CS614: Advanced Compilers

Bitwidth-Aware Register Allocation

Manas Thakur
CSE, IIT Bombay

Spring 2025

Subword access

- Programs that manipulate data at subword level are common in embedded systems, network applications, and multimedia codes.
- Several architectures (e.g. ARM) provide ability to read/write at bit-section level.

e.g., $R1_{0..7} \leftarrow R2_{0..3} + R3_{0..5}$

- Operands extended by leading zero bits to match the size of the result.

- **Idea:**
  - If we knew the number of relevant bits in each variable at each program point, we could try to pack multiple variables into one register.
  - Bitwidth-aware register allocation (Tallam and Gupta, POPL 2003).

Example

Assumption: All variables are 8-bit chars.

Unpacking

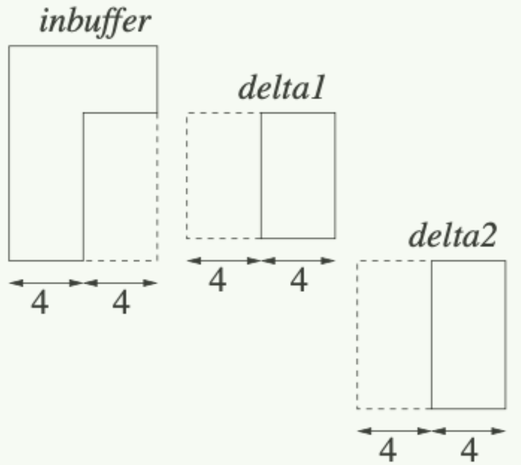
```
inbuffer = ...
...
delta1 = inbuffer & 0xf
...
last use of delta1
delta2 = (inbuffer >> 4)
         & 0xf
...
last use of delta2
```

Packing

```
alpha1 = ...
...
alpha2 = ...
...
outbuffer =
      (alpha2 & 0x7) |
      ((alpha1 & 0xf) << 3))
...
```

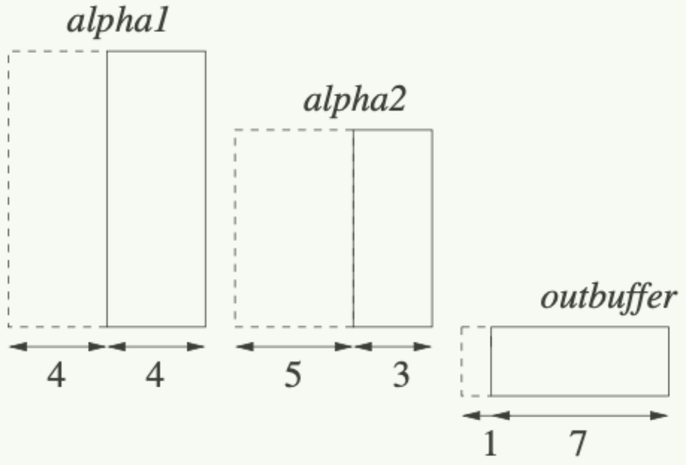
Unpacking

```
char inbuffer;
char delta1, delta2;
```



Packing

```
char outbuffer;
char alpha1, alpha2;
```



Bitwidth-Aware Register Allocation

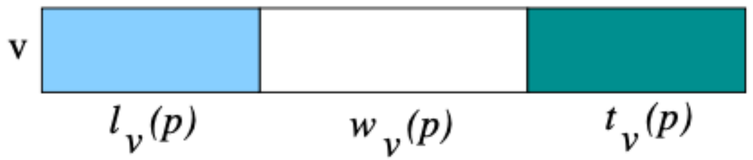
- **Recall:** If we knew the number of *relevant* bits in each variable at each program point, we could try to pack multiple variables into one register.

