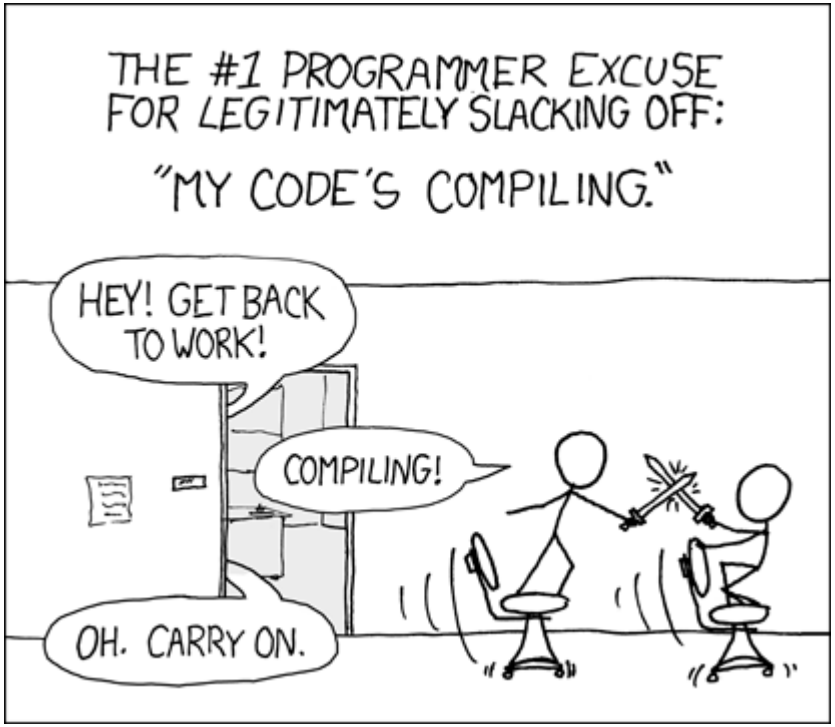
Technische Universiteit
Eindhoven
University of Technology

The life of a function

Roel Jordans



Goal

- See how a simple function passes through the backend
- Our function:

```
int test(int a, int b) {
    return 2*a + b + 3;
}
```

Starting point

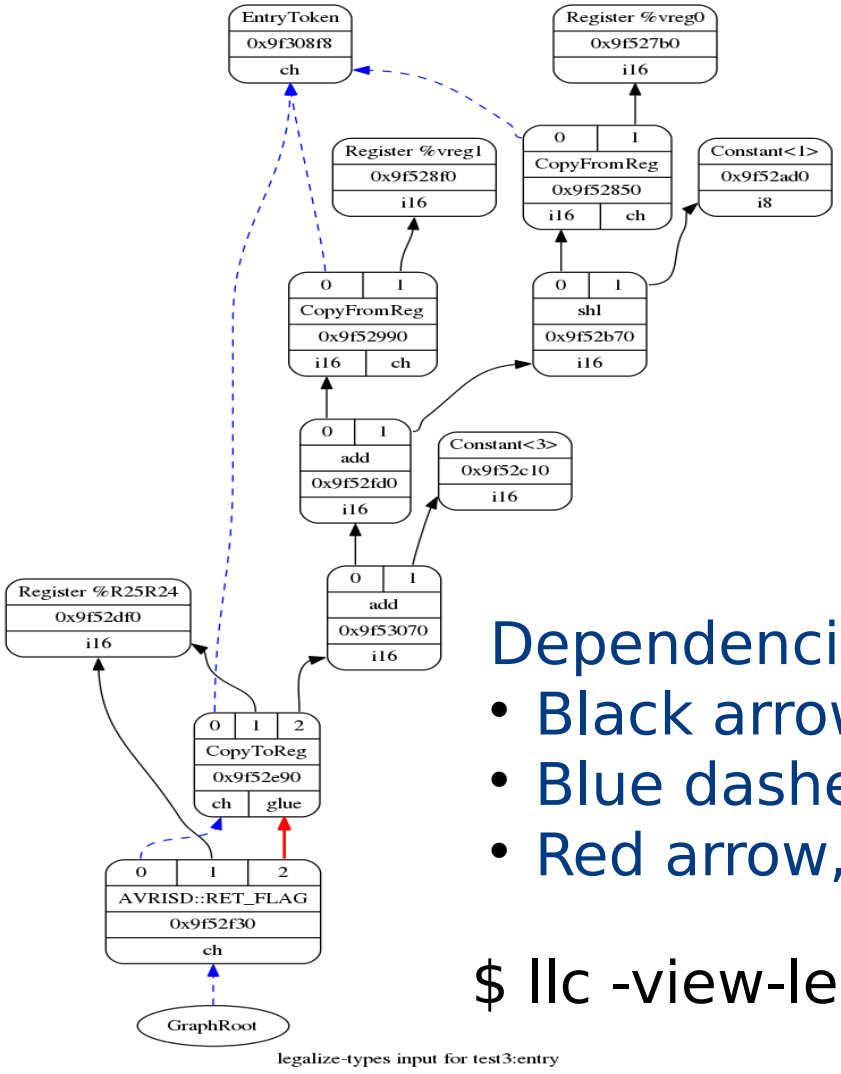
- After the frontend (+optimizations)

```
define i16 @test3(i16 %a, i16 %b) {
entry:
  %mul = shl i16 %a, 1
  %add = add i16 %b, 3
  %add1 = add i16 %add, %mul
  ret i16 %add1
}
```

