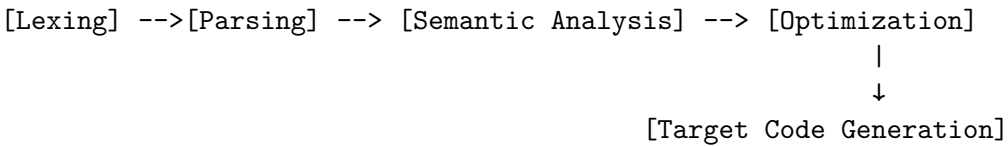
50.054 Code Generation (3 Address)

ISTD, SUTD

Learning Outcomes

1. Apply SSA-based register allocation to generate 3-address code from Pseudo Assembly
2. Handle register spilling

Recap Compiler Pipeline



- Target Code Generation
  - Input: some IR as input
  - Output: the target code (executable)

Instruction Selection

- 3-address instruction
  - RISC (Reduced Instruction Set Computer) architecture. E.g. Apple PowerPC, ARM, Pseudo Assembly
- 2-address instruction
  - CISC (Complex Instruction Set Computer) architecture. E.g. Intel x86
- 1-address instruction
  - Stack machine. E.g. JVM

Register Allocation Problem

- Pseudo Assembly is already a 3-address code.
  - Unlimited registers.
  - Unlimited temporary variables.
  - Operation can be applied to both registers and temporary variables
- In reality
  - Limited registers
  - Large set of temporary variables, but still limited
  - Most operation only work on registers
- To convert PA to a 3-address machine code
  - Map temporary variables to machine code temp variables (not too hard)
  - Allocate registers for temporary variables (hard problem)

Register Allocation

- 4 other registers `r0`, `r1`, `r2` and `r3`, besides `rret`.
- mapping temp variables to registers `{x : r0, y : r1, z : r2, w : r3}`

```
// PA1
1: x <- input
2: y <- x + 1
3: z <- y + 1
4: w <- y * z
5: rret <- w
6: ret
```

is translated into

```
1: r0 <- input
2: r1 <- r0 + 1
3: r2 <- r1 + 1
4: r3 <- r1 * r2
5: rret <- r3
6: ret
```

Register Allocation

- 3 other registers `r0`, `r1`, `r2` besides `rret`.
- some registers must be shared by multiple temp variables.
- mapping temp variables to registers
  - instructions 1-4: `{x : r0, y : r1, z : r2}`
  - instructions 5-8: `{w : r0, y : r1, z : r2}`

```
// PA1
1: x <- input
2: y <- x + 1
3: z <- y + 1
4: w <- y * z
5: rret <- w
6: ret
```

is translated into

```
1: r0 <- input
2: r1 <- r0 + 1
3: r2 <- r1 + 1
4: x  <- r0
5: r0 <- r1 * r2
6: rret <- r0
7: ret
```

Register Allocation

- 3 other registers `r0`, `r1`, `r2` besides `rret`.
- some registers must be shared by multiple temp variables.
- mapping temp variables to registers
  - instructions 1-4: `{x : r0, y : r1, z : r2}`
  - instructions 5-8: `{w : r0, y : r1, z : r2}`

```
// PA1
1: x <- input
2: y <- x + 1
3: z <- y + 1
4: w <- y * z
5: rret <- w
6: ret
```

is translated into

```
1: r0 <- input
2: r1 <- r0 + 1
3: r2 <- r1 + 1
4: r0 <- r1 * r2
5: rret <- r0
6: ret
```

since `x` is not live from 4 onwards.

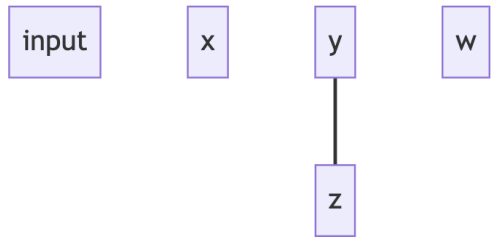
Register Allocation Problem

Given a program p, allocate the variables and assignment results to k number of registers, so that the target code is behaving the same as p and spilling is minimized.

Interference Graph

- A data structure to reason able the constraints of register allocation
- Two temporary variables are *interferring* when they are both "live" at the same time in a program.

