedded systems
  - transition from assembler to high-level languages (HLLs)
- *remember:* code generation requirements:
  - correct code
  - **efficient code** → **fast, small, low power**
  - efficient generation
- further requirements
  - possibility of formal verification
  - specification of ______________________________
  - support of DSP/multi-media algorithms and architectures
  - retargetable: quickly adaptable to new processors

Problems of Embedded Systems

- **______________________ register files, irregular data paths**
  - tight coupling of tasks register binding, instruction binding, instruction sequencing
- **allocation of memory addresses and address registers**
  - efficient use of address registers and specialised address generation units
- **code ________________________**
  - reduction of memory, important for cost sensitive applications

TMS320C25 - Data Path



register T

register P

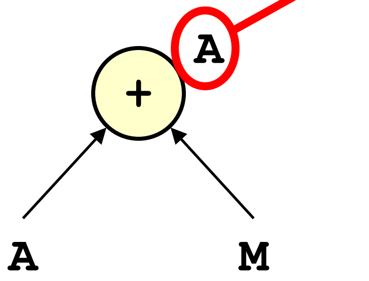
memory M

register A

TMS320C25 – Instruction Set (I)

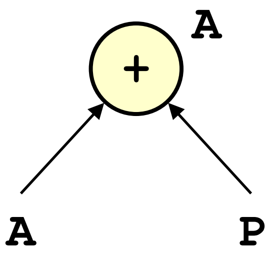
- addition

ADD



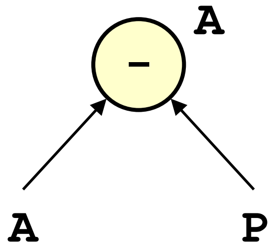
saved in register **A**

APAC



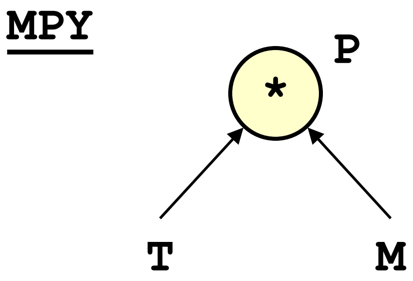
- subtraction

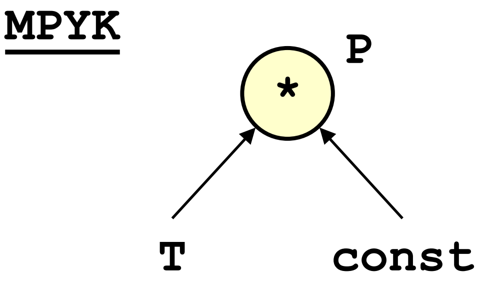
SPAC



TMS320C25 – Instruction Set (II)

- multiplication





- data transfer

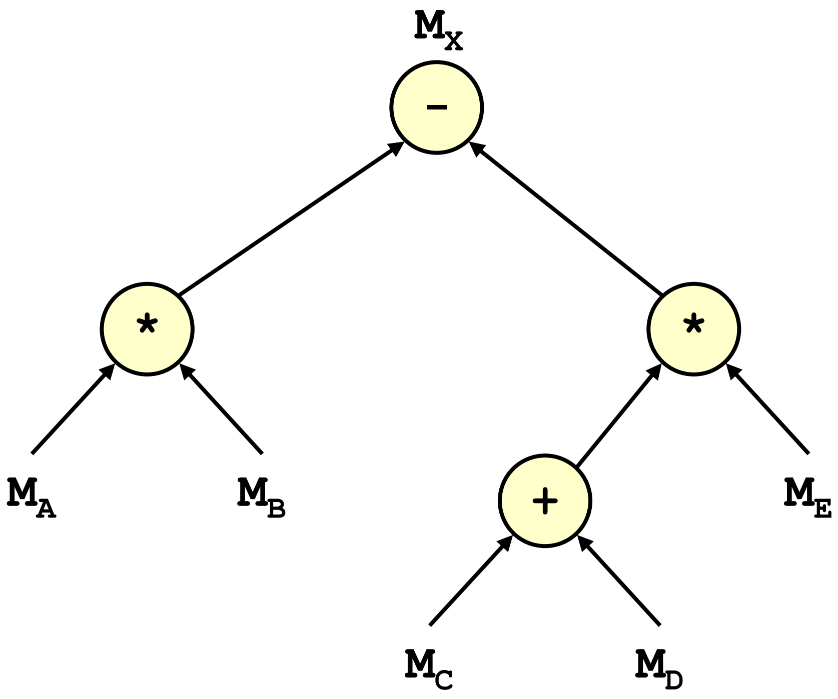
LACK	(const → A)
PAC	(P → A)
SACL	(A → M)
LAC	(M → A)
LT	(M → T)

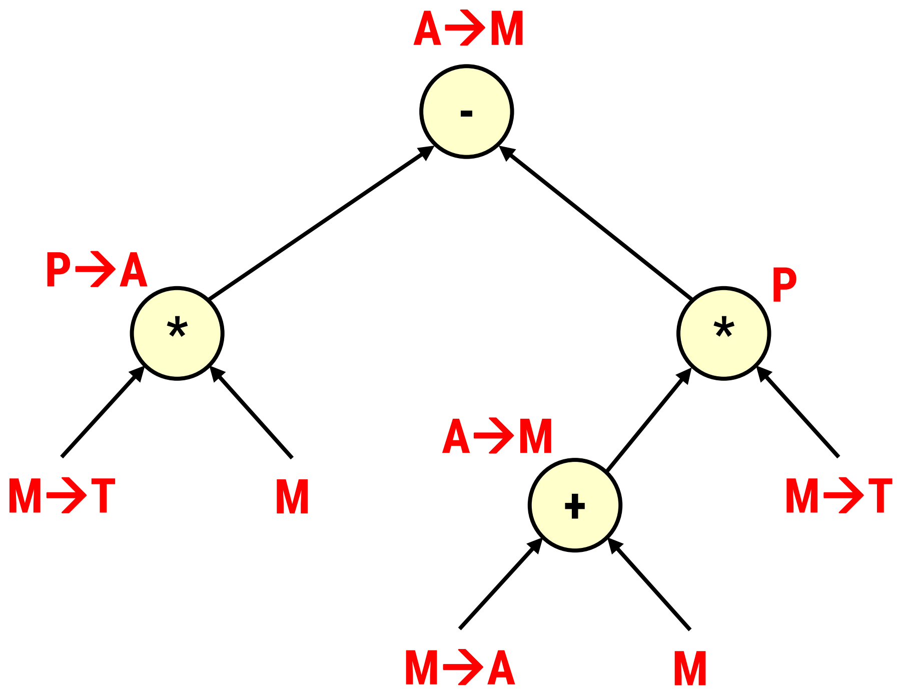
no other data transitions possible!