- **Two steps:**

  1. Bit-aware live-range construction
     - Variables are often declared to be of larger bitwidths than needed
     - Variables may not have the same relevant bitwidth throughout the program
  2. Packing multiple variables into a register
     - Shapes of live ranges to be understood to decide which variables to pack at what points of the program

Definitions

- Dead bits. Given a variable `v`, which according to its declaration is represented by `s` bits, a subset of these `s` bits, say `d`, are *dead* at program point `p` if all computations following point `p` that use the current value of `v` can be performed without explicitly referring to the bits in `d`.
- Live-range width. Given a program point `p` in variable `v`'s live range, the *width of* `v`'s *live range* at point `p`, denoted by $w_v(p)$, is defined such that the bits representing variable `v` according to its declaration can be divided into three contiguous sections as follows: a *leading section* of $l_v(p)$ *dead* bits; a *middle section* of $w_v(p)$ *live* bits; and a *trailing section* of $t_v(p)$ *dead* bits.



Bits that are not used (NOUSE)

- NOUSE(s_v,v) is a pair (l,t) denoting the number of leading (l) and trailing (t) bits of v not used at statement s_v.

s_v	Characteristics of s_v	$NOUSE(s_v, v)$
$v >> t$ $v << l$ $v \ \& \ c$	t is a compile time constant l is a compile time constant c is a compile time constant with l leading and t trailing zero bits	$(0, t)$ - t trailing bits of v are not used. $(l, 0)$ - l leading bits of v are not used. (l, t) - l leading bits and t trailing bits of v are not used.
$v \ \ op \ \ ...$ $v \ \mid \ ...$	op is an arithmetic or relational operator; v has at least l leading zero bits v has at least l leading zero bits and t trailing zero bits	$(l, 0)$ - l leading bits of v are not used. (l, t) - l leading bits and t trailing bits of v are not used.
$f(v)$	other forms of statements that use v	$(0, 0)$ - all bits of v are used.

- NOUSE can be computed if we have information about the bits that are guaranteed to be zero (Zero Bit Sections, ZBS) at each statement.

Dead Bit-Sections (DBS) Analysis

- $\mathtt{DBS_{in}[n,v]}$ tells us the number of leading and trailing dead bits of a variable v before statement n.
- $\mathtt{DBS_{out}[n,v]}$ tells us the number of leading and trailing dead bits of a variable v after statement n.
- $\top$ in DBS means all bits of the corresponding variable.

$$(l_1,t_1)\wedge(l_2,t_2) = (\min(l_1,l_2),\min(t_1,t_2))$$

$$DBS_{in}[n,v] := \begin{cases} NOUSE(n,v) \wedge DBS_{out}[n,v] & if\ n\ uses\ v \\ (\top,\top) & elseif\ n\ defines\ v \\ DBS_{out}[n,v] & otherwise \end{cases}$$

$$DBS_{out}[n,v] := \bigwedge_{s \in Succ(n)} (DBS_{in}[s,v]).$$

- ZBS and NOUSE are forward analyses; then DBS backward; fixed-point.

Zero Bit-Sections (ZBS) Analysis

$$ZBS_{in}[n,v] := \bigwedge_{p \in Pred(n)} (ZBS_{out}[p,v]).$$

$$ZBS_{out}[n,v] := \begin{cases} (l,t) & if\ n\ is\ v = c;\ c\ is\ a\ +ve\ constant;\ and \\ & \quad (l,t)\ represents\ c's\ zero\ bit\ sections \\ (l+c, t-c) \triangle (\top, 0) & elseif\ n\ is\ v = x\ >>\ c;\ ZBS_{in}[n,x] = (l,t)\ and \\ & \quad c\ is\ a\ +ve\ constant \\ (l-c, t+c) \nabla (0, \top) & elseif\ n\ is\ v = x\ <<\ c;\ ZBS_{in}[n,x] = (l,t)\ and \\ & \quad c\ is\ a\ +ve\ constant \\ ZBS_{in}[n,x] & elseif\ n\ is\ v = x \\ ZBS_{in}[n,x] \wedge ZBS_{in}[n,y] & elseif\ n\ is\ v = x \mid y \\ ZBS_{in}[n,x] \vee ZBS_{in}[n,y] & elseif\ n\ is\ v = x\ \&\ y \\ (0,0) & elseif\ n\ defines\ v \\ ZBS_{in}[n,v] & otherwise \end{cases}$$

