

Generatie en optimalisatie van de target code

Stel dat er intermediaire code gegenereerd is, bv. P-machine code of bytecode. De target taal bestaat uit instructies voor een specifieke concrete machine, die verschilt van de P-machine: de target code moet gewoonlijk rekening houden met verschillende adresseringswijzen, het gebruik van registers, enz.

Het toepassen van vaste schema's voor de generatie van target code zal weer een versie opleveren die verder geoptimaliseerd moet worden.

3 issues:

- **Instructie selectie**: welke instructies kiezen? Gespecialiseerde of algemene?
- **Register allocatie**: welke registers gebruiken we en waarvoor? (berekeningen gebeuren nu niet meer op de stack) Registers zijn schaars!
- **Code scheduling**: herschik de instructies zo dat we efficiënte code bekomen

We bekijken eerst de evaluatie van expressies.

In de intermediaire code gebeurt de evaluatie van een expressie op de stack.

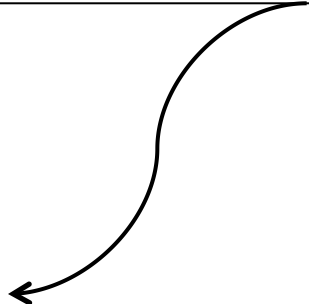
In de target machine gebeurt het rekenen in **registers**, met behulp van instructies zoals **add \$t2, \$t0, \$t1**, wat de som berekent van de inhoud van registers t0 en t1, en het resultaat plaatst in register t2.

Er zijn een beperkt aantal registers beschikbaar (bv. 10, geen 100).

In instructies zijn vaak relatieve adressen toegelaten, bv. **lw \$t1, 20 (\$fp)** laadt de inhoud van adres (20 + inhoud van register fp) in register t1.

Totnogtoe zag de vertaling van een assignment als $a = b + c - d$ eruit als volgt:

Veronderstel dat a, b, c, d
overeenkomen met locaties 1,2,3,4



```
iload 2      ; Push int b onto stack
iload 3      ; Push int c onto stack
iadd         ; Add top two stack values
iload 4      ; Push int d onto stack
isub         ; Subtract top two stack values
istore 1     ; Store top stack value into a
```

Figure 13.1: Bytecodes for $a = b + c - d$;

In plaats van de operanden op de stack te zetten worden er tijdelijke registers voor gealloceerd. We houden bij welke dat zijn en genereren instructies die de juiste tijdelijke registers gebruiken.

Als a,b,c,d geplaatst zijn op locaties 12, 16, 20, 24, relatief t.o.v. fp:

lw	\$t0,16(\$fp)	# Load b, at 16+\$fp, into \$t0
lw	\$t1,20(\$fp)	# Load c, at 20+\$fp, into \$t1
add	\$t2,\$t0,\$t1	# Add \$t0 and \$t1 into \$t2
lw	\$t3,24(\$fp)	# Load d, at 24+\$fp, into \$t3
sub	\$t4,\$t2,\$t3	# Subtract \$t3 from \$t2 into \$t4
sw	\$t4,12(\$fp)	# Store result into a, at 12+\$fp

Figure 13.2: MIPS code for $a = b + c - d$;

Vertaling van expressie-bomen

De voorgestelde methode voldoet niet meer wanneer meer (tijdelijke) registers nodig zijn dan er beschikbaar zijn. Dan is **register spilling** (= transport van data tussen registers en geheugen) onvermijdelijk. Dit transport is duur (=traag), dus we willen het zoveel mogelijk vermijden.

Voorbeeld: $(a - b) + ((c + d) + (e * f))$

Voorbeeld: $(a - b) + ((c + d) + (e * f))$

Een naïeve vertaling:

- bereken $a - b$ (nu 2 regs in gebruik)
- hou het resultaat in een register, maak de registers voor a en b vrij (nu 1 reg in gebruik)
- bereken $c + d$ (nu 3 regs in gebruik)
- hou het resultaat in een register, maak de registers voor c en d vrij (nu 2 regs in gebruik)
- bereken $e * f$ (nu 4 regs in gebruik)
- hou het resultaat in een register, maak de registers voor e en f vrij (nu 3 regs in gebruik)
- bereken $((c + d) + (e * f))$ (nu 3 regs in gebruik)
- hou het resultaat in een register, maak de registers voor $(c + d)$ en $(e * f)$ vrij (nu 2 regs in gebruik)
- bereken het resultaat (nu 2 regs in gebruik)
- hou het resultaat in een register, maak de registers voor $(a - b)$ en $((c + d) + (e * f))$ vrij (nu 1 reg in gebruik)

$$(a - b) + ((c + d) + (e * f))$$

Het is gemakkelijk in te zien dat er maar 3 registers nodig zijn indien men eerst $((c + d) + (e * f))$ evalueert en pas daarna $(a - b)$

Algemene methode: bereken eerst, bottom – up, de **register need** van subexpressies (of subtrees): het minimale aantal registers dat ervoor nodig is. Maak gebruik van het feit dat de registers, op één na, terug vrijgemaakt worden.