DAG – Example (I)

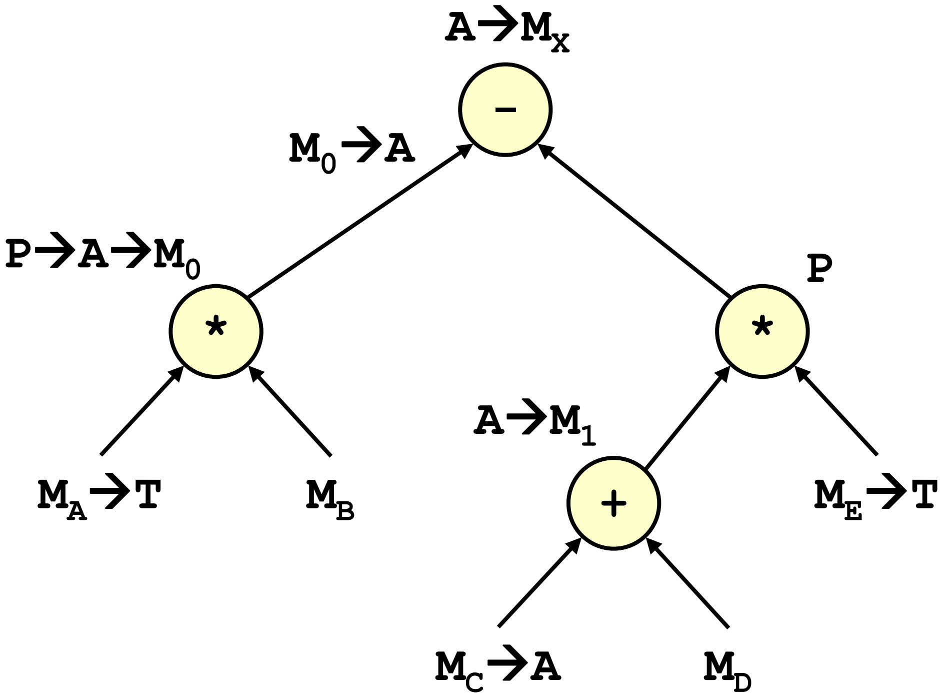
```
X = (A * B) – ((C + D) * E)
```

requirements of instruction set





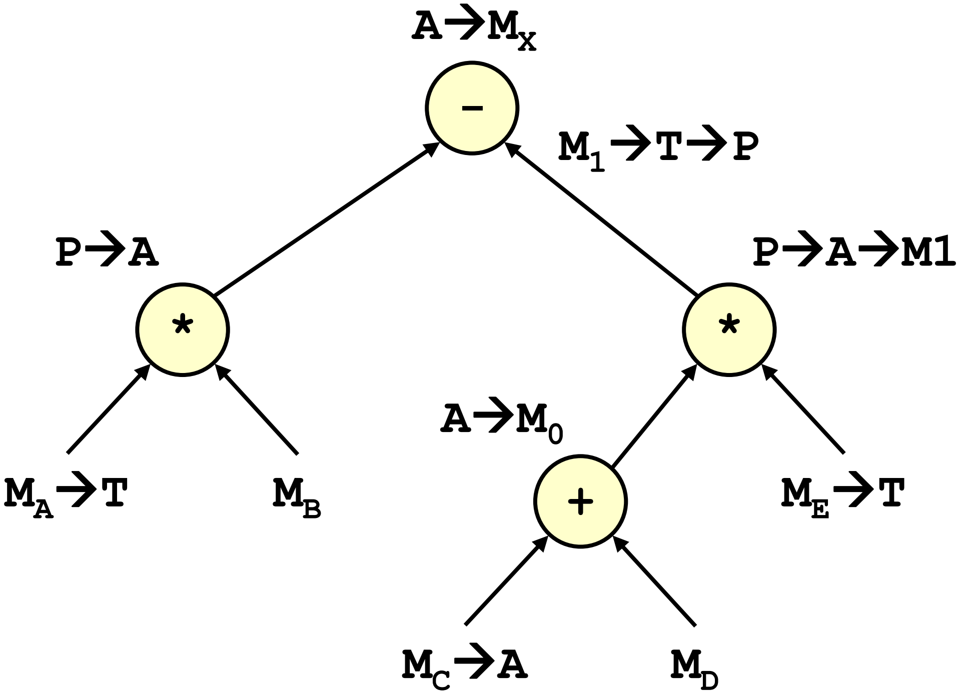
DAG – Example (II)



```
# LEFT sub tree first
# cost (# of instr.) = 12

LT      MA
MPY     MB
PAC
SACL    M0
LAC     MC
ADD     MD
SACL    M1
LT      ME
MPY     M1
LAC     M0
SPAC
SACL    MX
```

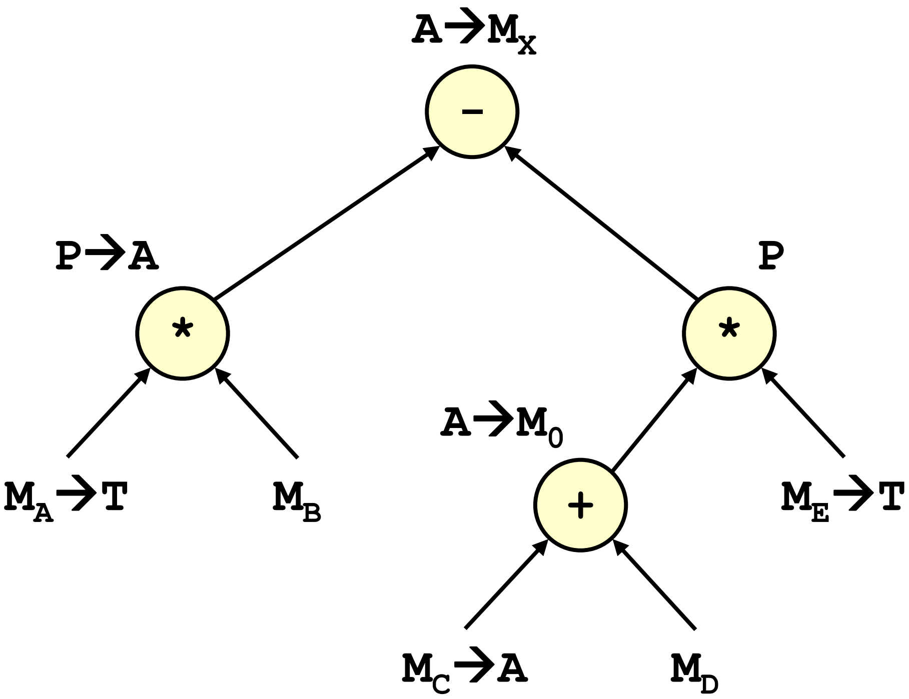
DAG – Example (III)



```
# RIGHT sub tree first
# cost (# of instr.) = 14

LAC     MC
ADD     MD
SACL    M0
LT      ME
MPY     M0
PAC
SACL    M1
LT      MA
MPY     MB
PAC
LT      M1        // M1→T
MPYK    1         // T →P
SPAC
SACL    MX
```