Let's ignore this slide for now :D

$$\begin{aligned} &(l_1,t_1) \wedge (l_2,t_2) = (min(l_1,l_2), min(t_1,t_2)). \\ &(l_1,t_1) \vee (l_2,t_2) = (max(l_1,l_2), max(t_1,t_2)). \\ &(l_1,t_1) \triangle (l_2,t_2) = (min(l_1,l_2), max(t_1,t_2)). \\ &(l_1,t_1) \nabla (l_2,t_2) = (max(l_1,l_2), min(t_1,t_2)). \end{aligned}$$

Example: ZBS

$int\ A;$ – 32 bits $short\ D, E;$ – 16 bits $char\ B, C;$ – 8 bits 1. $E = \ldots$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4)\|0xf$ $this\ was\ E$'s $last\ use.$ 5. $use\ A$ 6. $A = A >> 12$ 7. $B = \ldots$ 8. $C = (B\&0x7f) + 1$ 9. $last\ use\ of\ A$ 10. $last\ use\ of\ B\&0x80$ 11. $last\ use\ of\ C$ 12. $last\ use\ of\ D$	Zero Bit Sections 1. E: (0,0) 2. D: (0,0) 3. D: (4,0) 4. A: (12,0) 5. 6. A: (24,0) 7. B: (0,0) 8. C: (0,0) 9. 10. 11. 12.

Example: DBS

Variable name appearing alone represents all its bits are dead; no name indicates all its bits are used.

$$DBS_{in}[n,v] := \begin{cases} NOUSE(n,v) \wedge DBS_{out}[n,v] & if\ n\ uses\ v \\ (\top,\top) & elseif\ n\ defines\ v \\ DBS_{out}[n,v] & otherwise \end{cases}$$