Observatie: voor een expressie $(E1 \text{ op } E2)$ zijn er 2 mogelijkheden:

- E1 en E2 hebben even veel registers nodig; dan is er voor $(E1 \text{ op } E2)$ nog 1 meer nodig.
- E1 en E2 hebben een verschillend aantal registers nodig; dan zijn er voor $(E1 \text{ op } E2)$ het maximum van de 2 aantallen nodig.

We beginnen dus telkens met de evaluatie van de subexpressie met de **grootste** register need.

Algoritme voor Register Needs:

```
procedure RegisterNeeds (T)  
  if T.kind = Identifier or T.kind = IntegerLiteral  
  then T.regCount  $\leftarrow$  1  
  else  
    call RegisterNeeds (T.leftChild)  
    call RegisterNeeds (T.rightChild)  
    if T.leftChild.regCount = T.rightChild.regCount  
    then T.regCount  $\leftarrow$  T.rightChild.regCount + 1  
    else  
      T.regCount  $\leftarrow$  Max (T.leftChild.regCount, T.rightChild.regCount)  
    end  
  end
```

Figure 13.9: An Algorithm to Label Expression Trees with Register Needs

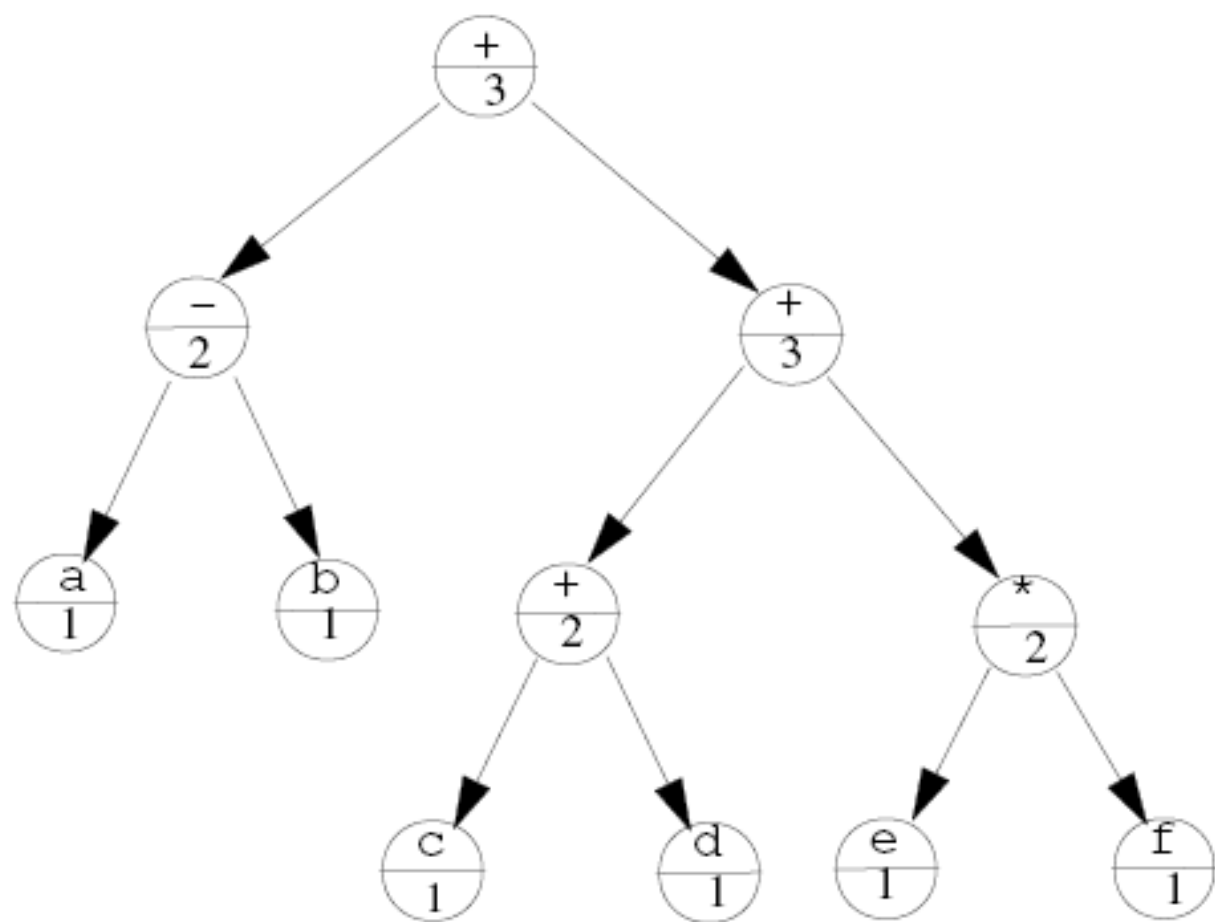


Figure 13.10: Expression Tree for $(a-b) + ((c+d)+(e*f))$ with Register Needs.