Multplication became shift!

int test(int a, int b) { return 2*a + b + 3; }

Or as a graph



Dependencies legend:

- Black arrow, data
- Blue dashed arrow, chaining
- Red arrow, glue

$ llc -view-legalize-dags test.ll

int test(int a, int b) { return 2*a + b + 3; }

So, these added nodes

- Represent the function call/return
- First things to be added in lowering
  - Copy inputs from call into virtual registers
  - Put output of function in virtual register
  - Replace the return node

int test(int a, int b) { return 2*a + b + 3; }

Want more helpful graphs?

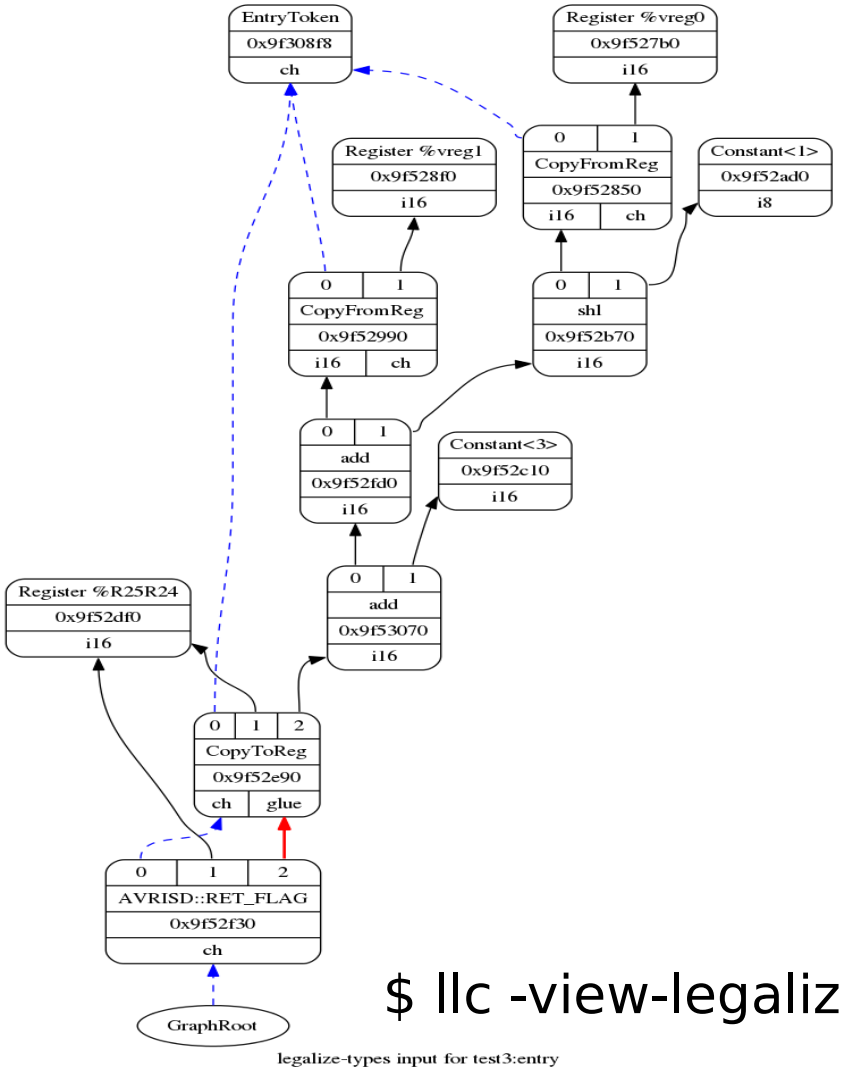
- llc can show more nice debug output:

```
-view-legalize-types-dags
-view-dag-combine1-dags
-view-legalize-dags
-view-dag-combine2-dags
-view-isel-dags
```