DAG – Example (IV)



```
# OPTIMAL code
# cost (# of instr.) = 10

LAC     MC
ADD     MD
SACL    M0
LT      MA
MPY     MB
PAC
LT      ME
MPY     M0
SPAC
SACL    MX
```

→ ***code generation based on ________________ ________________ is no longer optimal***

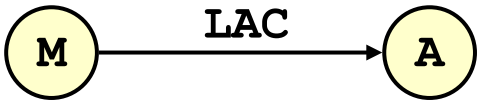
Register Transfer Graph

- **definition**

The register transfer graph for a processor architecture is a directed graph $G=(V,E)$ where each node represents a location in the data path at which data ____________________________.

An edge between two nodes r_i and r_j is labelled with the instructions that read an operand from r_i and write the value to r_j.

- example



RTG Criterion

- **definition**

The RTG criterion is satisfied if for all node triples (r_1,r_2,r_3) of the RTG for which

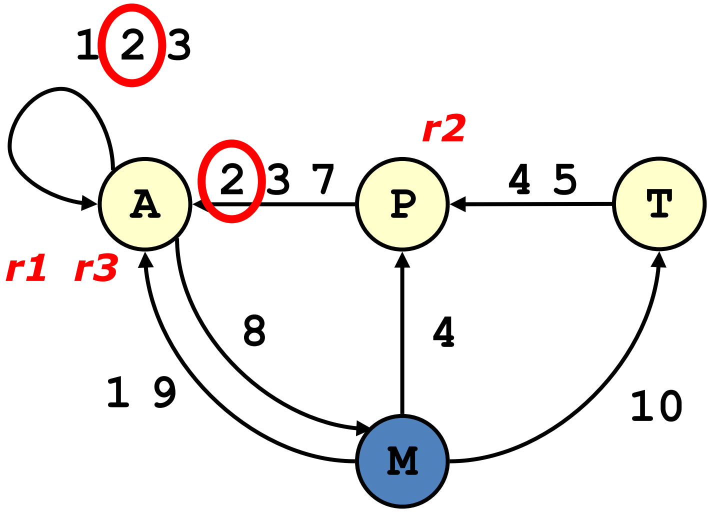
1. r_3 has incoming edges from *register* nodes r_1 and r_2 with identical labelling
2. there exists at least one cycle in the RTG including r_1 and r_2

the following holds:

each cycle between r_1 and r_2 includes a ____________________

RTG – TMS320C25

1:	ADD	*A = A + M*
2:	APAC	*A = A + P*
3:	SPAC	*A = A - P*
4:	MPY	*P = T * M*
5:	MPYK	*P = T * const*
6:	LACK	*A = const*
7:	PAC	*A = P*
8:	SACL	*M = A*
9:	LAC	*A = M*
10:	LT	*T = M*



paths:
A → M → P
A → A → M → P
A → M → T → P
A → A → M → T → P

→ *RTG criterion satisfied*

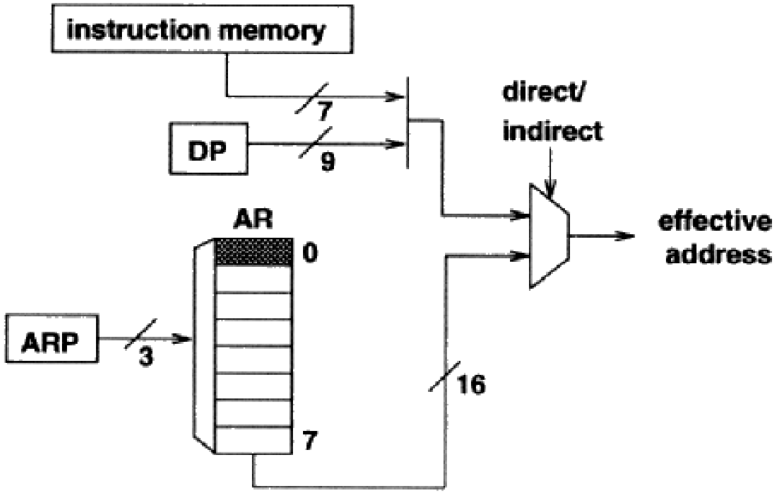
RTG – Code Generation

- **for processor architectures, that satisfy the RTG criterion,** ______________________________________
  ____[1]
  (n is number of DAG nodes)

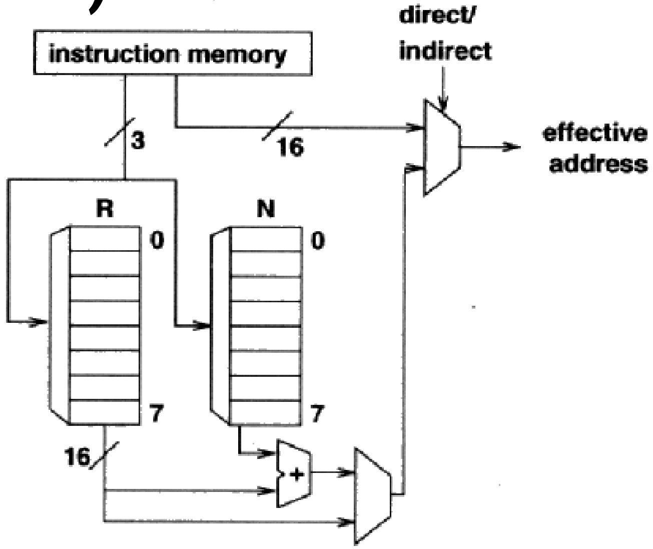
- code generation in two phases
  - optimal instruction selection and register binding (based on dynamic programming)
  - optimal scheduling