Resultierende code

“register targeting”



```
lw    $10, c           # Load c into register 10
lw    $11, d           # Load d into register 11
add   $10, $10, $11    # Compute c + d into register 10
lw    $11, e           # Load e into register 11
lw    $12, f           # Load f into register 12
mul   $11, $11, $12    # Compute e * f into register 11
add   $10, $10, $11    # Compute (c + d) + (e * f) into reg 10
lw    $11, a           # Load a into register 11
lw    $12, b           # Load b into register 12
sub   $11, $11, $12    # Compute a - b into register 11
add   $10, $11, $10    # Compute (a-b)+((c+d)+(e*f)) into reg 10
```

Figure 13.11: MIPS code for $(a-b) + ((c+d)+(e*f))$

procedure TREECG(*T, regList*)

Resultaat moet altijd in r1 komen

r1 \leftarrow HEAD(*regList*)

r2 \leftarrow HEAD(TAIL(*regList*))

if *T.kind* = *Identifier*

then

/* Load a variable. */

*/

call GENERATE(*lw, r1, T.IdentifierName*)

Load into r1

else

if *T.kind* = *IntegerLiteral*

then

/* Load a literal. */

*/

call GENERATE(*li, r1, T.IntegerValue*)

Load immediate into r1

else

/* *T.kind* must be a binary operator. */

*/

left \leftarrow *T.leftChild*

right \leftarrow *T.rightChild*

if *left.regCount* \geq LENGTH(*regList*) **and** *right.regCount* \geq LENGTH(*regList*)

then

/* Must spill a register into memory. */

*/

call TREECG(*left, regList*)

/* Get memory location. */

*/

temp \leftarrow GETTEMP()

call GENERATE(*sw, r1, temp*)

call TREECG(*right, regList*)

call GENERATE(*lw, r2, temp*)

Load temp into r2

/* Free memory location. */

*/

call FREETEMP(*temp*)

call GENERATE(*T.operation, r1, r2, r1*)

Result in r1

```

else
    /* There are enough registers; no spilling is needed. */
    if left.regCount ≥ right.regCount
    then
        call TREECG(left, regList)
        call TREECG(right, TAIL(regList))
        call GENERATE(T.operation, r1, r1, r2)
    else
        call TREECG(right, regList)
        call TREECG(left, TAIL(regList))
        call GENERATE(T.operation, r1, r2, r1)
end

```

Result in *r1*

Figure 13.12: An Algorithm to Generate Optimal Code from Expression Trees

Register allocatie over meerdere instructies

Het is wenselijk dat de associatie tussen variabelen en registers blijft bestaan gedurende meerdere “uses” van de variable.

Gewoonlijk zijn er verschillende klassen van registers:

- Alloceerbare registers (expliciet aangevraagd en vrijgegeven)
- Gereserveerde registers
- Werkregisters (under controle van de code generator)

Een veelgebruikte methode gebruikt **graph coloring**

Live ranges: start met een defining occurrence, eindigt met het laatste gebruik voor de volgende defining occurrence

Als 2 live ranges van variabelen niet overlappen, dan kan eenzelfde register gebruikt worden om hun waarde bij te houden (“no interference”)

Niet-overlappende ranges van eenzelfde variabele kunnen overeenkomen met verschillende registers

```
main() {  
    a = f(x);    // Start of first live range  
    print(a);    // End of first live range  
    ....  
    a = g(y)     // Start of second live range  
    print(a);    // End of second live range  
}
```

Veronderstel: hier geen a's

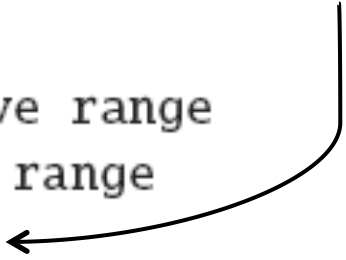


Figure 13.13: Example of Live Ranges

```

proc() {
    a = 100;
    b = 0;
    for (i=0;i<10;i++)
        b = b + i * i;
    print(a, b);
    c = 100;
    print(a*c);
}

```

Figure 13.14: A Simple Procedure with Candidates for Procedure-level Register Allocation.

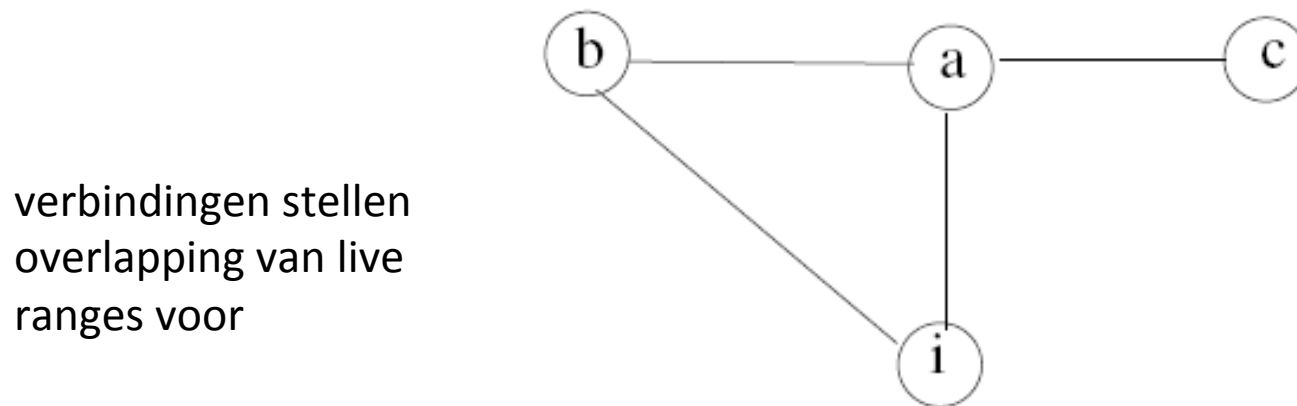


Figure 13.15: Interference Graph for procedure of Figure 13.14

Algoritme van Chaitin voor registerallocatie

Als er k registers beschikbaar zijn, dan komt het probleem om ze toe te kennen aan ranges (= knopen van de interference graph) overeen met het **graph coloring** probleem: ken k kleuren toe aan de knopen zo dat 2 buren altijd verschillende kleuren hebben – het is bekend dat dit NP compleet is.