$$DBS_{out}[n,v] := \bigwedge_{s \in Succ(n)} (DBS_{in}[s,v]).$$

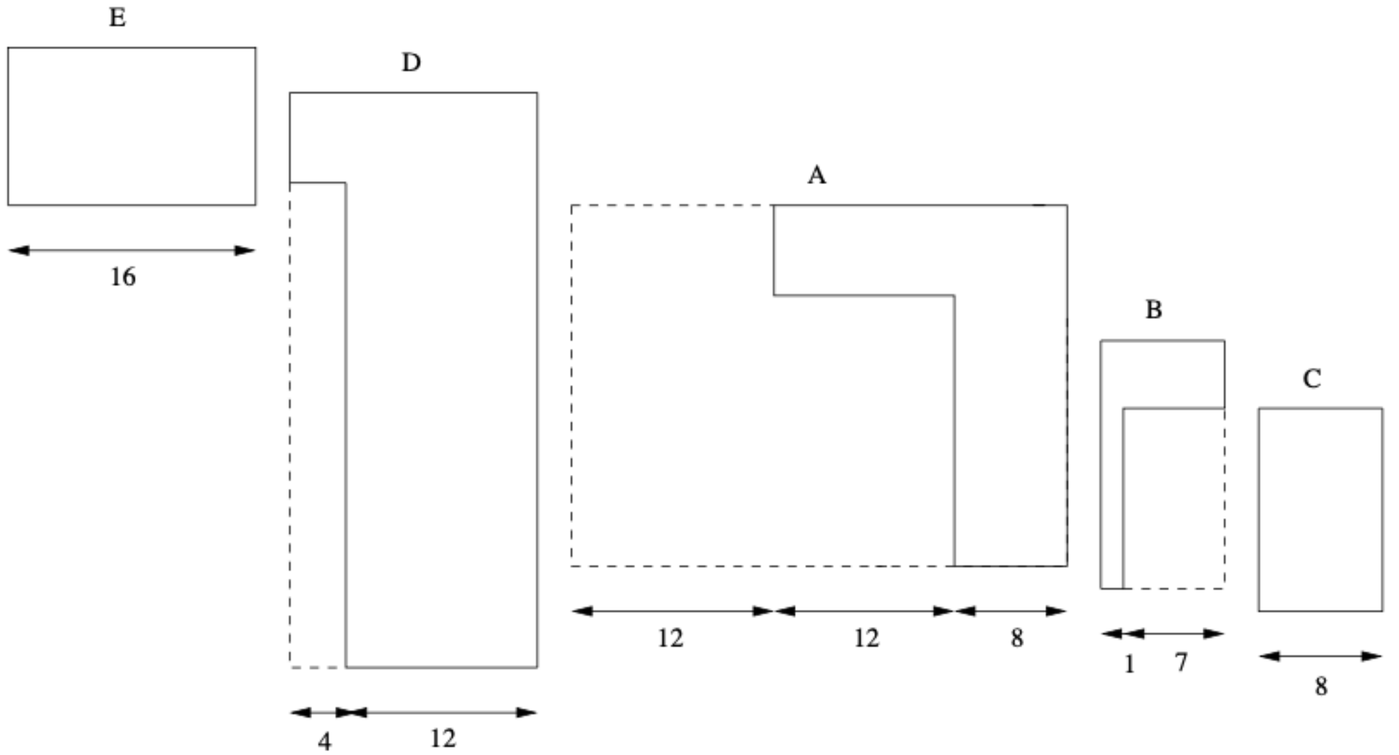
s_v	Characteristics of s_v	$NOUSE(s_v, v)$
$v >> t$	t is a compile time constant	$(0,t)$ - t trailing bits of v are not used.
$v << l$	l is a compile time constant	$(l,0)$ - l leading bits of v are not used.
v & c	c is a compile time constant with l leading and t trailing zero bits	(l,t) - l leading bits and t trailing bits of v are not used.
v *op* ...	*op* is an arithmetic or relational operator; v has at least l leading zero bits	$(l,0)$ - l leading bits of v are not used.
v | ...	v has at least l leading zero bits and t trailing zero bits	(l,t) - l leading bits and t trailing bits of v are not used.
$f(v)$	other forms of statements that use v	$(0,0)$ - all bits of v are used.

int A; – 32 bits *short* D, E; – 16 bits *char* B, C; – 8 bits 1. $E = \ldots$ 2. $D = E + 1$ 3. $D = D >> 4$ 4. $A = (E << 4) \mid 0xf$ *this was* E*'s last use.* 5. *use* A 6. $A = A >> 12$ 7. $B = \ldots$ 8. $C = (B \& 0x7f) + 1$ 9. *last use of* A 10. *last use of* $B \& 0x80$ 11. *last use of* C 12. *last use of* D	Zero Bit Sections 1. E: (0,0) 2. D: (0,0) 3. D: (4,0) 4. A: (12,0) 5. 6. A: (24,0) 7. B: (0,0) 8. C: (0,0) 9. 10. 11. 12.	Dead Bit Sections 0. A:B:C:D:E; *before* 1 1. A:B:C:D 2. A:B:C 3. A:B:C:D(4,0) 4. A(12,0):B:C:D(4,0):E 5. A(12,0):B:C:D(4,0):E 6. A(24,0):B:C:D(4,0):E 7. A(24,0):C:D(4,0):E 8. A(24,0):B(0,7):D(4,0):E 9. A:B(0,7):D(4,0):E 10. A:B:D(4,0):E 11. A:B:C:D(4,0):E 12. A:B:C:D:E

OUT sets.