[1] *G. Araujo, S. Malik: Optimal Code Generation for Embedded Memory Non-Homgenous Register Architectures. In Proceedings of 1995 International Symposium on System Synthesis, IEEE Computer Society Press, Los Alamtios, 1995*

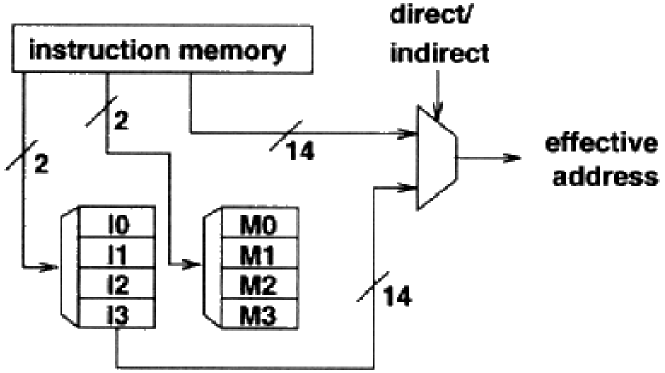
Address Generation Units (AGUs)



TMS320C2x



Motorola 56K



ADSP 210x

AGU Characteristics

- set of address registers **AR** for indirect addressing modes
- set of ______________________ **MR** for updating the **AR**
- modification operations in parallel to instruction execution
  - e.g. post-modifying: auto increment/decrement by
    - one address step (+/- 1)
    - the content of an **MR**

AGU Operations

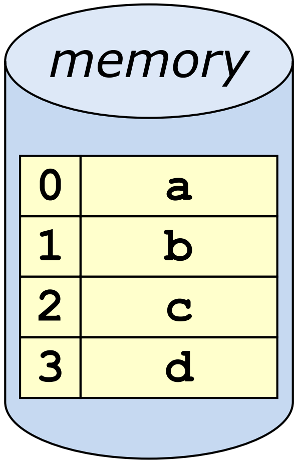
operation	function	cost
AR load	AR[ARP] = value	1
MR load	MR[MRP] = value	1
ARP load	ARP = value	0
MRP load	MRP = value	0
immediate modify	AR[ARP] += value	1
auto-increment	AR[ARP] ++	0
auto-decrement	AR[ARP] --	0
auto-modify	AR[ARP] += MR[MRP]	0

AGU Problems

- allocation of memory addresses and address/modification registers
  - scalar variables:
    - what is an optimal _______________________ of variables in memory (given *1* or *n* address registers)?
    - how to efficiently use **`MR`**?
  - array variables:
    - how to efficiently use **`AR/MR`** for loops?

Addressing Scalar Variables

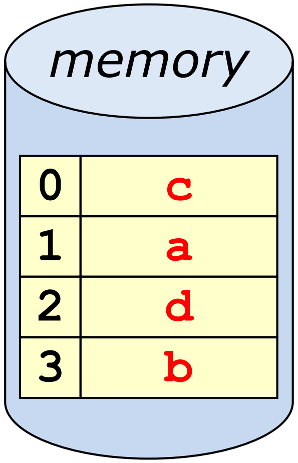
1 **AR** available
variable access sequence: **b,d,a,c,d,a,c,b,a,d,a,c,d**



0	a
1	b
2	c
3	d

```
LOAD AR, 1
AR += 2
AR -= 3
AR += 2
AR ++
AR -= 3
AR += 2
AR --
AR --
AR += 3
AR -= 3
AR += 2
AR ++
```

→ cost = 9



0	c
1	a
2	d
3	b

```
LOAD AR, 3
AR --
AR --
AR --
AR += 2
AR --
AR --
AR += 3
AR -= 2
AR ++
AR --
AR --
AR += 2
```

→ cost = 5

Access Graph

- **definition**

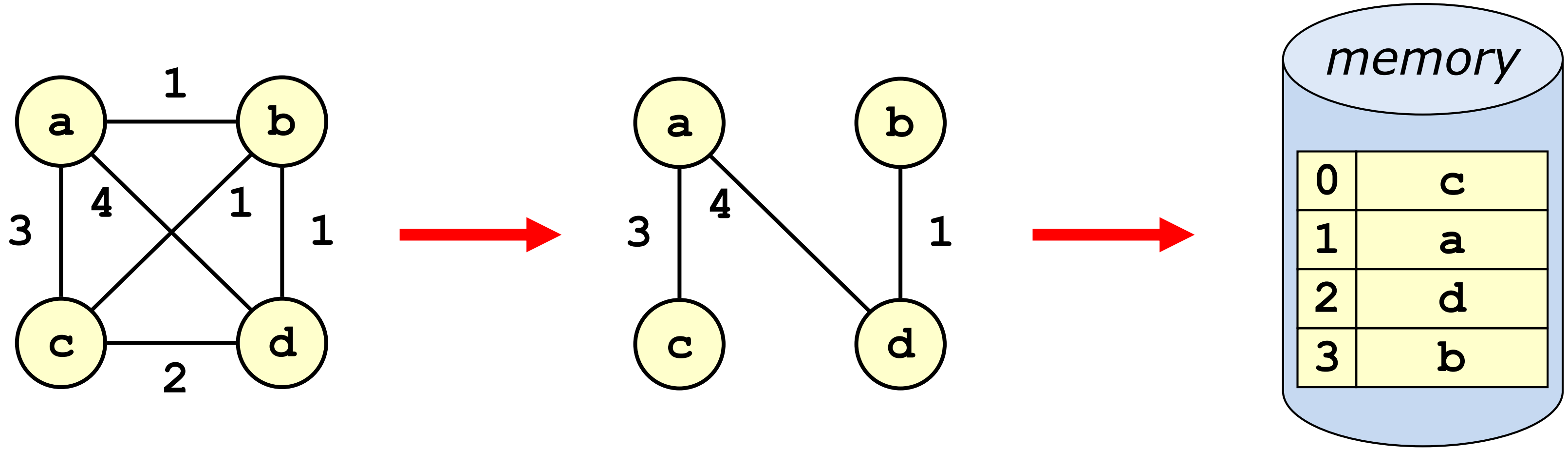
  The access graph for a given set of variables and an access sequence consists of

  - nodes that model the variables
  - edges that connect nodes if the corresponding variables are adjacent in the access sequence
  - edge weights that denote the number of transitions between the corresponding variables in the access sequence

- an optimal address allocation is given by a ______________ ______________ in the access graph (path that visits each node exactly once) with maximal edge weight *(NP-hard problem)*

Access Graph – Example (I)