- Graphs are shown *before* the step

int test(int a, int b) { return 2*a + b + 3; }

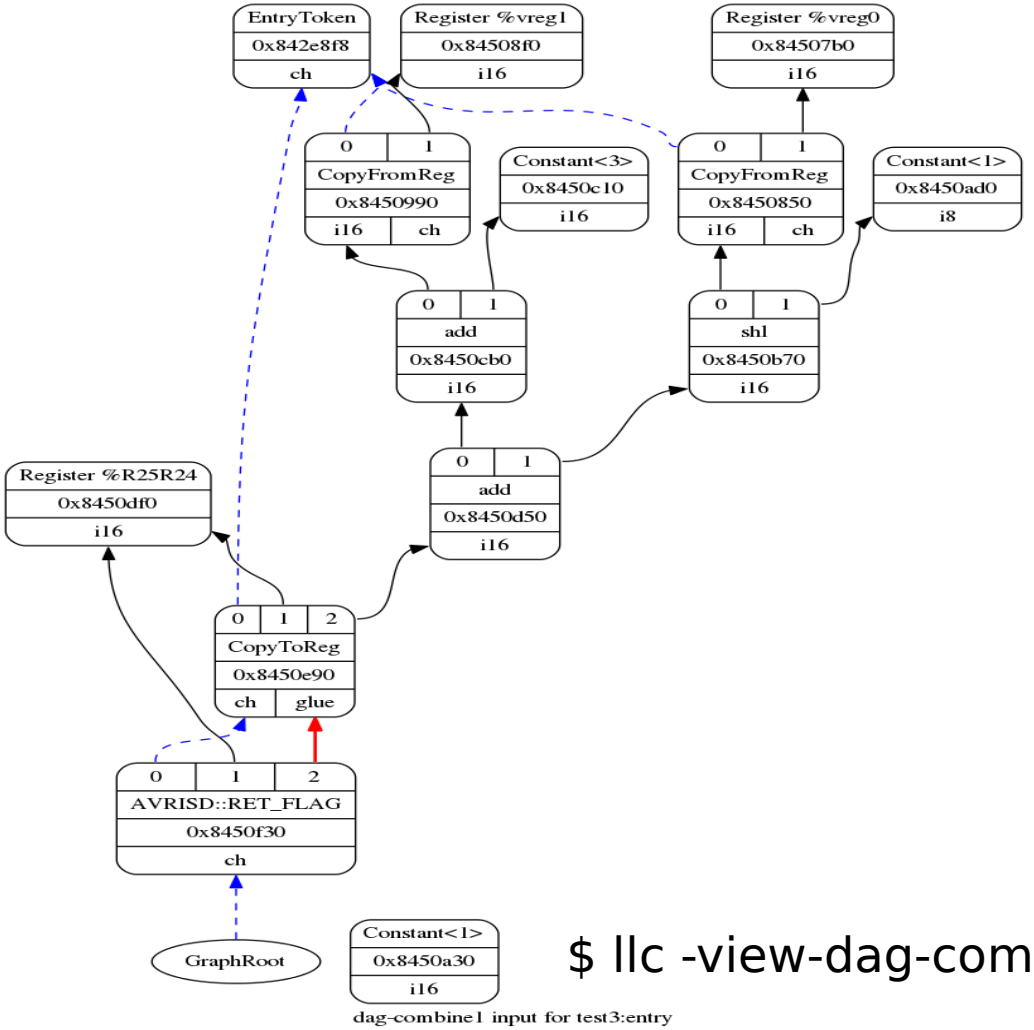
Before legalize-types



$ llc -view-legalize-types-dags test.ll

int test(int a, int b) { return 2*a + b + 3; }