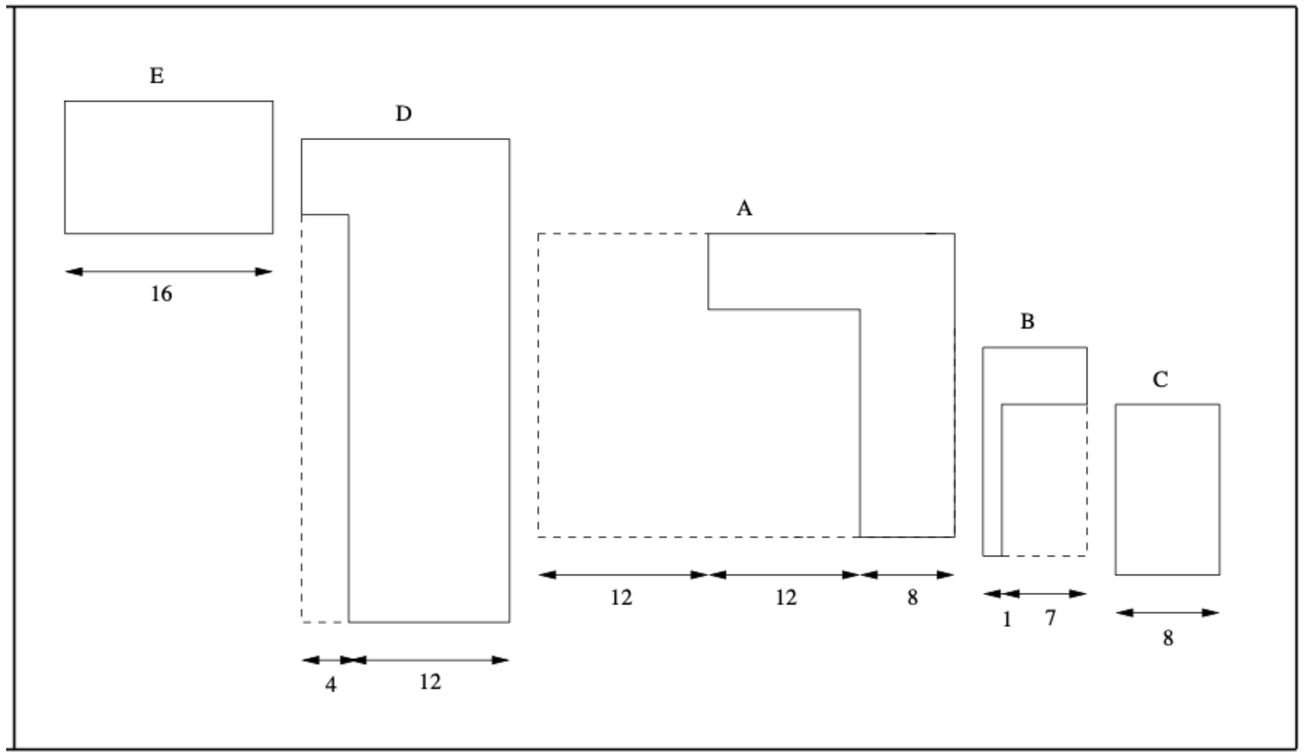
Example: Live-Range Construction

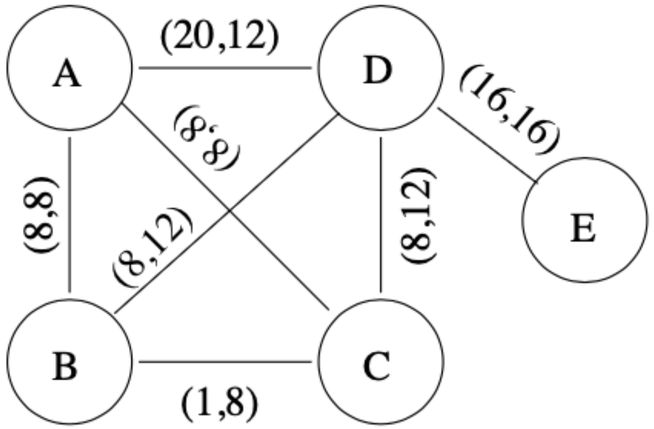
Dead
Bit Sections
0. A:B:C:D:E; $before$ 1
1. A:B:C:D
2. A:B:C
3. A:B:C:D(4,0)
4. A(12,0):B:C:D(4,0):E
5. A(12,0):B:C:D(4,0):E
6. A(24,0):B:C:D(4,0):E
7. A(24,0):C:D(4,0):E
8. A(24,0):B(0,7):D(4,0):E
9. A:B(0,7):D(4,0):E
10. A:B:D(4,0):E
11. A:B:C:D(4,0):E
12. A:B:C:D:E



