RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Register Allocation
Part 3

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

RUTGERS

Announcements

- First homework due this Friday, February 11.
  There is a two days extension from February 9
- First project (local instruction scheduler) will be posted later today
  Deadline for code: March 2
  Deadline for report: March 4
- Second homework has been posted

Local Register Allocation (and Assignment)

Readings: EaC 13.1-13.2, Appendix A (ILOC)

<u>Local</u>: within single basic block
<u>Global</u>: across procedure/function

RUTGERS

Live Ranges (live on exit)

Assume i and j are two instructions in a basic block

A **value** (virtual register) is *live* between its *definition* and its *uses*

- Find definitions (x ← …) and uses (y ← … x ...)
- From definition to <u>last</u> use is its *live range*
  - → How many (static) definitions can you have for a virtual register?
- Can represent live range as an interval [*i*,*j*] (in block)
  - → ***live on exit***

Let *MAXLIVE* be the maximum, over each instruction *i* in the block, of the number of values (virtual registers) live at *i*.

- If MAXLIVE ≤ *k*, allocation should be easy
  - → no need to reserve set of *F* registers for spilling
- If MAXLIVE > *k*, some values must be spilled to memory

Finding live ranges is harder in the global case

RUTGERS

Top-down Allocator

The idea:

- Machine has k physical registers
- Keep "busiest" values in an assigned register
- Use the feasible (reserved) set, *F*, for the rest
- *F* is the minimal set of registers needed to execute any instruction with all operands in memory:
  - → Move values with no assigned register from/to memory by adding LOADs and STOREs (SPILL CODE)
  - → For ILOC, the feasible set needs 3 registers due to StoreAO instruction

Basic algorithm (**not graph coloring**!):

- Rank values by number of occurrences (or some other metric)
- Allocate first *k – F* values to registers
- Rewrite code (with spill code) to reflect these choices

An Example : Top-Down

- Live Ranges

```
1 | loadI   1028     ⇒ r1
2 | load    r1       ⇒ r2
3 | mult    r1, r2   ⇒ r3
4 | loadI   5        ⇒ r4
5 | sub     r4, r2   ⇒ r5
6 | loadI   8        ⇒ r6
7 | mult    r5, r6   ⇒ r7
8 | sub     r7, r3   ⇒ r8
9 | store   r8       ⇒ r1
```

NOTE: live ranges (on exit) of each instruction

RUTGERS

An Example : Top-Down

- Live Ranges

```
1 | loadI   1028     ⇒ r1
2 | load    r1       ⇒ r2
3 | mult    r1, r2   ⇒ r3
4 | loadI   5        ⇒ r4
5 | sub     r4, r2   ⇒ r5
6 | loadI   8        ⇒ r6
7 | mult    r5, r6   ⇒ r7
8 | sub     r7, r3   ⇒ r8
9 | store   r8       ⇒ r1    WAIT: r1 is reused as a target register?
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1 | loadI   1028     ⇒ r1     // r1
2 | load    r1       ⇒ r2     // r1
3 | mult    r1, r2   ⇒ r3     // r1
4 | loadI   5        ⇒ r4     // r1
5 | sub     r4, r2   ⇒ r5     // r1
6 | loadI   8        ⇒ r6     // r1
7 | mult    r5, r6   ⇒ r7     // r1
8 | sub     r7, r3   ⇒ r8     // r1
9 | store   r8       ⇒ r1     //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1 | loadI   1028    ⇒ r1    // r1
2 | load    r1      ⇒ r2    // r1 r2
3 | mult    r1, r2  ⇒ r3    // r1 r2
4 | loadI   5       ⇒ r4    // r1 r2
5 | sub     r4, r2  ⇒ r5    // r1
6 | loadI   8       ⇒ r6    // r1
7 | mult    r5, r6  ⇒ r7    // r1
8 | sub     r7, r3  ⇒ r8    // r1
9 | store   r8      ⇒ r1    //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1 | loadI   1028     ⇒ r1    // r1
2 | load    r1       ⇒ r2    // r1 r2
3 | mult    r1, r2   ⇒ r3    // r1 r2 r3
4 | loadI   5        ⇒ r4    // r1 r2 r3
5 | sub     r4, r2   ⇒ r5    // r1    r3
6 | loadI   8        ⇒ r6    // r1    r3
7 | mult    r5, r6   ⇒ r7    // r1    r3
8 | sub     r7, r3   ⇒ r8    // r1
9 | store   r8       ⇒ r1    //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
loadI   1028     ⇒ r1    // r1
load    r1       ⇒ r2    // r1 r2
mult    r1, r2   ⇒ r3    // r1 r2 r3
loadI   5        ⇒ r4    // r1 r2 r3 r4
sub     r4, r2   ⇒ r5    // r1    r3    r5
loadI   8        ⇒ r6    // r1    r3    r5 r6
mult    r5, r6   ⇒ r7    // r1    r3          r7
sub     r7, r3   ⇒ r8    // r1                   r8
store   r8       ⇒ r1    //
```