Before dag-combine1

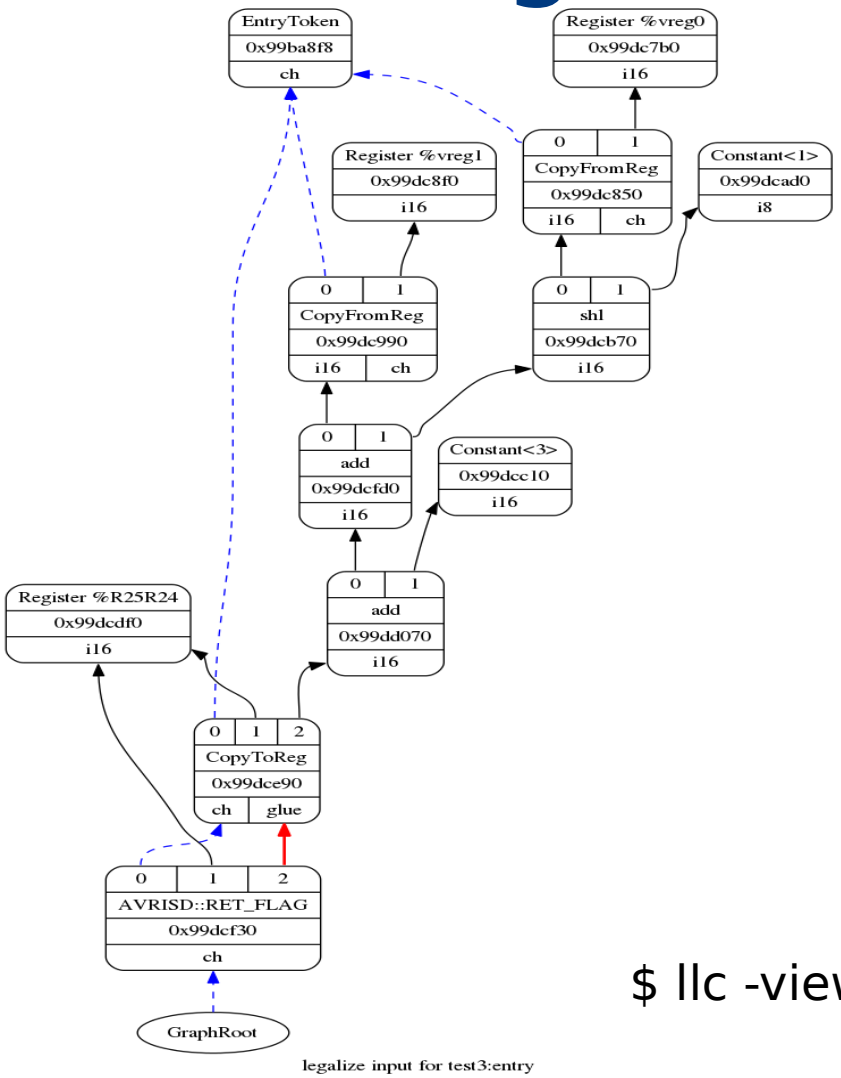


dag-combine1 input for test3:entry

$ llc -view-dag-combine1-dags test.ll

int test(int a, int b) { return 2*a + b + 3; }

Before legalize

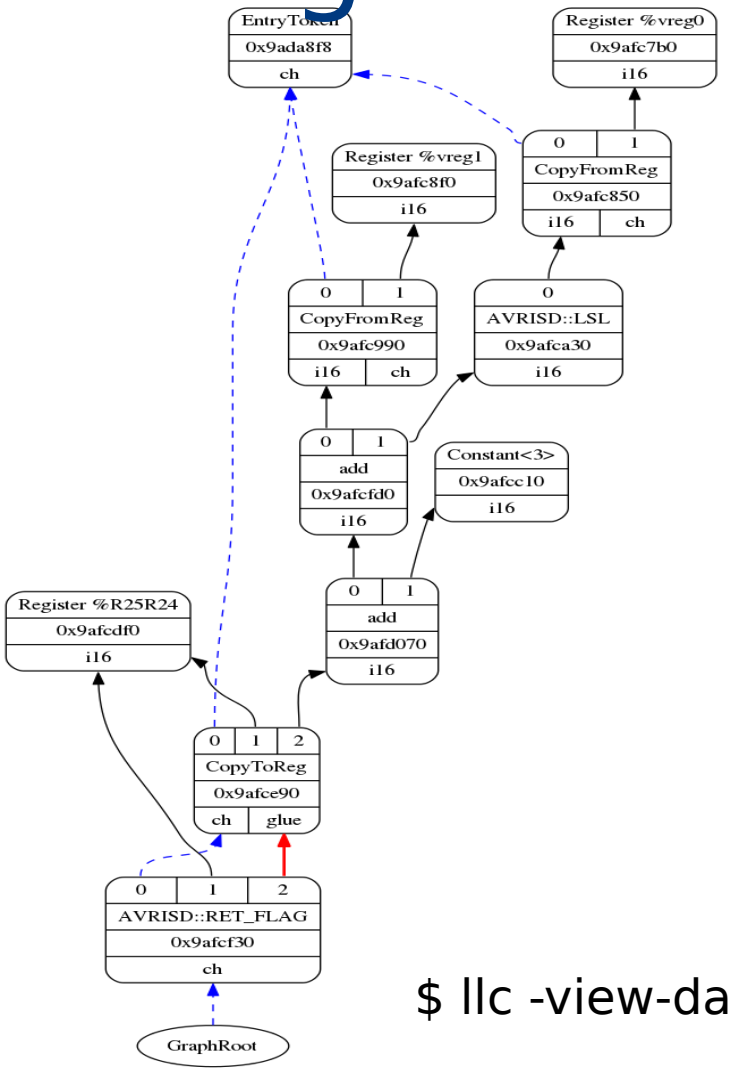