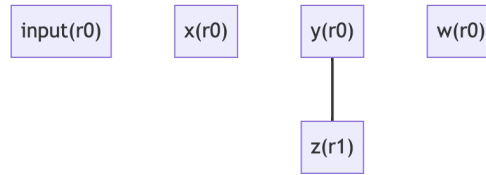
```
// PA1
1: x <- input // {input}
2: y <- x + 1  // {x}
3: z <- y + 1  // {y}
4: w <- y * z  // {y,z}
5: rret <- w   // {w}
6: ret         // {}
```



Interference Graph

- A data structure to reason able the constraints of register allocation
- Two temporary variables are *interferring* when they are both "live" at the same time in a program.

```
// PA1
1: x <- input // {input}
2: y <- x + 1  // {x}
3: z <- y + 1  // {y}
4: w <- y * z  // {y,z}
5: rret <- w   // {w}
6: ret         // {}
```

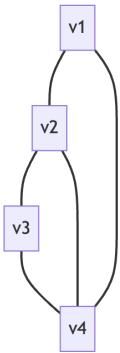


Register Allocation and Graph coloring

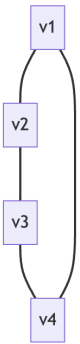
- Register allocation - check whether k registers are enough w/o spilling
- Graph coloring - check whether a graph can be colored with k colors such that the neighbooring vertices having different colors.
  - An NP-complete problem
  - A polynomial algorithm exists for a specific type of graphs.

Chordal Graph

A graph $G = (V, E)$ is *chordal* if, for all cycle $v_1, ..., v_n$ in G with $n > 3$ there exists an edge $(v_i, v_j) \in E$ and $i, j \in \{1, ..., n\}$ such that (v_i, v_j) is not part of the cycle.



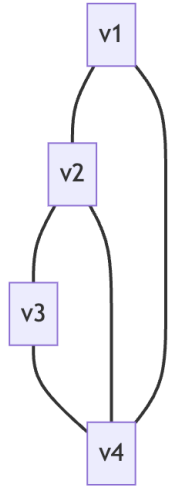
is chordal, because of (v_2, v_4).



is chordless

Coloring chordal graph

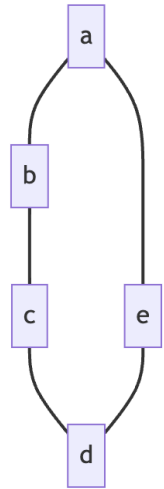
- A clique is a sub graph of which all the vertexes are connected.
- Coloring chordal graph can be done in polynomial time.
  1. initialize a stack s
  2. From a chordal graph g, find a vertex v whose neighors are still in a clique, if we remove v.
  3. remove v from g and let the remaining graph be g', and push v in s.
  4. check whether g' is empty,.
     - 4.1 if g' is not empty, let $g = g'$ and goto 2.
     - 4.2 if g' is empty, color the original graph in the order of popping all the vertexes from q.



An Example

```
// PA2
1: a <- 0              // {}
2: b <- 1              // {a}
3: c <- a + b          // {a, b}
4: d <- b + c          // {b, c}
5: a <- c + d          // {c, d}
6: e <- 2              // {a}
7: d <- a + e          // {a, e}
8: r_ret <- e + d      // {e, d}
9: ret
```

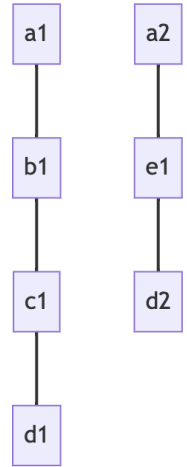
Actually can be allocated with only 2 registers



Not colorable using 2 colors.

SSA programs give chordal interference graph

```
// PA_SSA2
1: a1 <- 0              // {}
2: b1 <- 1              // {a1}
3: c1 <- a1 + b1        // {a1, b1}
4: d1 <- b1 + c1        // {b1, c1}
5: a2 <- c1 + d1        // {c1, d1}
6: e1 <- 2              // {a2}
7: d2 <- a2 + e1        // {a2, e1}
8: r_ret <- e1 + d2     // {e1, d2}
9: ret
```



SSA program register allocation

1. Convert the PA program into a SSA.
2. Perform Liveness Analysis on the SSA.
3. Generate the live range table based on the liveness analysis results.
   - live range table is a better data structure compared to interference graph
4. Allocate registers based on the live range table. Detect potential spilling.
5. Depends on the last approach, either
   - 5.1 convert SSA back to PA and generate the target code according to the live range table, or
   - 5.2 generate the target code from SSA with register coalesced for the phi assignment operands.