NOTE: live ranges (on exit) of each instruction

RUTGERS

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - *k* = 4, *F* = 1, (*k*–*F*) = 3

Note: ILOC needs larger F set -> homework

```
1  loadI   1028    ⇒ r1    // r1
2  load    r1      ⇒ r2    // r1 r2
3  mult    r1, r2  ⇒ r3    // r1 r2 r3
4  loadI   5       ⇒ r4    // r1 r2 r3 r4
5  sub     r4, r2  ⇒ r5    // r1    r3    r5
6  loadI   8       ⇒ r6    // r1    r3    r5 r6
7  mult    r5, r6  ⇒ r7    // r1    r3          r7
8  sub     r7, r3  ⇒ r8    // r1                   r8
9  store   r8      ⇒ r1    //
```

- Consider statements with MAXLIVE > (*k-F*)   *basic algorithm*

Spill heuristic: - 1. number of occurrences of virtual register

- 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$

```
1  loadI   1028    ⇒ r1   // r1
2  load    r1      ⇒ r2   // r1 r2
3  mult    r1, r2  ⇒ r3   // r1 r2 r3
4  loadI   5       ⇒ r4   // r1 r2 r3 r4             -- MAXLIVE = 4
5  sub     r4, r2  ⇒ r5   // r1    r3    r5
6  loadI   8       ⇒ r6   // r1    r3    r5 r6       -- MAXLIVE = 4
7  mult    r5, r6  ⇒ r7   // r1    r3          r7
8  sub     r7, r3  ⇒ r8   // r1                  r8
9  store   r8      ⇒ r1   //
```

- Consider statements with MAXLIVE > *(k-F)*   *basic algorithm*

  Spill heuristic: - 1. number of occurrences of virtual register

  - 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - *k* = 4, *F* = 1, (*k*–*F*) = 3

	#occ.	length
r1:	4	8
r2:	3	3
r3:	2	5
r4:	2	1
r5:	2	2
r6,r7,r8:	2.	1

```
1  loadI    1028     ⇒ r1    // r1
2  load     r1       ⇒ r2    // r1 r2
3  mult     r1, r2   ⇒ r3    // r1 r2 r3
4  loadI    5        ⇒ r4    // r1 r2 r3 r4          -- MAXLIVE = 4
5  sub      r4, r2   ⇒ r5    // r1    r3    r5
6  loadI    8        ⇒ r6    // r1    r3    r5 r6    -- MAXLIVE = 4
7  mult     r5, r6   ⇒ r7    // r1    r3          r7
8  sub      r7, r3   ⇒ r8    // r1                   r8
9  store    r8       ⇒ r1    //
```

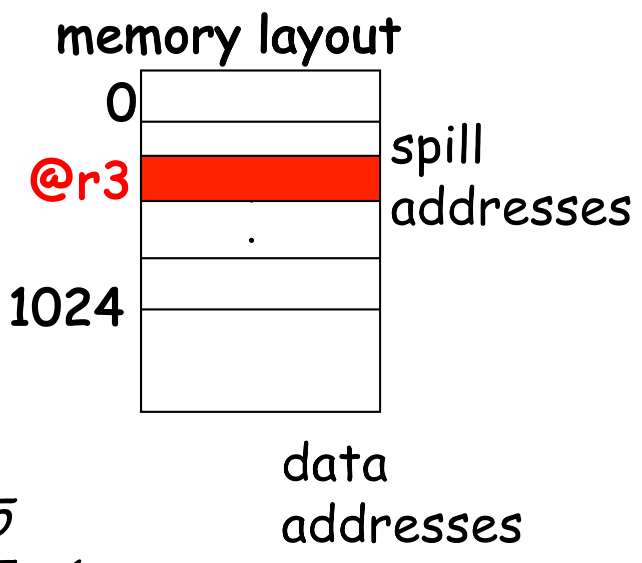
- Consider statements with MAXLIVE > (*k-F*)   *basic algorithm*

  Spill heuristic: - 1. number of occurrences of virtual register
  - 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$

```
1  loadI  1028    ⇒ r1   // r1
2  load   r1      ⇒ r2   // r1 r2
3  mult   r1, r2  ⇒ r3   // r1 r2 r3
4  loadI  5       ⇒ r4   // r1 r2 r3 r4
5  sub    r4, r2  ⇒ r5   // r1    r3    r5
6  loadI  8       ⇒ r6   // r1    r3    r5 r6
7  mult   r5, r6  ⇒ r7   // r1    r3          r7
8  sub    r7, r3  ⇒ r8   // r1                   r8
9  store  r8      ⇒ r1   //
```