legalize input for test3:entry

$ llc -view-legalize-dags test.ll

int test(int a, int b) { return 2*a + b + 3; }

Before dag-combine2



$ llc -view-dag-combine2-dags test.ll

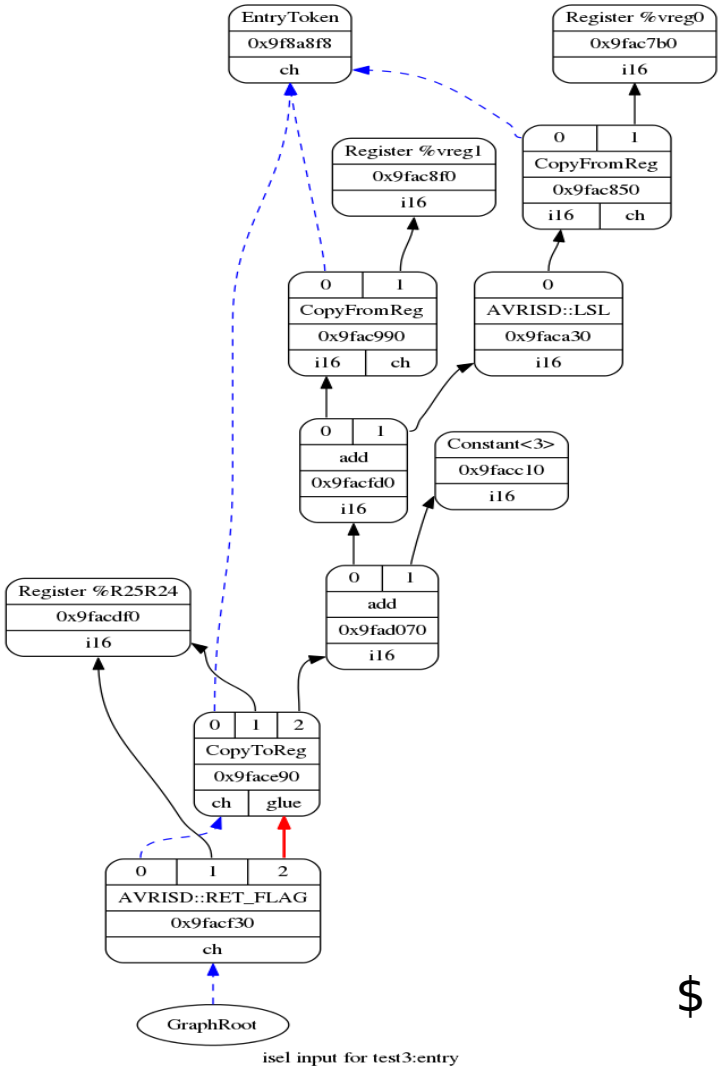
int test(int a, int b) { return 2*a + b + 3; }