1 **`AR`** available
variable access sequence: `b,d,a,c,d,a,c,b,a,d,a,c,d`



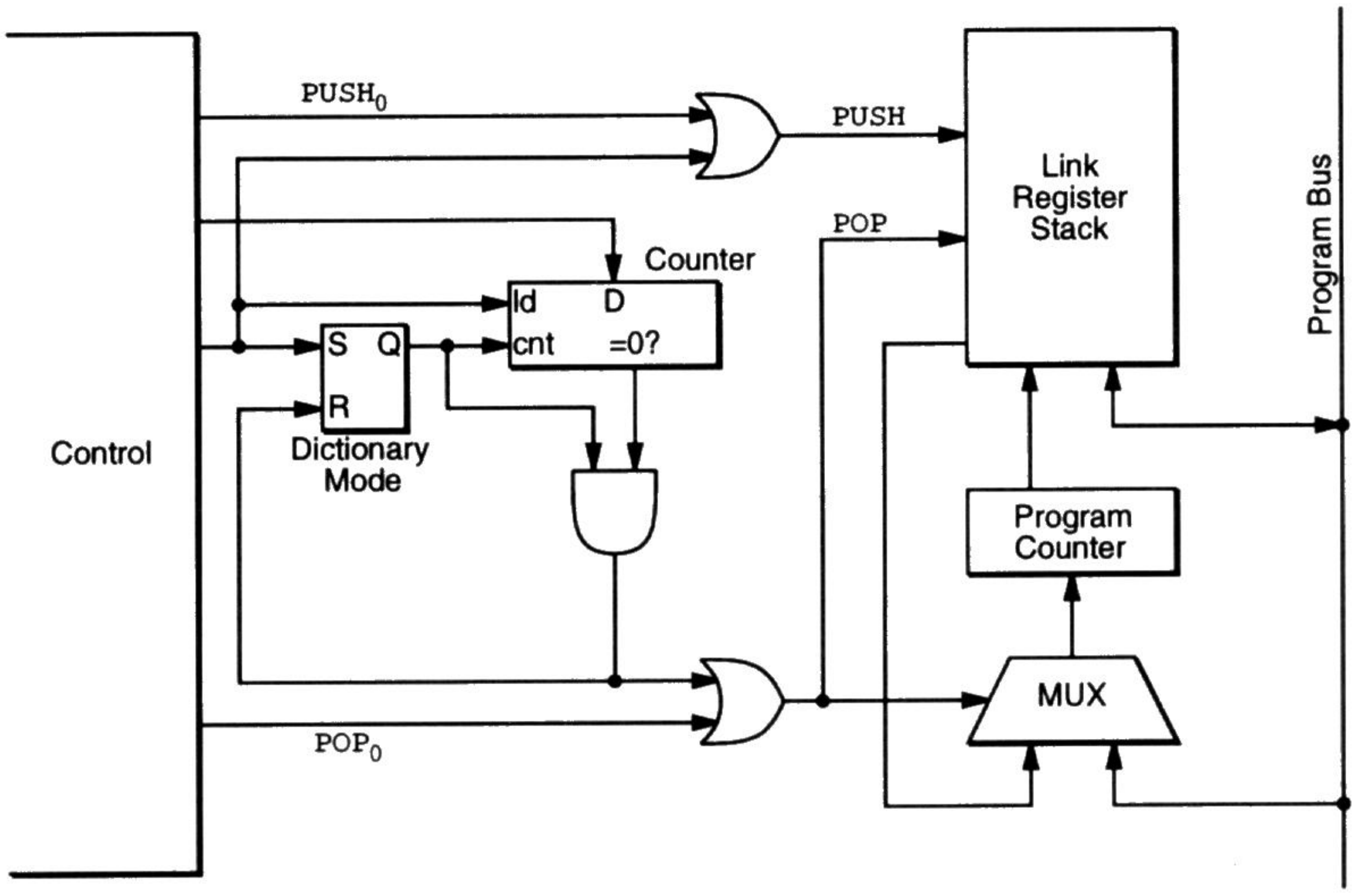
0	c
1	a
2	d
3	b

Code Compression

- target code is redundant and can be compressed
  - GP systems: decompression at program loading time
  - for embedded systems the reduction of program ROMs or RAMs is important → **decompression in the cache**

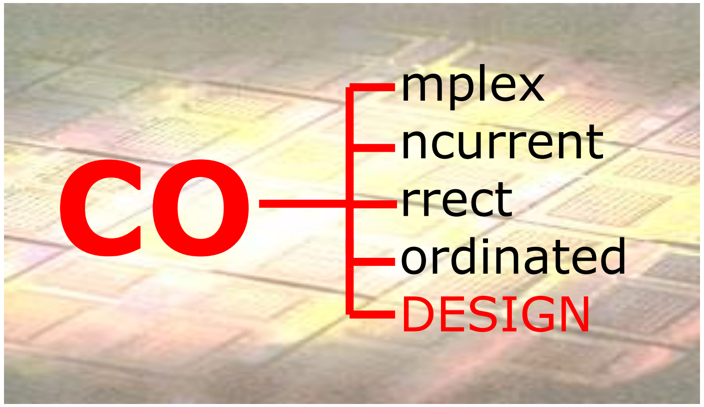
- external pointer macro (EPM) model
  - ___________________: contains frequently used code sequences (mini-subroutines)
  - ___________________: contains instructions and pointers to the dictionary
  - implementation in SW or HW

EPM Model in Hardware



Contents

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



Classification

- portable compiler
  - ____________________ retargetable
  - code generation by tree pattern matching

- compiler-compiler (CC)
  - ____________ retargetable (semi-automatic)
  - compiler is generated from a description of the target architecture (processor model)

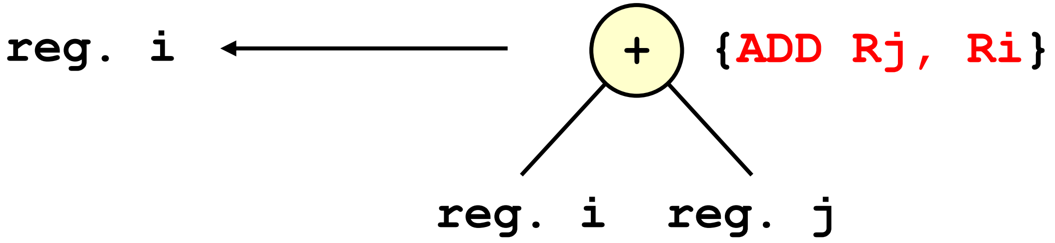
- machine independent compiler
  - ____________________ retargetable
  - compiler generates code for several processors/ processor variants
  - for parametrisable architectures