- Consider statements with MAXLIVE > $(k-F)$ *basic algorithm*

  Spill heuristic: - 1. number of occurrences of virtual register
  - 2. length of live range (tie breaker)

Note: EAC Top down algorithm does not look at live ranges and MAXLIVE, but counts overall occurrences across entire basic block

RUTGERS

An Example : Top-Down

- 3 physical registers for allocation: ra, rb, rc
- 1 physical register designated to be in the feasible set *F*

```
1  loadI    1028      ⇒ ra      // r1
2  load     ra        ⇒ rb      // r1 r2
3  mult     ra, rb    ⇒ f1      // r1 r2
   storeAI  f1  ⇒ r0, @r3      // spill code
4  loadI    5         ⇒ rc      // r1 r2   r4              -- MAXLIVE = 3
5  sub      rc, rb    ⇒ rb      // r1        r5
6  loadI    8         ⇒ rc      // r1        r5 r6.        -- MAXLIVE = 3
7  mult     rb, rc    ⇒ rb      // r1              r7
   loadAI r0, @r3     ⇒ f1      // spill code
8  sub      rb, f1    ⇒ rb      // r1                  r8
9  store    rb        ⇒ ra      //
```

- Insert spill code for every occurrence of spilled virtual register in basic block using feasible register **f1**; Remove spilled register from consideration for allocation

Spill code

- A virtual register is spilled by using only registers from the feasible set (F), not the allocated set (k-F) = {ra, rb, … }
- How to insert spill code, with F = {f1, f2, … }?
  - → For the definition of the spilled value r (assignment of the value to the virtual register r), use a feasible register f as the target register; assign fixed memory location for spilled value (spill offset @r), and store feasible register value to address offset @r:

```
add ra, rb  ⇒ f          // target of operation is spilled;
storeAI f ⇒ r0, @r   // spilled value "lives" in memory offset @r
```

  - → For the use of the spilled value, load value from memory into a feasible register:

```
loadAI r0, @r ⇒ f     // value lives at memory with offset @r
add f, rb  ⇒ ra          // loaded into feasible register
```

- How many feasible registers do we need for an *add* instruction?

RUTGERS

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values “used soon” in registers
- Only parts of a live range may be assigned to a physical register (≠ top-down allocation’s “all-or-nothing” approach)

Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

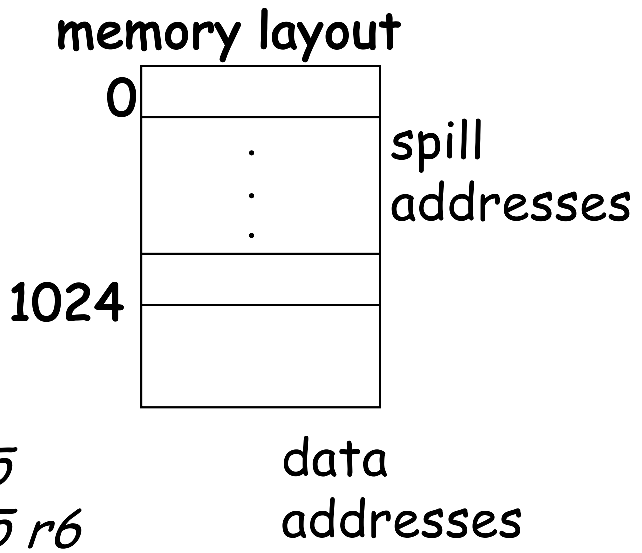
Replacement (heuristic):

- Spill the value whose next use is farthest in the future
- Sound familiar? Think page replacement ...

An Example : Bottom-up

- Bottom up (3 registers to allocate)

```
1 | loadI   1028     ⇒ r1    // r1
2 | load    r1       ⇒ r2    // r1 r2
3 | mult    r1, r2   ⇒ r3    // r1 r2 r3
4 | loadI   5        ⇒ r4    // r1 r2 r3 r4
5 | sub     r4, r2   ⇒ r5    // r1    r3    r5
6 | loadI   8        ⇒ r6    // r1    r3    r5 r6
7 | mult    r5, r6   ⇒ r7    // r1    r3          r7
8 | sub     r7, r3   ⇒ r8    // r1                   r8
9 | store   r8       ⇒ r1    //
```



RUTGERS

Next topic

Bottom-up register allocation

Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;