Variable Packing

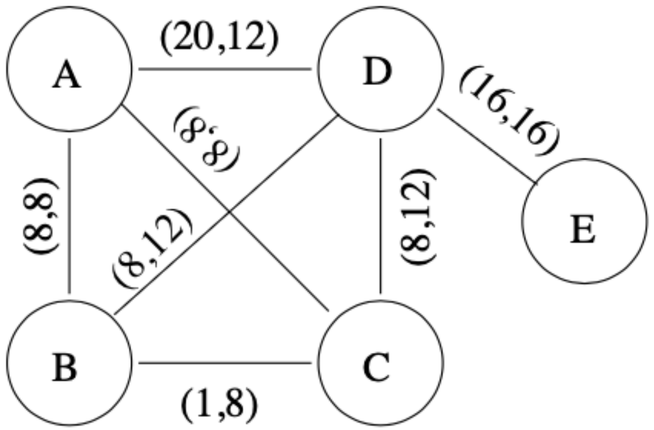
- **STEP 1:** Use bitwise live-ranges to draw extended interference graphs.
- An edge between X and Y is labeled with (p,q), where p and q are the contributions of X and Y, respectively, to the *maximum interference width* in their live ranges.

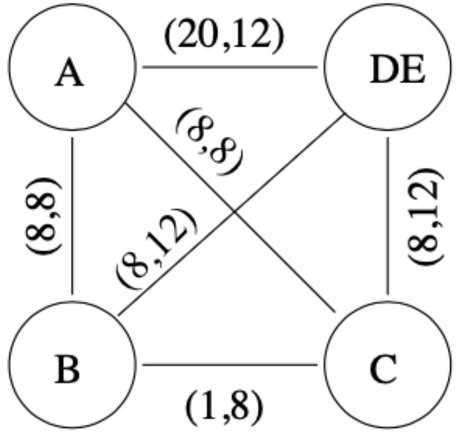


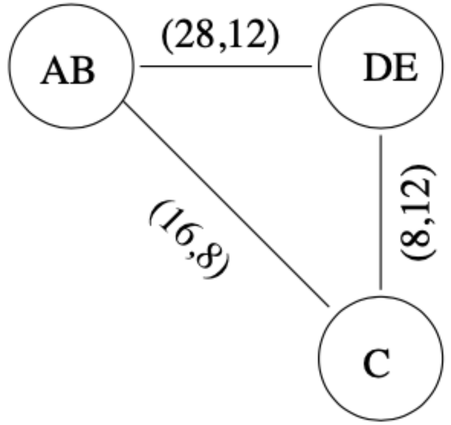


Variable Packing (Cont.)

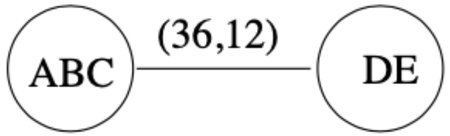
- **STEP 2:** Iteratively coalesce nodes that can be packed together.







Each register is 32 bits.



Therefore, we would in total need two registers.

How many with standard graph-coloring based RA?