SSA program liveness analysis

We define the $join(s_i)$ function as follows

$$join(s_i) = \bigsqcup_{v_j \in succ(v_i)} \Theta_{i,j}(s_j)$$

$$\Theta_{i,j} \quad = \quad \{(t_i/t_k) \mid t_k = phi(..., i : t_i, ...) \in \overline{\phi}\}$$

The monotonic functions can be defined by the following cases.

- case $l : \overline{\phi}\ ret,\ s_l = \{\}$
- case $l : \overline{\phi}\ t \leftarrow src,\ s_l = join(s_l) - \{t\} \cup var(src)$
- case $l : \overline{\phi}\ t \leftarrow src_1\ op\ src_2,\ s_l = join(s_l) - \{t\} \cup var(src_1) \cup var(src_2)$
- case $l : \overline{\phi}\ r \leftarrow src,\ s_l = join(s_l) \cup var(src)$
- case $l : \overline{\phi}\ r \leftarrow src_1\ op\ src_2,\ s_l = join(s_l) \cup var(src_1) \cup var(src_2)$
- case $l : \overline{\phi}\ ifn\ t\ goto\ l',\ s_l = join(s_l) \cup \{t\}$
- other cases: $s_l = join(s_l)$

Live Range Table

```
// PA_SSA2
1: a1 <- 0              // {}
2: b1 <- 1              // {a1}
3: c1 <- a1 + b1        // {a1, b1}
4: d1 <- b1 + c1        // {b1, c1}
5: a2 <- c1 + d1        // {c1, d1}
6: e1 <- 2              // {a2}
7: d2 <- a2 + e1        // {a2, e1}
8: r_ret <- e1 + d2     // {e1, d2}
9: ret
```

- Follow the dominator tree to allocate registers

var	1	2	3	4	5	6	7	8	9	reg
a1		*	*							r0
b1			*	*						r1
c1				*	*					r0
d1					*					r1
a2						*	*			r0
e1							*	*		r1
d2								*		r0

Handle Register Spilling

```
// PA3
1: x <- 1          // {}
2: y <- x + 1      // {x}
3: z <- x * x      // {x,y}
4: w <- y * x      // {x,y,z}
5: u <- z + w      // {z,w}
6: r_ret <- u      // {u}
7: ret             // {}
```

Which variable to spill

- **least urgent**
- most conflict/interferring

Consider we only have 2 registers

var	1	2	3	4	5	6	7	reg
x		*	*	*				r0
y			*	*				r1
z				*	*			??
w					*			
u						*		

Handle Register Spilling

```
// PA3
1: x <- 1          // {}
2: y <- x + 1      // {x(3)}
3: z <- x * x      // {x(3),y(4)}
4: w <- y * x      // {x(4),y(4),z(5)}
5: u <- z + w      // {z(5),w(5)}
6: r_ret <- u      // {u(6)}
7: ret             // {}
```

- at instruction 4, `w` is less urgent compared to `x` and `y`.

Consider we only have 2 registers

var	1	2	3	4	5	6	7	reg
x		*	*	*				r0
y			*	*				r1
z				-	*			r1
w					*			r0
u						*		r1

Handle Register Spilling

```
// PA3
1: x <- 1          // {}
2: y <- x + 1      // {x(3)}
3: z <- x * x      // {x(3),y(4)}
4: w <- y * x      // {x(4),y(4),z(5)}
5: u <- z + w      // {z(5),w(5)}
6: r_ret <- u      // {u(6)}
7: ret             // {}
```

We need to free up some reigsters to spill r0 back to z,

Consider we only have 2 registers

```
// PA3_REG
1: r0 <- 1          // x is r0
2: r1 <- r0 + 1     // y is r1
   y  <- r1         // temp save y
3: r1 <- r0 * r0    // z is r1
   z  <- r1         // spill to z
   r1 <- y          // y is r1
4: r0 <- r1 * r0    // w is r0
                    // x,y are dead
   r1 <- z          // z is r1
5: r1 <- r1 + r0    // u is r1
                    // z,w are dead
6: r_ret <- r1
7: ret
```