Chaitin's algorithm gebruikt een heuristiek:

Als er een knoop n bestaat met minder dan k buren, dan kan hij altijd gekleurd worden; dus kleur eerst de graph die je bekomt door deze knoop weg te laten, en kies achteraf de kleur voor n

Opmerking: het algoritme lost het graph coloring probleem niet echt op!


```

procedure GCRegAlloc(proc, regCount)
    ig  $\leftarrow$  BUILDINTERFERENCEGRAPH(proc)
    stack  $\leftarrow$   $\emptyset$ 
    while ig  $\neq$   $\emptyset$  do
        if  $\exists d \in ig \mid neighborCount(d) < regCount$ 
        then
            ig  $\leftarrow ig - \{d\}$ 
            call PUSH(d)
        else
            d  $\leftarrow$  FINDSPILLNODE(ig)
            ig  $\leftarrow ig - \{d\}$ 
            /* Generate code to spill d's live range          */
        while stack  $\neq$   $\emptyset$  do
            d  $\leftarrow$  POP()
            reg(d)  $\leftarrow$  any register not assigned to neighbors(d)
    end
function FINDSPILLNODE(ig) returns Node
    bestCost  $\leftarrow \infty$ 
    foreach n  $\in ig$  do
        if  $\frac{cost(n)}{neighborCount(n)} < bestCost$ 
        then
            ans  $\leftarrow n$ 
            bestCost  $\leftarrow \frac{cost(n)}{neighborCount(n)}$ 
    return (ans)
end

```

Cost(*n*): bv. 10^i waar *i* = de nesting diepte van de live range die overeenkomt met *n*

Figure 13.16: Chaitin's graph coloring register allocator.

Code scheduling

De meeste processoren gebruiken **pipelining**: instructies werken in fazen, en verschillende instructies kunnen op hetzelfde ogenblik in verschillende fazen van uitvoering zijn.

Wat we willen vermijden is **stalling**: een situatie waar sommige van deze fazen moeten uitgesteld worden omdat de operanden nog niet beschikbaar zijn.

Voorbeeld:

```
li $11, 100
lw $12, b      (fetch b, kan traag zijn)
add $10, $11, $12  (stall, $12 nog niet klaar)
```

Waarschijnlijk is het volgende beter:

```
lw $12, b
li $11, 100
add $10, $11, $12
```

1. lw \$10,a	6. add \$10,\$10,\$12
2. lw \$11,b	7. mul \$11,\$11,\$10
3. mul \$11,\$10,\$11	8. mul \$12,\$10,\$12
4. lw \$10,c	9. add \$12,\$11,\$12
5. lw \$12,d	10. sw \$12,a

Figure 13.21: MIPS code for $a = ((a * b) * (c + d)) + (d * (c + d))$

Bouw de **dependency graph**: knopen zijn instructies, een pijl stelt het feit voor dat een instructie een operand levert voor een andere instructie

Dubbele circels: loads

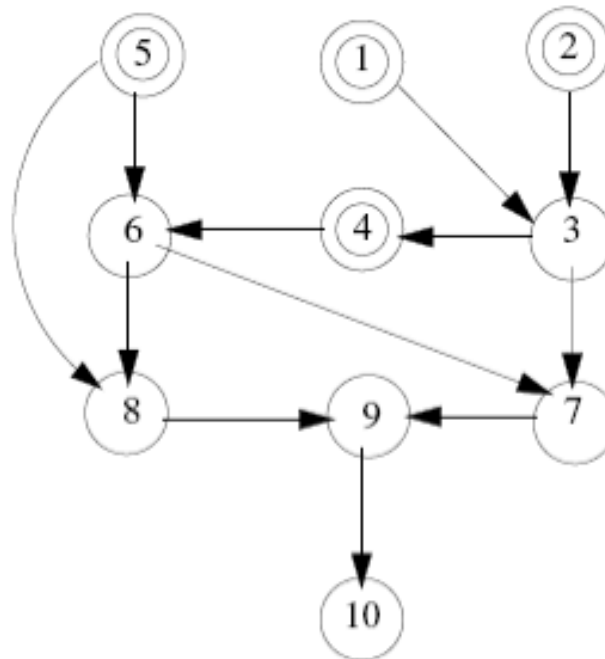


Figure 13.22: Dependency DAG for $a = ((a * b) * (c + d)) + (d * (c + d))$

```
procedure SCHEDULEDAG(dependencyDAG)  
  candidates  $\leftarrow$  ROOTS(dependencyDAG)  
  while candidates  $\neq \emptyset$  do  
    call SELECT(candidates, "Is not stalled by last instruction generated")  
    call SELECT(candidates, "Can stall some successor")  
    call SELECT(candidates, "Exposes the most new roots if generated")  
    call SELECT(candidates, "Has the longest path to a leaf")  
    inst  $\leftarrow$  Any node  $\in$  candidates  
    Schedule inst as next instruction to be executed  
    dependencyDAG  $\leftarrow$  dependencyDAG - { inst }  
    candidates  $\leftarrow$  ROOTS(dependencyDAG)  
end
```