Wait, something changed!

- The shl operation got lowered to AVRISD::LSL
- AVR only allows 1-bit shifts
  - AVRISelLowering.cpp has code to check if it should either do
    - LSL (in case it really is a 1-bit shift)
    - A sequence of LSL's (n-bit shift)
    - LSLLOOP (variable shift amounts)

int test(int a, int b) { return 2*a + b + 3; }

Before isel



$ llc -view-isel-dags test.ll

int test(int a, int b) { return 2*a + b + 3; }

Instruction-selection

- So, no more nice graphs from here :'(
- But we have other debug output
  - llc -debug test.ll
  - llc -debug-only=avr-isel test.ll
- Produces a lot of text, pipe through less
  - llc -debug-only=isel test.ll |& less

Instruction-selection input

- Same point as the previous graph!

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16 = add t4, t16
    t15: i16 = add t14, Constant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = AVRISD::RET_FLAG t12, Register:i16 %R25R24, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16 = add t4, t16
    t15: i16 = add t14, Constant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = AVRISD::RET_FLAG t12, Register:i16 %R25R24, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16 = add t4, t16
    t15: i16 = add t14, Constant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16 = add t4, t16
    t15: i16 = add t14, Constant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16 = add t4, t16
    t15: i16,i8 = ADIWRdK t14, TargetConstant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16 = AVRISD::LSL t2
      t14: i16,i8 = ADDWRdRr t4, t16
    t15: i16,i8 = ADIWRdK t14, TargetConstant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16,i8 = LSLWRd t2
      t14: i16,i8 = ADDWRdRr t4, t16
    t15: i16,i8 = ADIWRdK t14, TargetConstant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection steps

- Bottom-up greedy selection
  - Match as many nodes as possible
  - Nodes selected only once

```
Optimized legalized selection DAG: BB#0 'test3:entry'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
        t4: i16,ch = CopyFromReg t0, Register:i16 %vreg1
          t2: i16,ch = CopyFromReg t0, Register:i16 %vreg0
        t16: i16,i8 = LSLWRd t2
      t14: i16,i8 = ADDWRdRr t4, t16
    t15: i16,i8 = ADIWRdK t14, TargetConstant:i16<3>
  t12: ch,glue = CopyToReg t0, Register:i16 %R25R24, t15
  t13: ch = RET Register:i16 %R25R24, t12, t12:1
```

Instruction-selection finish

- Did we really finish now?
  - Need to copy the input values from the correct registers
  - All instructions are matched into low-level operations
  - But some are *pseudo* operations
    - AVR doesn't really have LSLWRd or ADDWRdRr operations
      - Only a single byte versions are available
    - We'll ignore those for now and leave them in the code...

Scheduling

- Put the operations into execution order
- Changes from graph/tree notation into a sequence
- Still SSA
  - Defines and uses virtual registers
    - Virtual registers now define *live-intervals* for operation results
    - Virtual registers need to match the *register class* allowed for the operands/results of operations

Scheduling result

```
*** MachineFunction at end of ISel ***
# Machine code for function test3: SSA
Function Live Ins: %R25R24 in %vreg0, %R23R22 in %vreg1

BB#0: derived from LLVM BB %entry
    Live Ins: %R25R24 %R23R22
        %vreg1<def> = COPY %R23R22; DREGS:%vreg1
        %vreg0<def> = COPY %R25R24; DREGS:%vreg0
        %vreg2<def,tied1> = LSLWRd %vreg0<tied0>,
                            %SREG<imp-def,dead>
                            ; DREGS:%vreg2,%vreg0
        %vreg3<def,tied1> = ADDWRdRr %vreg1<tied0>, %vreg2<kill>,
                            %SREG<imp-def,dead>
                            ; IWREGS:%vreg3 DREGS:%vreg1,%vreg2
        %vreg4<def,tied1> = ADIWRdK %vreg3<tied0>, 3,
                            %SREG<imp-def,dead>
                            ; IWREGS:%vreg4,%vreg3
        %R25R24<def> = COPY %vreg4; IWREGS:%vreg4
        RET %R25R24<imp-use>
```