Dealing with Phi Assignments

```
// PA4
1: x <- input    // {input}
2: s <- 0        // {x}
3: c <- 0        // {s,x}
4: b <- c < x    // {c,s,x}
5: ifn b goto 9 // {b,c,s,x}
6: s <- c + s    // {c,s,x}
7: c <- c + 1    // {c,s,x}
8: goto 4        // {c,s,x}
9: r_ret <- s    // {s}
10: ret          // {}
```

```
// PA_SSA4
1: x1 <- input1  // {input1(1)}
2: s1 <- 0       // {x1(4)}
3: c1 <- 0       // {s1(4),x1(4)}
4: c2 <- phi(3:c1, 8:c3)
   s2 <- phi(3:s1, 8:s3)
   b1 <- c2 < x1 // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9 // {b1(5),c2(6),s2(6,9),x1(4)}
6: s3 <- c2 + s2 // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1  // {c2(7),s3(4),x1(4)}
8: goto 4        // {c3(4),s3(4),x1(4)}
9: r_ret <- s2   // {s2(9)}
10: ret          // {}
```

Two options

1. convert them to normal assignments
2. registers coalesced for the phi operands

Convert Phi Assignment to Normal Assignments

```
// PA_SSA4
1: x1 <- input1  // {input1(1)}
2: s1 <- 0       // {x1(4)}
3: c1 <- 0       // {s1(4),x1(4)}
4: c2 <- phi(3:c1, 8:c3)
   s2 <- phi(3:s1, 8:s3)
   b1 <- c2 < x1 // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9 // {b1(5),c2(6),s2(6,9),x1(4)}
6: s3 <- c2 + s2 // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1  // {c2(7),s3(4),x1(4)}
8: goto 4        // {c3(4),s3(4),x1(4)}
9: r_ret <- s2   // {s2(9)}
10: ret          // {}
```

```
// PA_SSA_PA4
1: x1 <- input1  // {input1(1)}
2: s1 <- 0       // {x1(4)}
3: c1 <- 0       // {s1(3.1),x1(4)}
3.1: c2 <- c1
     s2 <- s1    // {s1(3.1),x1(4),c1(3.1)}
4: b1 <- c2 < x1 // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9 // {b1(5),c2(6),s2(6,9),x1(4)}
6: s3 <- c2 + s2 // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1  // {c2(7),s3(7.1),x1(4)}
7.1: c2 <- c3
     s2 <- s3    // {s3(7.1),x1(4),c3(7.1)}
8: goto 4        // {c2(4),s2(6,9),x1(4)}
9: r_ret <- s2   // {s2(9)}
10: ret          // {}
```

Convert Phi Assignment to Normal Assignments

```
// PA_SSA_PA4
1: x1 <- input1   // {input1(1)}
2: s1 <- 0        // {x1(4)}
3: c1 <- 0        // {s1(3.1),x1(4)}
3.1: c2 <- c1
     s2 <- s1     // {s1(3.1),x1(4),c1(3.1)}
4: b1 <- c2 < x1  // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9  // {b1(5),c2(6),s2(6,9),x1(4)}
6: s3 <- c2 + s2  // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1   // {c2(7),s3(7.1),x1(4)}
7.1: c2 <- c3
     s2 <- s3     // {s3(7.1),x1(4),c3(7.1)}
8: goto 4         // {c2(4),s2(6,9),x1(4)}
9: r_ret <- s2    // {s2(9)}
10: ret           // {}
```

	1	2	3	.1	4	5	6	7	.1	8	9	reg
in	*											r0
x1		*	*	*	*	-	-	-	-	-		r1
s1			*	*								r2
c1				*								r0
s2					*	*	*			*	*	r2
c2					*	*	*	*		*		r0
b1						*						r1
s3								*	*			r2
c3									*			r0