Figure 13.23: An Algorithm to Schedule Code from a Dependency DAG

Selecteer, in die volgorde, instructies 1, 2, 5, 3, 4, 6, 7, 8, 9, 10 van de graph. Dat levert de volgende versie op

1. lw \$10,a	6. add \$10,\$10,\$12
2. lw \$11,b	7. mul \$11,\$11,\$10
3. lw \$12,d	8. mul \$12,\$10,\$12
4. mul \$11,\$10,\$11	9. add \$12,\$11,\$12
5. lw \$10,c	10 sw \$12,a

Figure 13.24: Scheduled MIPS code for $a = ((a * b) * (c + d)) + (d * (c + d))$

Nog altijd een stall na instructie 5 !

Met één extra register (\$13) kunnen we een versie bekomen zonder delays, dus zonder stalls:

1. lw \$10,a	6. add \$10,\$13,\$12
2. lw \$11,b	7. mul \$11,\$11,\$10
3. lw \$12,d	8. mul \$12,\$10,\$12
4. lw \$13,c	9. add \$12,\$11,\$12
5. mul \$11,\$10,\$11	10. sw \$12,a

Figure 13.25: Delay-free MIPS code for $a = ((a * b) * (c + d)) + (d * (c + d))$

Instructie - selectie

De meeste machines hebben instructies waarin verschillende adresseringswijzen worden gecombineerd: indexing, increments, Gevolg: er is geen **unieke** manier om een intermediate code instructie te vertalen.

bv. voor $a := a+1$:

`inc a` of `load a, add 1` ?

Systematische benadering: gebruik van low-level tree – structured intermediaire representatie.

