Compiler Construction

Chapter 6 – Register Machines and Instruction Selection

Prof. Dr. Jörg Kreiker
joerg.kreiker@informatik.hs-fulda.de

Fachbereich Angewandte Informatik
Hochschule Fulda – University of Applied Sciences

June 22, 2017

Agenda

1. Register machines
2. MC68000
3. Adressing
4. Instruction Selection – General Ideas
5. Peephole Optimization

CMA and JVM are slow

- Each operation works with the stack (e.g. 3 accesses per addition)
- Memory access is slow (even using caches)
- Each stack access computes $\mathrm{FP}+k$
- Even old processors have registers like an accumulator register to prevent storing intermediate results and writing constants onto the stack

Modern Register Machines

Hochschule Fulda
University of Applied Sciences

Development of register machines:

- Number of registers limited by number of transistors

$\rightarrow$ only few registers with special purposes: AX: accumulator, BX: second argument, CX: counter, DX: data pointer

- limited contexts, e.g. mul always stores result to AX
- Today: Number of registers limited by instruction length
- universal contexts
- Map to complex instructions vs. make best use of available registers

Notation: Use $M[\cdot]$ for memory access, R_i for registers, $+$ for addition (infix), $:=$ for assignment, etc.

Hochschule Fulda University of Applied Sciences

Example: Motorola MC68000 Processor of the 680x0-series features with 8 data and 8 address registers and many ways of addressing

Notation	Description	Meaning
D_n	data register direct	D_n
A_n	address register direct	A_n
(A_n)	adress register indirect	$M[A_n]$
$d(A_n)$	adress register indirekt with displacement	$M[A_n + d]$
$d(A_n, D_m)$	adress register indirect with index and displacement	$M[A_n + D_m + d]$
x	absolute short	$M[x]$
x	absolute long	$M[x]$
$\#x$	immediate	x

Addressing in the MC68000

- 2-address-machine, maximal 2 arguments per instruction

  $$\text{add}\ \ D_1\ \ D_2$$

  adds the contents of D_1 and D_2 and stores the result to D_2
- Most instructions work on bytes, words (2 bytes) or double words (4 Bytes): Use suffixes .B, .W, .D (default: word)
- Execution time of instruction: cost of operation plus cost of addressing operands

add instruction takes 4 cycles. Arguments incur the following overhead

	Addressing	.B, .W	.D
D_n	data register direct	0	0
A_n	address register direct	0	0
(A_n)	adress register indirect	4	8
$d(A_n)$	adress register indirect with displacement	8	12
$d(A_n, D_m)$	adress register indirect with index and displacement	10	14
x	absolute short	8	12
x	absolute long	12	16
$\#x$	immediate	4	8

$\Rightarrow$ Smart argument selection gets rid of many moves and intermediate results.

Consider cost as sum $o + a_1 + a_2$

Instruction	move.B $8(A_1, D_1.\mathrm{W}), D_5$	
needs	$4 + 10 + 0 = 14$	cycles

Alternative:

adda	$\#8, A_1$	Kosten:	$8 + 8 + 0 = 16$
adda	$D_1.\mathrm{W}, A_1$	Kosten:	$8 + 0 + 0 = 8$
move.B	$(A_1), D_5$	Kosten:	$4 + 4 + 0 = 8$

with total cost 32 or:

adda	$D_1.\mathrm{W}, A_1$	Kosten:	$8 + 0 + 0 = 8$
move.B	$8(A_1), D_5$	Kosten:	$4 + 8 + 0 = 12$

with total cost 20

- The various code sequences are equivalent wrt memory and result
- Differ wrt value of register A_1
- Practical algorithm needs to reflect such constraints

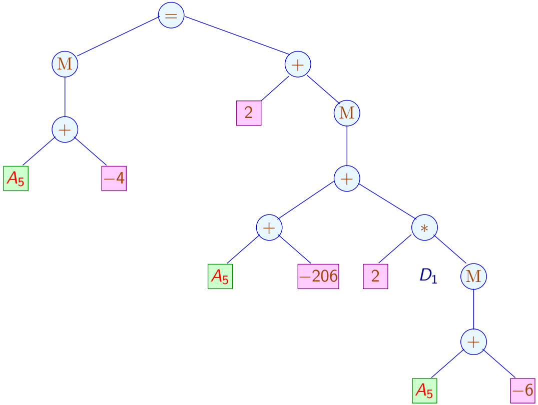
$$\text{int } b,\ i,\ a[100];$$
$$b = 2 + a[i];$$