Scheduling algorithms

- Many ways:
  - List scheduling
  - Modulo scheduling
  - …
- On regions of different sizes:
  - Basic block
  - Loop
  - …
- More on this in the next lecture

Register allocation

- Change virtual registers into real ones
- Start with scheduled operations
  - Refine the live intervals
  - Assign a register to each interval
- For now: Assume that there are enough registers available

Live intervals: leaving SSA

```
BB#0: derived from LLVM BB %entry
 Live Ins: %R25R24 %R23R22
1  %vreg1<def> = COPY %R23R22<kill>; DREGS:%vreg1
2  %vreg0<def> = COPY %R25R24<kill>; DREGS:%vreg0
3  %vreg2<def> = COPY %vreg0<kill>; DREGS:%vreg2,%vreg0
4  %vreg2<def,tied1> = LSLWRd %vreg2<tied0>, %SREG<imp-def,dead>
5    ; DREGS:%vreg2
6  %vreg3<def> = COPY %vreg2<kill>; IWREGS:%vreg3 DREGS:%vreg2
7  %vreg3<def,tied1> = ADDWRdRr %vreg3<tied0>, %vreg1<kill>,
8    %SREG<imp-def,dead>; IWREGS:%vreg3 DREGS:%vreg1
9  %vreg4<def> = COPY %vreg3<kill>; IWREGS:%vreg4,%vreg3
10 %vreg4<def,tied1> = ADIWRdK %vreg4<tied0>, 3,
11   %SREG<imp-def,dead>; IWREGS:%vreg4
12 %R25R24<def> = COPY %vreg4<kill>; IWREGS:%vreg4
13 RET %R25R24<imp-use,kill>
```

Register coalescing: before

```
BB#0: derived from LLVM BB %entry
 Live Ins: %R25R24 %R23R22
16B   %vreg1<def> = COPY %R23R22<kill>; DREGS:%vreg1
32B   %vreg0<def> = COPY %R25R24<kill>; DREGS:%vreg0
48B   %vreg2<def> = COPY %vreg0<kill>; DREGS:%vreg2,%vreg0
64B   %vreg2<def,tied1> = LSLWRd %vreg2<tied0>, %SREG<imp-def,dead>
        ; DREGS:%vreg2
80B   %vreg3<def> = COPY %vreg2<kill>; IWREGS:%vreg3 DREGS:%vreg2
96B   %vreg3<def,tied1> = ADDWRdRr %vreg3<tied0>, %vreg1<kill>,
        %SREG<imp-def,dead>; IWREGS:%vreg3 DREGS:%vreg1
112B  %vreg4<def> = COPY %vreg3<kill>; IWREGS:%vreg4,%vreg3
128B  %vreg4<def,tied1> = ADIWRdK %vreg4<tied0>, 3,
        %SREG<imp-def,dead>; IWREGS:%vreg4
144B  %R25R24<def> = COPY %vreg4<kill>; IWREGS:%vreg4
160B  RET %R25R24<imp-use,kill>
```

```
********** INTERVALS **********
R22 [0B,16r:0)  0@0B-phi
R23 [0B,16r:0)  0@0B-phi
R24 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
R25 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
%vreg0 [32r,48r:0)  0@32r
%vreg1 [16r,96r:0)  0@16r
%vreg2 [48r,64r:0)[64r,80r:1)  0@48r 1@64r
%vreg3 [80r,96r:0)[96r,112r:1)  0@80r 1@96r
%vreg4 [112r,128r:0)[128r,144r:1)  0@112r 1@128r
```