i: global
b: global, array
a: local

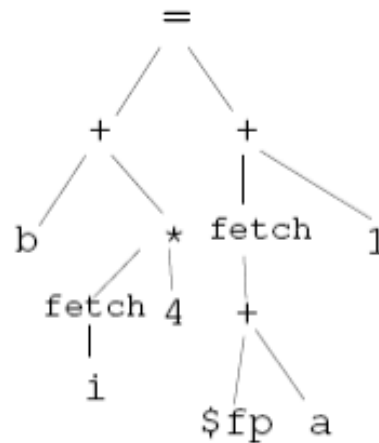


Figure 13.26: Low-Level IR Representation of $b[i]=a+1$

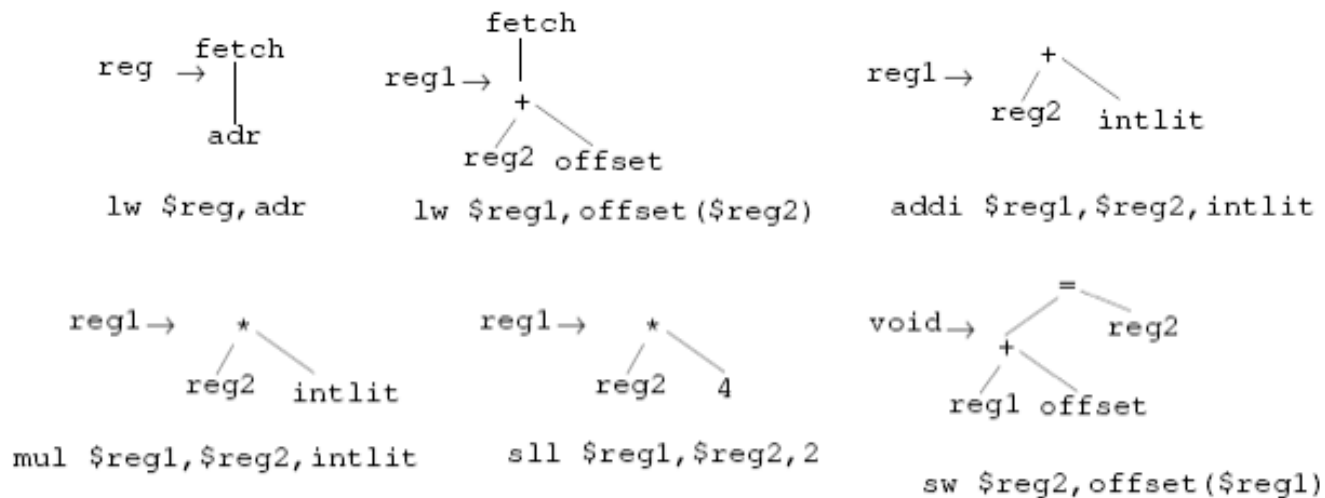


Figure 13.27: IR Tree Patterns for Various MIPS Instructions

Match de
rechterkanten,
vervang ze door de
overeenkomstige
linkerkanten

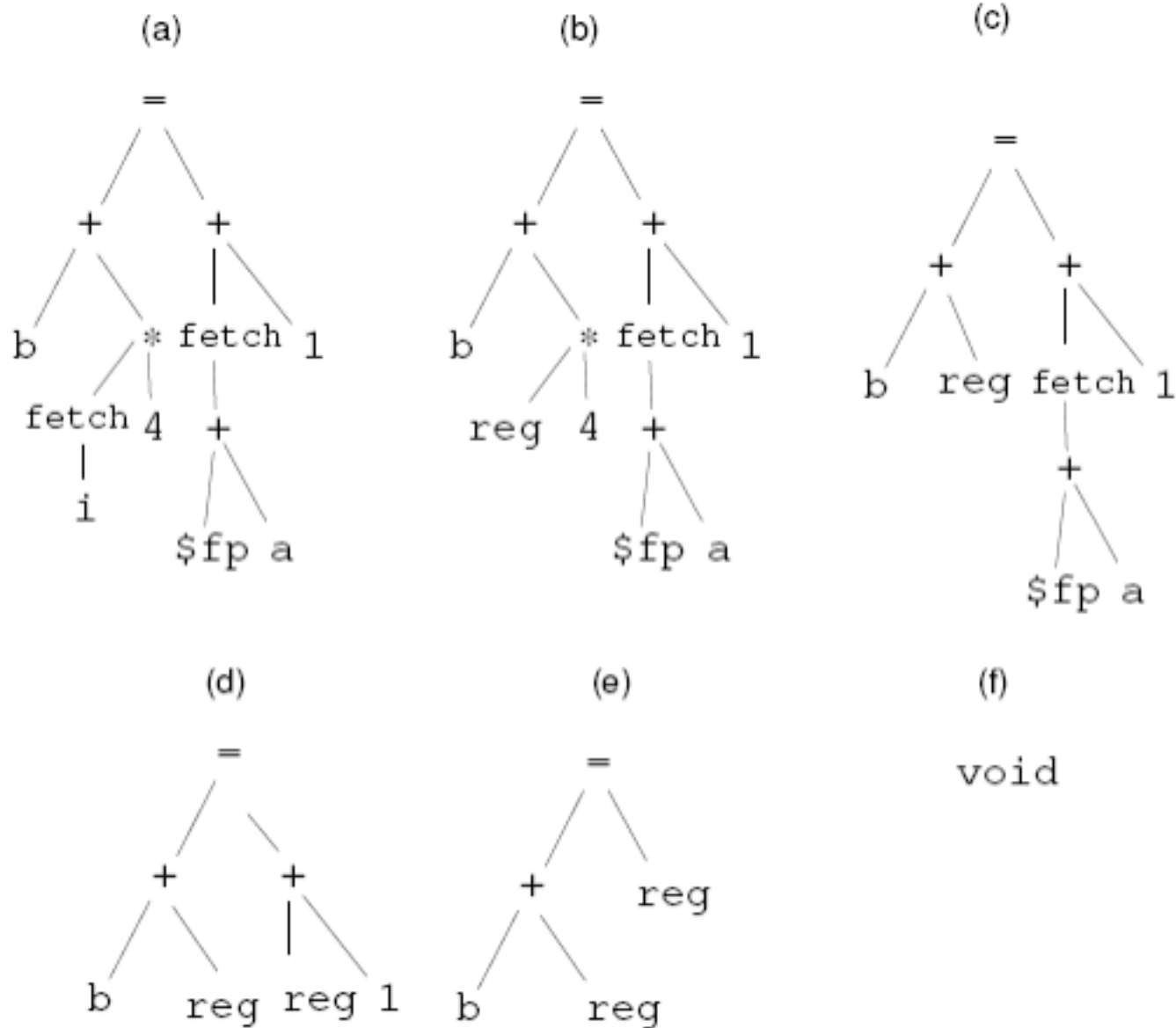


Figure 13.28: Instruction Selection Using Patterns

```
lw    $t1,i
mul   $t1,$t1,4
lw    $t2,a($fp)
addi  $t2,$t2,1
sw    $t2,b($t1)
```

Figure 13.29: MIPS code for $b[i]=a+1$

Rekening houdend
met het feit dat `sll`
goedkoper is dan
`mul`:

```
lw    $t1,i
sll   $t1,$t1,2
lw    $t2,a($fp)
addi  $t2,$t2,1
sw    $t2,b($t1)
```

BURS: vergelijkbaar
met shift-reduce
parsing

Figure 13.30: Improved MIPS code for $b[i]=a+1$

Peephole optimalisatie

Zoek naar speciale gevallen die verbeterd kunnen worden:

Product met 0

Add 0

Product met 1

...