Assume variables are addressed relative to frame pointer A_5 with adresses $-4, -6, -206$. Assignment translates to

$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

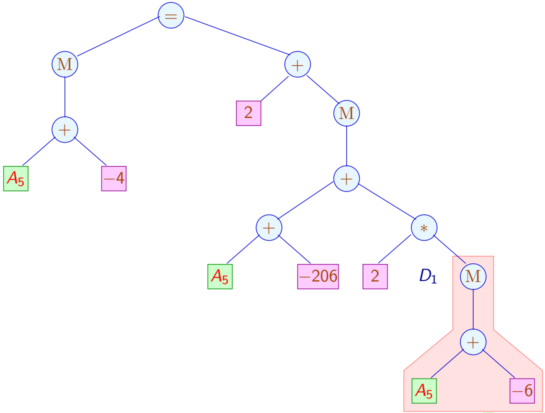
$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

Expression corresponds to syntax tree:



$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

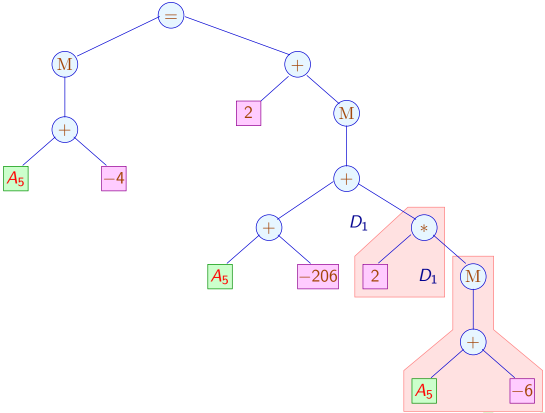
Possible instruction selections move $-6(A_5)$, D_1



Hochschule Fulda University of Applied Sciences

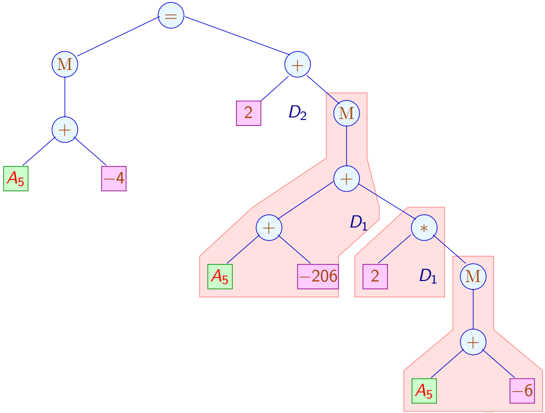
$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

Possible instruction selections add D_1, D_1



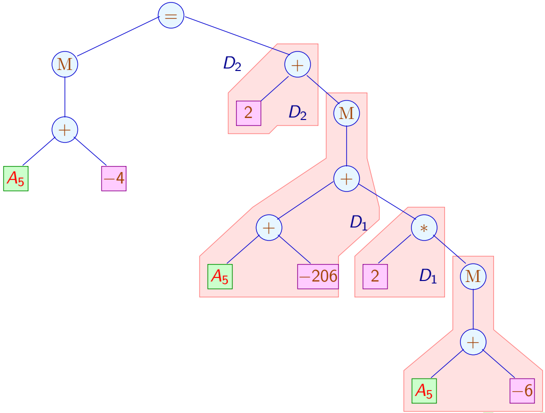
$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

Possible instruction selections move $-206(A_5, D_1),\ D_2$



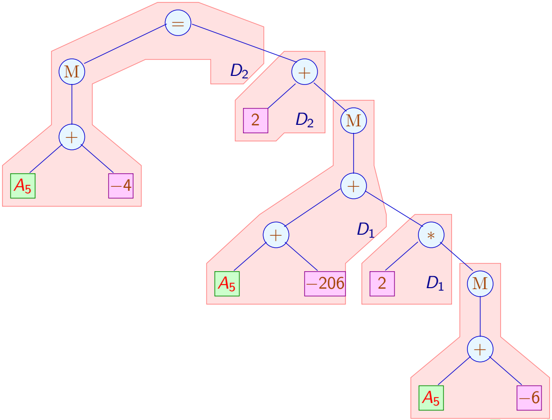
$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