Tree Pattern Matching

- rules for transforming a syntax tree or DAG are given as tree patterns

`replacement` ⟵ `pattern {action}`

- example



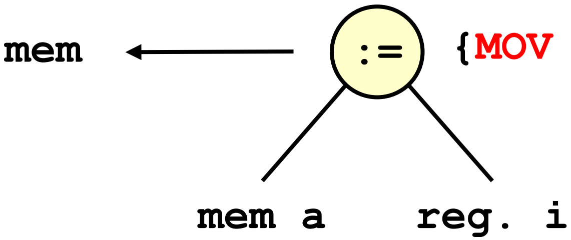
→ ***stepwise replacement by tree pattern matching until the tree contains only one node***

Target Instruction (I)

(1) reg i ← const c {MOV #c, Ri}

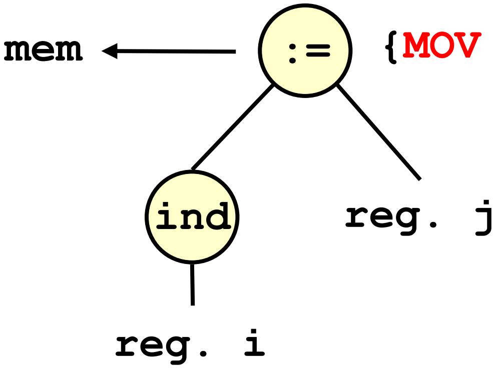
(2) reg i ← mem a {MOV a, Ri}

(3)



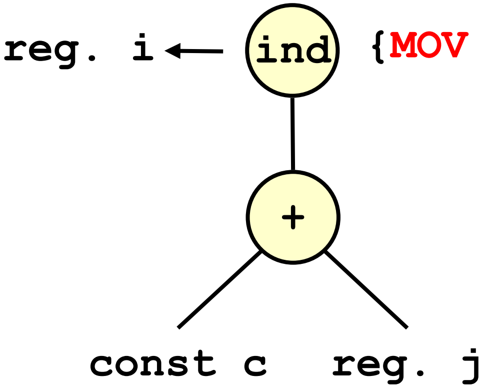
{MOV Ri, a}

(4)



{MOV Rj, *Ri}

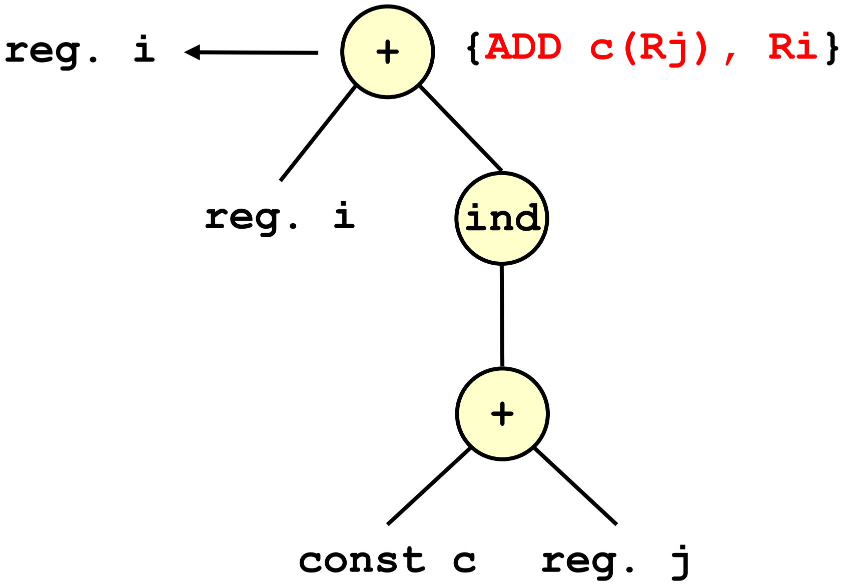
(5)



{MOV c(Rj),Ri}

Target Instructions (II)

(6)



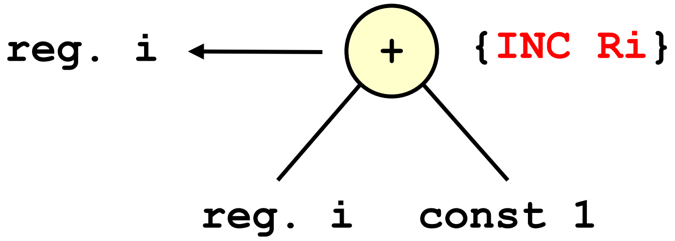
reg. i ← + {ADD c(Rj), Ri}

reg. i ind

+

const c reg. j

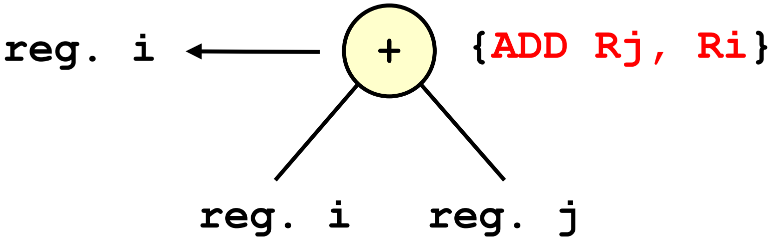
(8)



reg. i ← + {INC Ri}

reg. i const 1

(7)