Onderzoek een klein venster, om patronen te vinden die vervangen kan worden

Neemt weer de vorm aan van tree rewriting

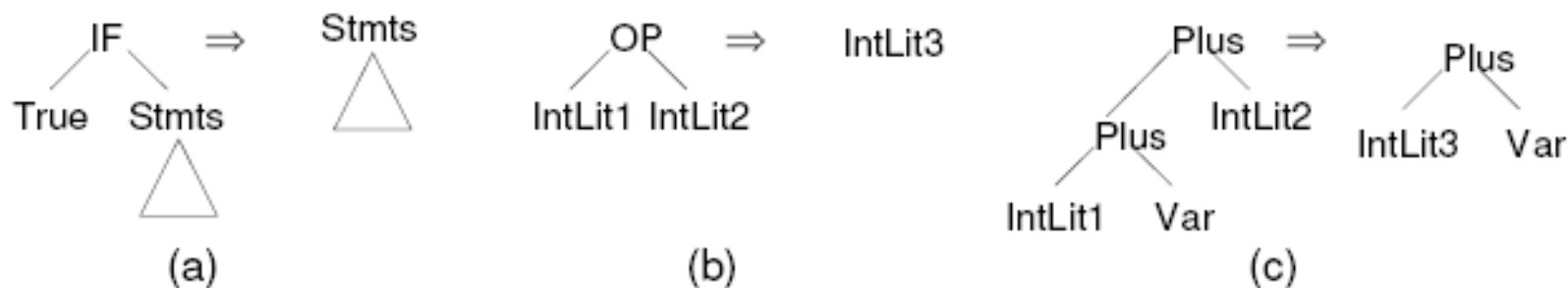


Figure 13.31: AST-Level Peephole Optimization

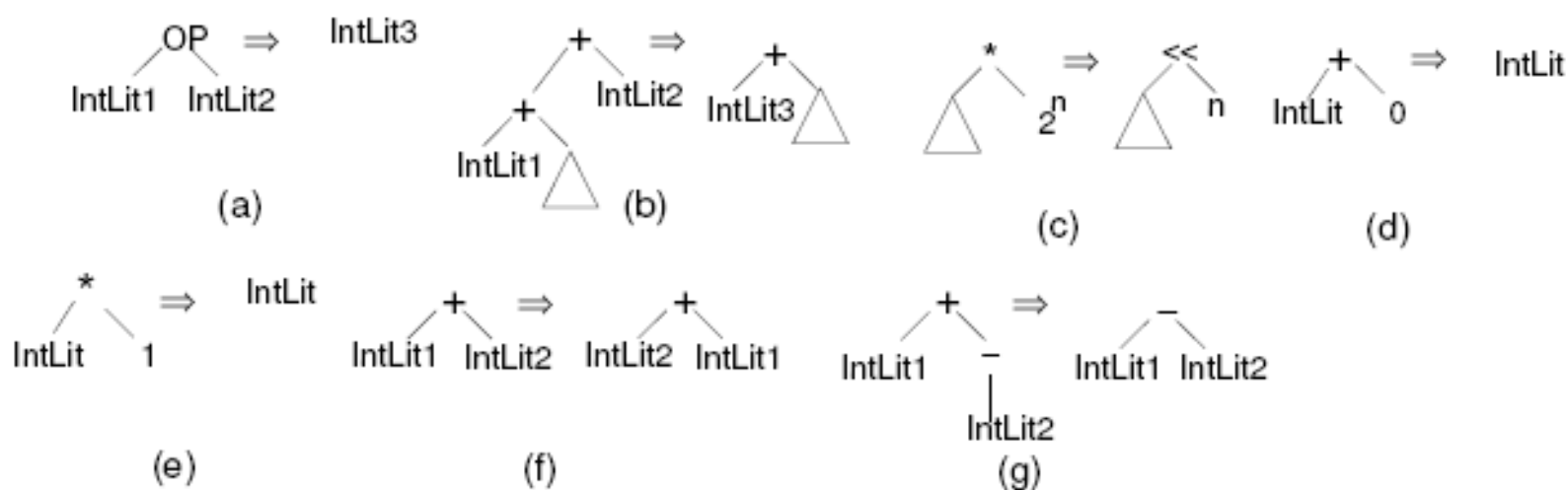