Possible instruction selections addq #2, D_2



$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

Possible instruction selections move D_2, $-4(A_5)$



Possible Code Sequences

move	$-6(A_5),\ D_1$	Cost:	12
add	$D_1,\ D_1$	Cost:	4
move	$-206(A_5, D_1),\ D_2$	Cost:	14
addq	$\#2,\ D_2$	Cost:	4
move	$D_2,\ -4(A_5)$	Cost:	12
Total cost :			46

Alternative

Hochschule Fulda
University of Applied Sciences

move.L	$A_5,\ A_1$	Cost:	4
adda.L	$\#-6,\ A_1$	Cost:	12
move	$(A_1),\ D_1$	Cost:	8
mulu	$\#2,\ D_1$	Cost:	44
move.L	$A_5,\ A_2$	Cost:	4
adda.L	$\#-206,\ A_2$	Cost:	12
adda.L	$D_1,\ A_2$	Cost:	8
move	$(A_2),\ D_2$	Cost:	8
addq	$\#2,\ D_2$	Cost:	4
move.L	$A_5,\ A_3$	Cost:	4
adda.L	$\#-4,\ A_3$	Cost:	12
move	$D_2,\ (A_3)$	Cost:	8

Total cost : 124

In general, complex instructions have an advantage

- complex addressing produces shorter and faster code
- simpler sequences often need more auxiliary registers
- more complex sequence overwrites fewer auxiliary registers
- $\rightsquigarrow$ find the fastest sequence of instructions for a given syntax tree

Idea: compute all possible sequences and their cost and select the cheapest.

- Grammar describes all legal code sequences:
  1. available register classes: non-terminals
  2. operators and constants: terminals
  3. instructions: rules
  4. choose the cheapest derivation of a tree

$\rightarrow$ Tree Parsing

Grammar Example

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

- two register classes D (Data) and A (Address)
- arithmetic only for data
- addresses are only loaded into address registers
- moves between data and address registers

Derivation Example

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A+A]$	$D \rightarrow D+D$	$A \rightarrow D$

Instruction Example: $M[A+c]$

Alternatively, derivation by $D \rightarrow M[A+A]$ possible.

Derivation Example

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Instruction Example: $M[A + c]$

Alternatively, derivation by $D \rightarrow M[A + A]$ possible.

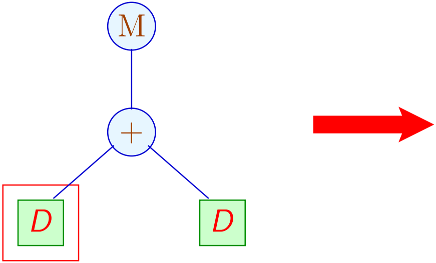
Loads :	Computations :	Moves :
$D \to M[A]$	$D \to c$	$D \to A$
$D \to M[A + A]$	$D \to D + D$	$A \to D$

Instruction Example: $M[A + c]$

Alternatively, derivation by $D \to M[A + A]$ possible.

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Instruction Example: $M[A + c]$



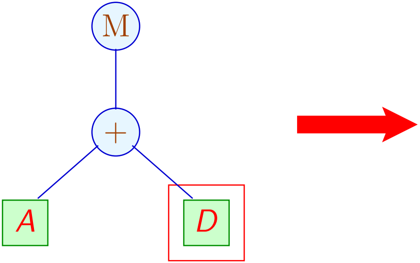
Alternatively, derivation by $D \rightarrow M[A + A]$ possible.

Derivation Example

Hochschule Fulda
University of Applied Sciences

Loads :	Computations :	Moves :
$D \to M[A]$	$D \to c$	$D \to A$
$D \to M[A + A]$	$D \to D + D$	$A \to D$

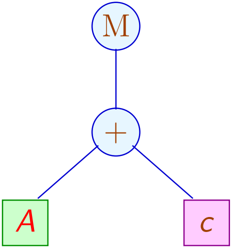
Instruction Example: $M[A + c]$



Alternatively, derivation by $D \to M[A + A]$ possible.

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Instruction Example: $M[A + c]$

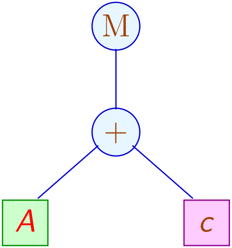


Alternatively, derivation by $D \rightarrow M[A + A]$ possible.