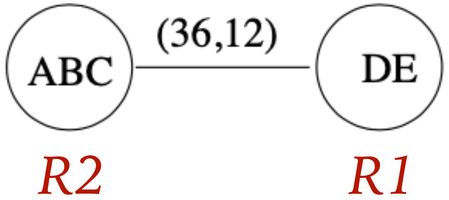
Assign registers and generate code

Original code:

1. $E = \ldots$
2. $D = E + 1$
3. $D = D >> 4$
4. $A = (E << 4)|0xf$
   ; *this was E's last use.*
5. *use* A
6. $A = A >> 12$
7. $B = \ldots$
8. $C = (B\&0x7f) + 1$
9. *last use of* A
10. *last use of* $B\&0x80$
11. *last use of* C
12. *last use of* D

int A;	– 32 bits
short D, E;	– 16 bits
char B, C;	– 8 bits

Dead
Bit Sections
0. A:B:C:D:E; *before* 1
1. A:B:C:D
2. A:B:C
3. A:B:C:D(4,0)
4. A(12,0):B:C:D(4,0):E
5. A(12,0):B:C:D(4,0):E
6. A(24,0):B:C:D(4,0):E
7. A(24,0):C:D(4,0):E
8. A(24,0):B(0,7):D(4,0):E
9. A:B(0,7):D(4,0):E
10. A:B:D(4,0):E
11. A:B:C:D(4,0):E
12. A:B:C:D:E



Code after register allocation:

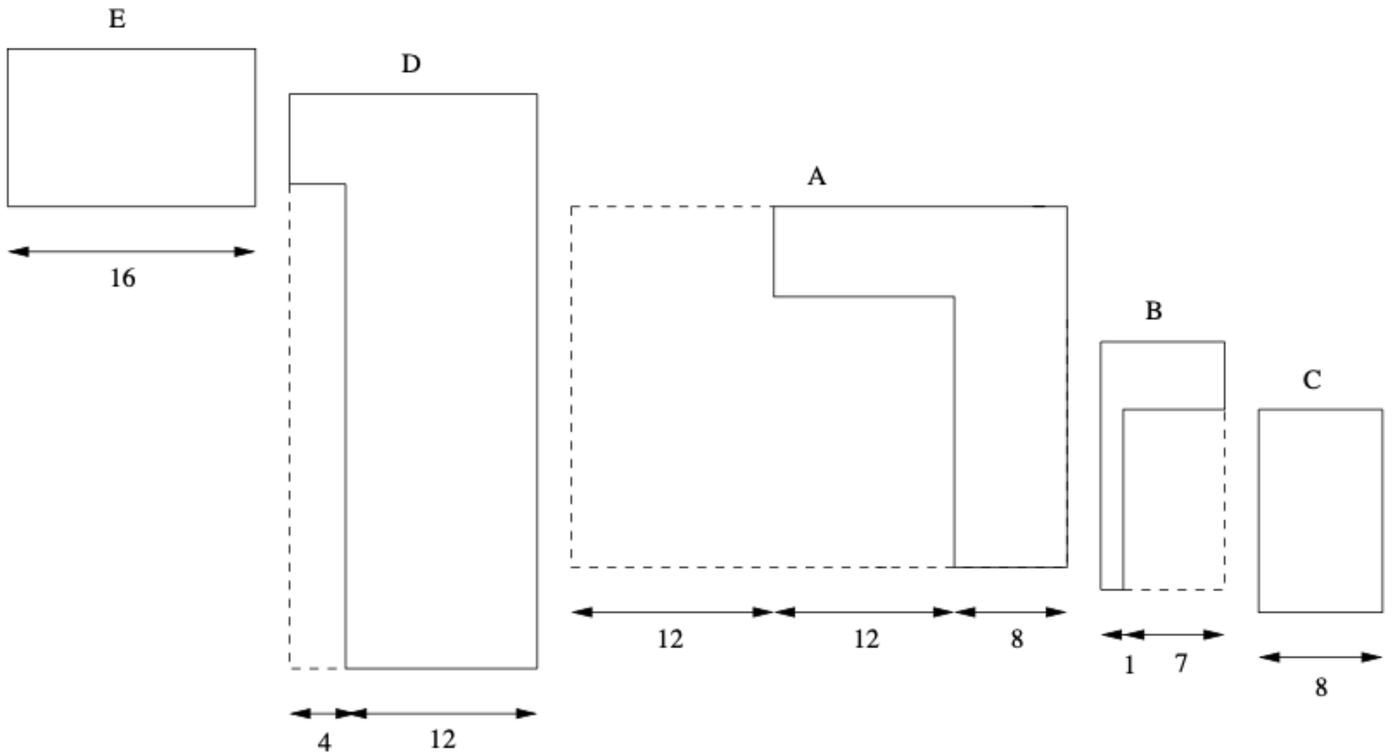
1. $R1_{0..15} = \ldots$
2. $R1_{16..31} = R1_{0..15} + 1$
3. $R1_{16..27} = R1_{20..31}$
4. $R2_{4..19} = R1_{0..15}$; $R2_{0..3} = 0xf$
5. *use* $R2_{0..19}$
6. $R2_{0..7} = R2_{12..19}$
7. $R2_{8..15} = \ldots$
8. $R2_{16..23} = R2_{8..14} + 1$
9. *last use of* $R2_{0..7}$
10. *last use of* $R2_{15}$
11. *last use of* $R2_{16..23}$
12. *last use of* $R1_{16..27}$