Figure 13.32: IR-Level Peephole Optimizations

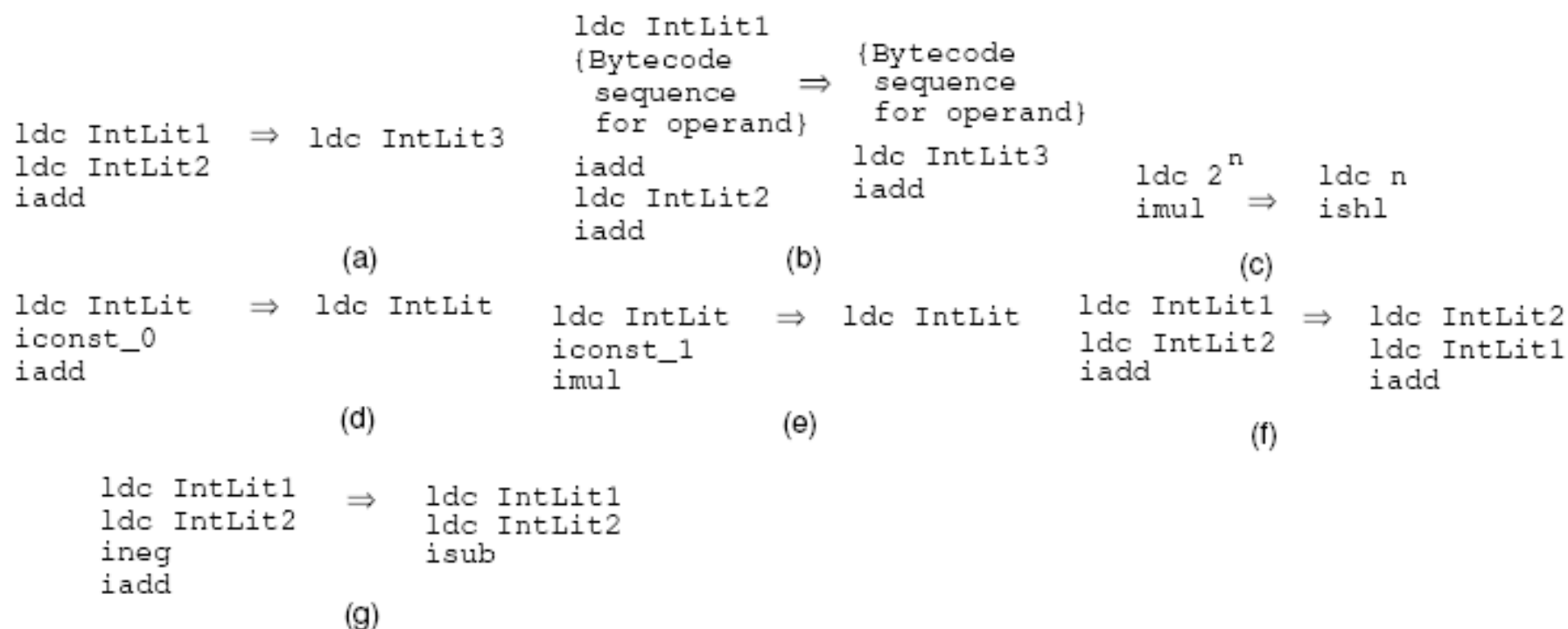


Figure 13.33: Bytecode-Level Peephole Optimizations

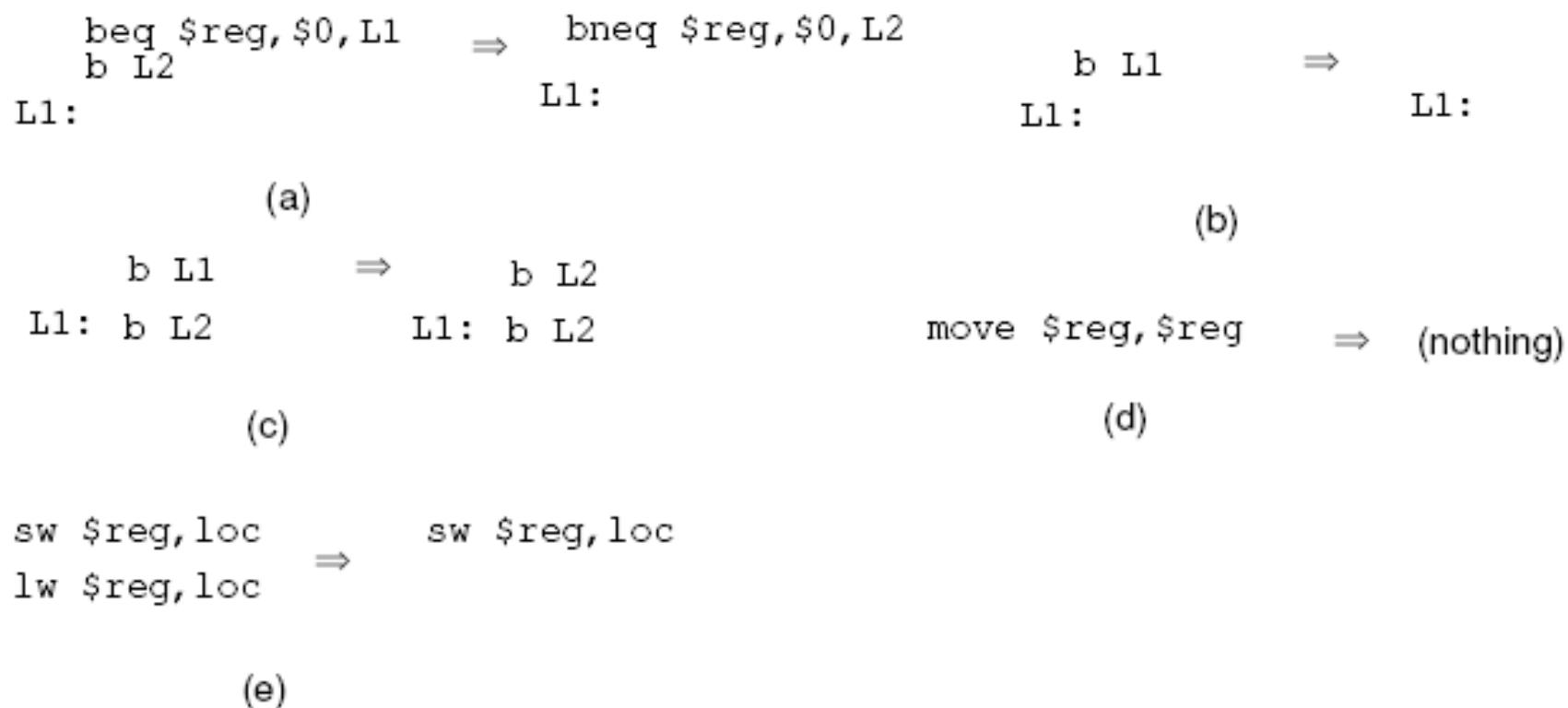


Figure 13.34: Code-Level Peephole Optimizations