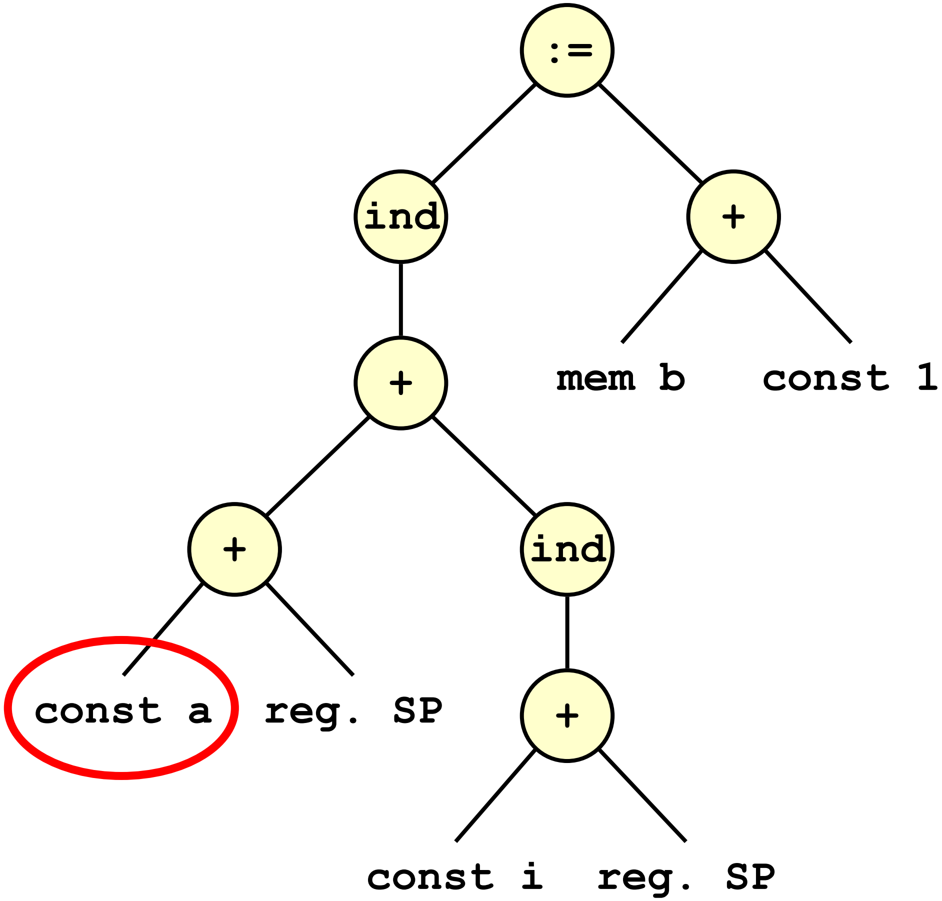
reg. i ← + {ADD Rj, Ri}

reg. i reg. j

Tree Pattern Matching – Example (I)

```
a[i] := b + 1
```

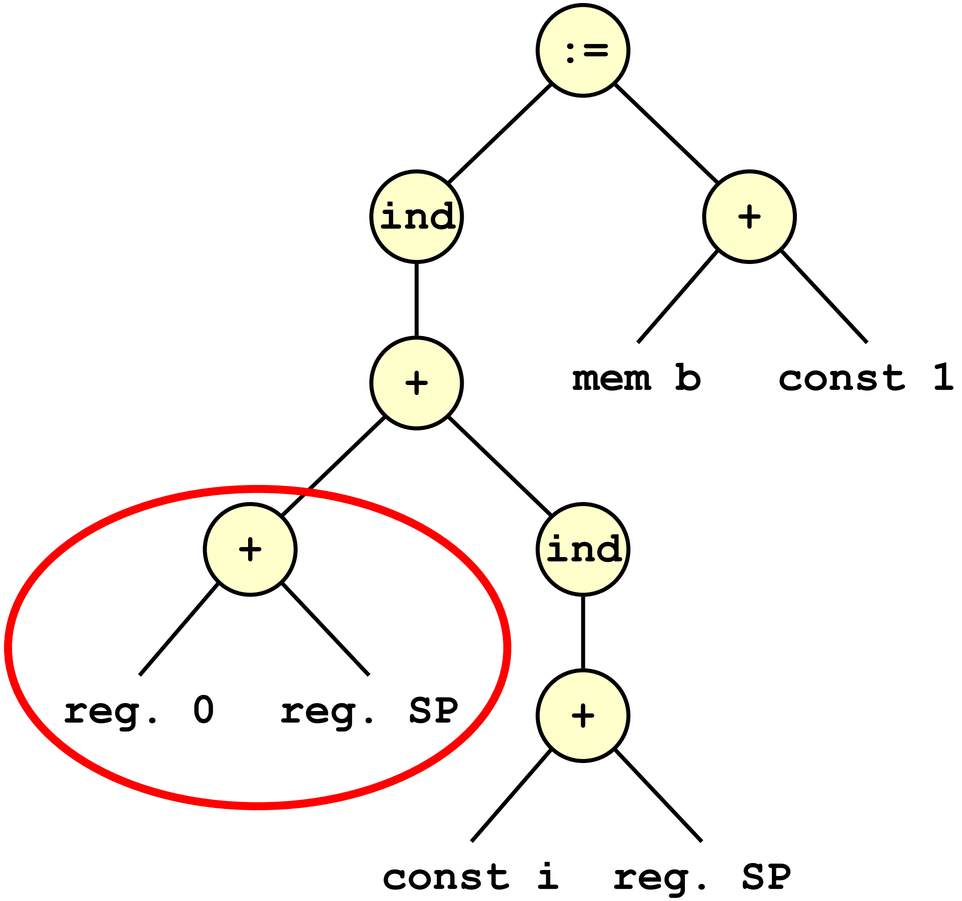
```
(1) MOV #a, R0          (1)
```



Tree Pattern Matching – Example (II)

```
a[i] := b + 1
```

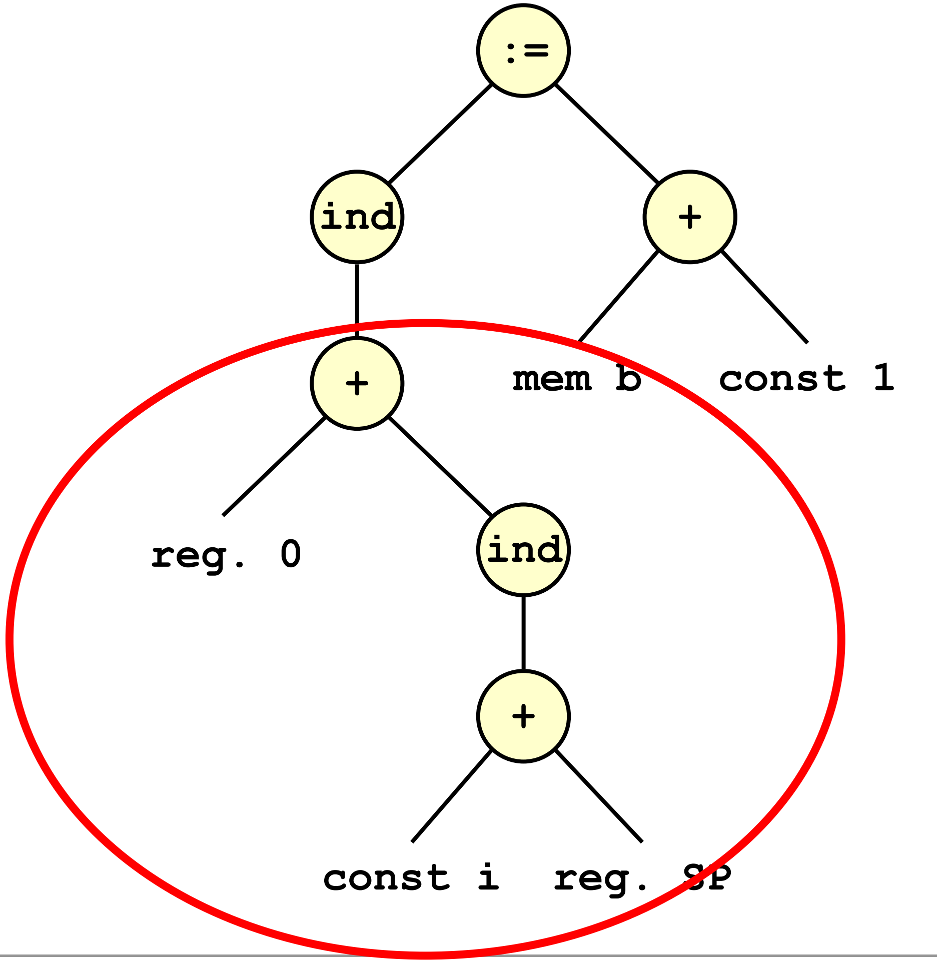
```
(1) MOV #a, R0        (1)
(2) ADD SP, R0        (7)
```