Note the little-endian format!

Could we have done better?

How many maximum bits do we need at a given point of time?

Dead
Bit Sections
0. A:B:C:D:E; *before* 1
1. A:B:C:D
2. A:B:C
3. A:B:C:D(4,0)
4. A(12,0):B:C:D(4,0):E
5. A(12,0):B:C:D(4,0):E
6. A(24,0):B:C:D(4,0):E
7. A(24,0):C:D(4,0):E
8. A(24,0):B(0,7):D(4,0):E
9. A:B(0,7):D(4,0):E
10. A:B:D(4,0):E
11. A:B:C:D(4,0):E
12. A:B:C:D:E



Yes!

Original code:

1. $E = \ldots$
2. $D = E + 1$
3. $D = D >> 4$
4. $A = (E << 4)|0xf$
   ; *this was E's last use.*
5. *use* A
6. $A = A >> 12$
7. $B = \ldots$
8. $C = (B\&0x7f) + 1$
9. *last use of* A
10. *last use of* $B\&0x80$
11. *last use of* C
12. *last use of* D

Code using 2 registers:

1. $R1_{0..15} = \ldots$
2. $R1_{16..31} = R1_{0..15} + 1$
3. $R1_{16..27} = R1_{20..31}$
4. $R2_{4..19} = R1_{0..15}; R2_{0..3} = 0xf$
5. *use* $R2_{0..19}$
6. $R2_{0..7} = R2_{12..19}$
7. $R2_{8..15} = \ldots$
8. $R2_{16..23} = R2_{8..14} + 1$
9. *last use of* $R2_{0..7}$
10. *last use of* $R2_{15}$
11. *last use of* $R2_{16..23}$
12. *last use of* $R1_{16..27}$

Code using 1 register:

1. $R_{0..15} = \ldots$
2. $R_{16..31} = R_{0..15} + 1$
3. $R_{16..27} = R_{20..31}$
   $\boxed{R_{20..31} = R_{16..27}}$
4. $R_{4..19} = R_{0..15}; R_{0..3} = 0xf$
5. *use* $R_{0..19}$
6. $R_{0..7} = R_{12..19}$
7. $R_{8..15} = \ldots$
   $\boxed{R_{16} = R_{15}}$
8. $R_{8..15} = R_{8..14} + 1$
9. *last use of* $R_{0..7}$
10. *last use of* R_{16}
11. *last use of* $R_{8..15}$
12. *last use of* $R_{20..31}$

Would require more analysis.

RA: The Conclusion

- Register allocation is important: ***directly impacts the execution*** profile of a program.
- Several ***improvements to graph coloring*** have been proposed to target better RA.
- ***SSA*** (and its destruction, in particular) has interesting effects on RA.
- There is also work on ***validating*** RA.
- RA has interesting ***ordering*** tradeoffs with other phases of code generation.
  - which we will study after the midsem!