Convert Phi Assignment to Normal Assignments

```
// PA_SSA_PA4
1: x1 <- input1   // {input1(1)}
2: s1 <- 0        // {x1(4)}
3: c1 <- 0        // {s1(3.1),x1(4)}
3.1: c2 <- c1
     s2 <- s1     // {s1(3.1),x1(4),c1(3.1)}
4: b1 <- c2 < x1  // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9  // {b1(5),c2(6),s2(6,9),x1(4)}
6: s3 <- c2 + s2  // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1   // {c2(7),s3(7.1),x1(4)}
7.1: c2 <- c3
     s2 <- s3     // {s3(7.1),x1(4),c3(7.1)}
8: goto 4         // {c2(4),s2(6,9),x1(4)}
9: r_ret <- s2    // {s2(9)}
10: ret           // {}
```

```
// PA4_REG1
1: r0 <- input1   // input is r0
   r1 <- r0       // x1 is r1
2: r2 <- 0        // s1 is r2
3: r0 <- 0        // c1 is r0
                  // c2 is r0
                  // s2 is r2
                  // no need to load r1 from x1
                  // b/c x1 is still active in r1
                  // from 3 to 4
4: x1 <- r1       // spill r1 to x1
   r1 <- r0 < r1  // b1 is r1
5: ifn r1 goto 9  //
6: r2 <- r0 + r2  // s3 is r2
7: r0 <- r0 + 1   // c3 is r0
                  // c2 is r0
                  // s2 is r2
8: r1 <- x1       // restore r1 from x1
   goto 4         // x1 is inactive but needed in 4
9: r_ret <- r2    //
10: ret           //
```

What if at instruction 7, we allocate r1 to s3 instead of r2?

Registers coalesced for the phi operands

```
// PA_SSA4
1: x1 <- input1   // {input1(1)}
2: s1 <- 0        // {x1(4)}
3: c1 <- 0        // {s1(4),x1(4)}
4: c2 <- phi(3:c1, 8:c3)
   s2 <- phi(3:s1, 8:s3)
   b1 <- c2 < x1 // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9 // {b1(5),c2(6),s2(6),x1(4)}
6: s3 <- c2 + s2 // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1  // {c2(7),s3(4),x1(4)}
8: goto 4        // {c3(4),s3(4),x1(4)}
9: r_ret <- s2   // {s2(9)}
10: ret          // {}
```

- c1 and c3 share the same register
- s1 and s3 share the same register

var	1	2	3	4	5	6	7	8	9	reg
in	*									
x1		*	*	*	-	-	-	-		r1
s1			*							r2
c1										r0
s2				*	*	*			*	r2
c2				*	*	*	*			r0
b1					*					r1
s3							*	*		r2
c3								*		r0

Registers coalesced for the phi operands

```
// PA_SSA4
1: x1 <- input1  // {input1(1)}
2: s1 <- 0       // {x1(4)}
3: c1 <- 0       // {s1(4),x1(4)}
4: c2 <- phi(3:c1, 8:c3)
   s2 <- phi(3:s1, 8:s3)
   b1 <- c2 < x1 // {c2(4),s2(6,9),x1(4)}
5: ifn b1 goto 9 // {b1(5),c2(6),s2(6),x1(4)}
6: s3 <- c2 + s2 // {c2(6),s2(6),x1(4)}
7: c3 <- c2 + 1  // {c2(7),s3(4),x1(4)}
8: goto 4        // {c3(4),s3(4),x1(4)}
9: r_ret <- s2   // {s2(9)}
10: ret          // {}
```

- c1 and c3 share the same register
- s1 and s3 share the same register
- heuristic, not always produce more efficient code than the conservative approach

```
// PA4_REG3 same as PA4_REG1
1: r0 <- input1  // input is r0
   r1 <- r0      // x1 is r1
2: r2 <- 0       // s1 is r2
3: r0 <- 0       // c1 is r0
                 // c2 is r0
                 // s2 is r2
                 // no need to load r1 from x1
                 // b/c x1 is still active in r1
                 // from 3 to 4
4: x1 <- r1      // spill r1 to x1
   r1 <- r0 < r1 // b1 is r1
5: ifn r1 goto 9 //
6: r2 <- r0 + r2 // s3 is r2
7: r0 <- r0 + 1  // c3 is r0
                 // c2 is r0
                 // s2 is r2
8: r1 <- x1      // restore r1 from x1
   goto 4        // x1 is inactive but needed in 4
9: r_ret <- r2   //
10: ret          //
```

Summary

1. Convert the PA program into a SSA.
2. Perform Liveness Analysis on the SSA.
3. Generate the live range table based on the liveness analysis results.
4. Allocate registers based on the live range table. Detect potential spilling.
5. Depending on the last approach, either
   - 5.1 convert SSA back to PA and generate the target code according to the live range table, or
   - 5.2 generate the target code from SSA with register coalesced for the phi assignment operands.