Tree Pattern Matching – Example (III)

```
a[i] := b + 1
```

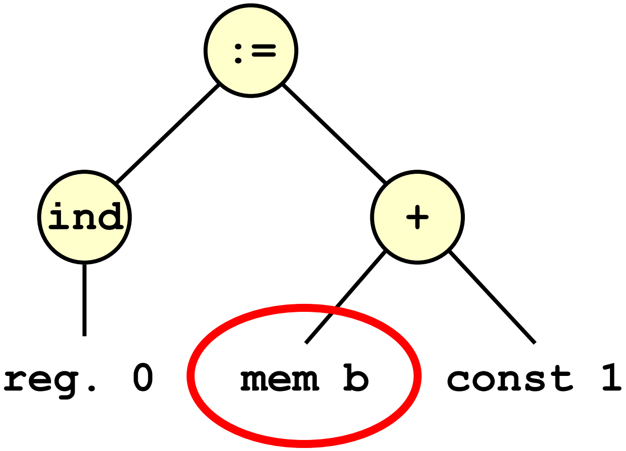
```
(1) MOV #a, R0          (1)
(2) ADD SP, R0          (7)
(3) ADD i(SP), R0       (6)
```



Tree Pattern Matching – Example (IV)

`a[i] := b + 1`

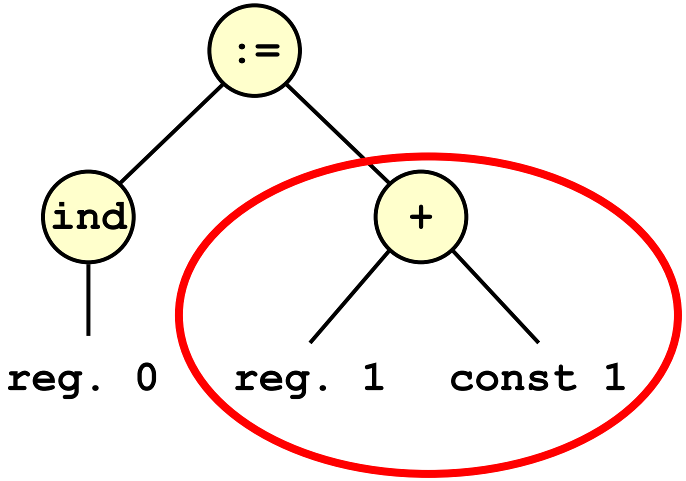
```
(1) MOV #a, R0          (1)
(2) ADD SP, R0          (7)
(3) ADD i(SP), R0       (6)
(4) MOV b, R1           (2)
```



Tree Pattern Matching – Example (V)

`a[i] := b + 1`

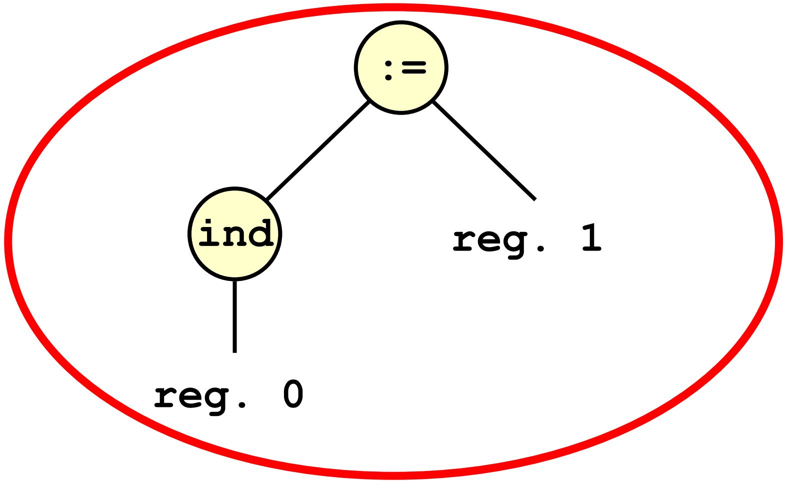
```
(1) MOV #a, R0          (1)
(2) ADD SP, R0          (7)
(3) ADD i(SP), R0       (6)
(4) MOV b, R1           (2)
(5) INC R1              (8)
```



Tree Pattern Matching – Example (VI)

```
a[i] := b + 1
```

```
(1) MOV #a, R0          (1)
(2) ADD SP, R0          (7)
(3) ADD i(SP), R0       (6)
(4) MOV b, R1           (2)
(5) INC R1              (8)
(6) MOV R1, *R0         (4)
```



CC - Processor Models

- description of target architecture necessary for the compiler- compiler
  - ____________________ models
    - describe the instruction set
    - simulation relatively fast (100 – 1000 times slower than target machine)
    - not very accurate (no pipelining effects)
  - ____________________ models
    - describe the processor on the register transfer level
    - accurate
    - simulation rather slow
  - mixed models

CC - Case Studies

- ***FlexWare***
  - developed at *SGS-Thomson Microelectronics*
  - instruction set simulator (*Insulin*) and code generator (*CodeSyn*)
  - mixed processor model
  - for DSPs, ASIPs

- ***CHESS***
  - developed at *IMEC Leuven*
  - instruction set simulator and code generator
  - target architecture described in *nML*
  - for DSP architectures