Derivation Example

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Instruction Example: $M[A + c]$



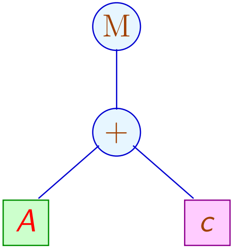
Alternatively, derivation by $D \rightarrow M[A + A]$ possible.

Idea: Compute all possible derivations of a tree by *tree parsing:*

- compute possible derivations *bottom-up* from the leaves
- check which RHS matches
- apply rule backwards
- add LHS to predecessor in tree
- at top, choose cheapest derivation
- descend and emit instructions for each rule applied

Loads :	Computations :	Moves :
$D \to M[A]$	$D \to c$	$D \to A$
$D \to M[A+A]$	$D \to D+D$	$A \to D$

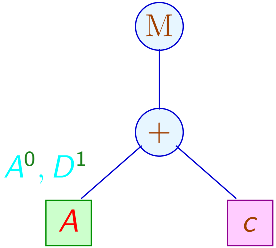
Consider $M[A+c]$



Example

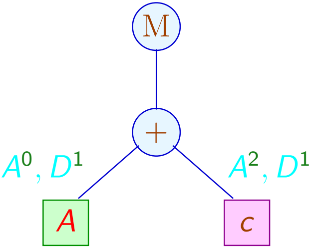
Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Consider $M[A + c]$



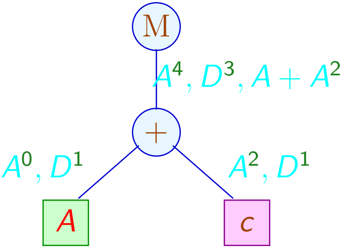
Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Consider $M[A + c]$



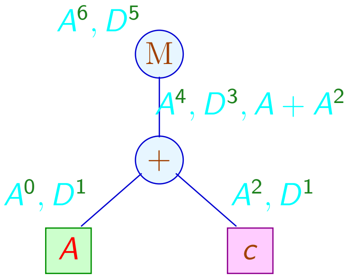
Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A+A]$	$D \rightarrow D+D$	$A \rightarrow D$

Consider $M[A+c]$



Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

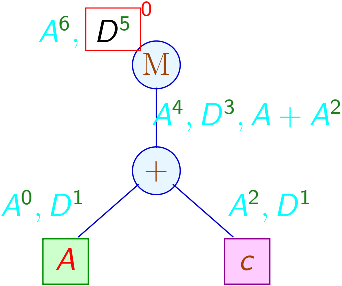
Consider $M[A + c]$



Hochschule Fulda
University of Applied Sciences

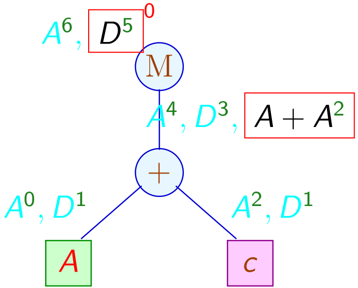
Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Consider $M[A + c]$



Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A+A]$	$D \rightarrow D+D$	$A \rightarrow D$

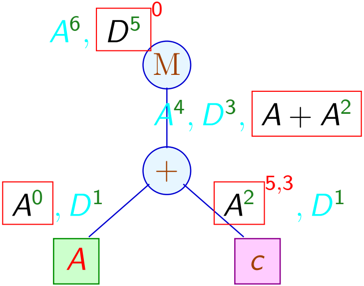
Consider $M[A+c]$



Example

Loads :	Computations :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

Consider $M[A + c]$



- Optimize according to runtime, code size, parallelism, energy consumption
- Translate rule application into finite, deterministic tree automaton
- Works only if each instruction has exactly one result

Omissions

- In modern architectures, complex instructions may lead to pipeline stalls
- Modern processors have universal registers
  - instruction applicability does not depend on its arguments
  - code improvements may happen on the emitted code

$\leadsto$ generate only simple register code and optimize later

The generated code contains many redundancies, such as:

`move` R_7 R_7

`pop` 0

`move` R_4 R_7
`sub` R_4 R_4 R_7

Peephole optimization matches certain patterns and replaces them by simpler patterns