Register coalescing: after

```
# Machine code for function test3: Post SSA
Function Live Ins: %R25R24 in %vreg0, %R23R22 in %vreg1

0B BB#0: derived from LLVM BB %entry
      Live Ins: %R25R24 %R23R22
16B  %vreg1<def> = COPY %R23R22; DREGS:%vreg1
32B  %vreg3<def> = COPY %R25R24; IWREGS:%vreg3
64B  %vreg3<def,tied1> = LSLWRd %vreg3<tied0>, %SREG<imp-def,dead>
       ; IWREGS:%vreg3
96B  %vreg3<def,tied1> = ADDWRdRr %vreg3<tied0>, %vreg1,
       %SREG<imp-def,dead>; IWREGS:%vreg3 DREGS:%vreg1
128B %vreg3<def,tied1> = ADIWRdK %vreg3<tied0>, 3, %SREG<imp-def,dead>
       ; IWREGS:%vreg3
144B %R25R24<def> = COPY %vreg3; IWREGS:%vreg3
160B RET %R25R24<imp-use>
```

```
********** INTERVALS **********
R22 [0B,16r:0)  0@0B-phi
R23 [0B,16r:0)  0@0B-phi
R24 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
R25 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
%vreg1 [16r,96r:0)  0@16r
%vreg3 [32r,64r:2)[64r,96r:0)
       [96r,128r:1)[128r,144r:3)
          0@64r 1@96r 2@32r 3@128r
```

Register allocation

```
0B BB#0: derived from LLVM BB %entry
      Live Ins: %R25R24 %R23R22
16B  %vreg1<def> = COPY %R23R22; DREGS:%vreg1
32B  %vreg3<def> = COPY %R25R24; IWREGS:%vreg3
64B  %vreg3<def,tied1> = LSLWRd %vreg3<tied0>, %SREG<imp-def,dead>
       ; IWREGS:%vreg3
96B  %vreg3<def,tied1> = ADDWRdRr %vreg3<tied0>, %vreg1,
       %SREG<imp-def,dead>; IWREGS:%vreg3 DREGS:%vreg1
128B %vreg3<def,tied1> = ADIWRdK %vreg3<tied0>, 3, %SREG<imp-def,dead>
       ; IWREGS:%vreg3
144B %R25R24<def> = COPY %vreg3; IWREGS:%vreg3
160B RET %R25R24<imp-use>
```

```
********** INTERVALS **********
R22 [0B,16r:0)  0@0B-phi
R23 [0B,16r:0)  0@0B-phi
R24 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
R25 [0B,32r:0)[144r,160r:1)  0@0B-phi 1@144r
%vreg1 [16r,96r:0)  0@16r
%vreg3 [32r,64r:2)[64r,96r:0)
       [96r,128r:1)[128r,144r:3)
         0@64r 1@96r 2@32r 3@128r
```

Greedy allocation:

Step 1: Allocate %vreg1, suggestion R23R22
Step 2: Allocate %vreg3, suggestion R25R24

Register allocation: result

```
> %R23R22<def> = COPY %R23R22
Deleting identity copy.
> %R25R24<def> = COPY %R25R24
Deleting identity copy.
> %R25R24<def,tied1> = LSLWRd %R25R24<kill,tied0>, %SREG<imp-def,dead>
> %R25R24<def,tied1> = ADDWRdRr %R25R24<kill,tied0>, %R23R22<kill>,
                                                      %SREG<imp-def,dead>
> %R25R24<def,tied1> = ADIWRdK %R25R24<kill,tied0>, 3,
                                                      %SREG<imp-def,dead>
> %R25R24<def> = COPY %R25R24<kill>
Deleting identity copy.
> RET %R25R24<imp-use>
```

Pseudo operation expansion

- We're almost there
- But still have our pseudo operations
  - LSLWRd
  - ADDWRdRr
- We can't emit those... Lets expand them now so we can finish
- Custom MachineFunctionPass
  - Implemented in AVRExpandPseudoInsts.cpp

The result

```
    .text
    .file   "test3.c"
    .globl  test3
    .align  2
    .type   test3,@function
test3:                                  ; @test3
; BB#0:                                 ; %entry
    lsl r24
    rol r25
    add r24, r22
    adc r25, r23
    adiw    r24, 3
    ret
.Lfunc_end0:
    .size   test3, .Lfunc_end0-test3
```

Technische Universiteit
Eindhoven
University of Technology