Register Allocation

Eva Rose  Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-10.pdf

November 13, 2014
1. Basic Block Code Generation
2. Basic Block DAG
3. Cross-BB Register Allocation
4. Interference Graphs
5. HACS & Project Milestone 2 Part B
Sixth compilation phase

source program

Tokens

Lexical Analysis

Syntax Analysis

Symbol Table

Syntax Analysis

Semantic Analysis

Intermediate Representation Generator

Intermediate Representation Generator

Optimizer

Intermediate Representation Generator

Optimizer

Code Generator

Intermediate Representation Generator

Machine-Dependent Code Optimizer

Assembly

target machine code
1. Basic Block Code Generation
2. Basic Block DAG
3. Cross-BB Register Allocation
4. Interference Graphs
5. HACS & Project Milestone 2 Part B
Next-Use

How many registers are needed inside a basic block?
Idea

Now generate the instructions but... 

- Maintain mapping between variables and registers.
- Avoid loading values already in a register.
- Store only as needed.
- Reuse registers that have no further use.
Let’s translate

\[
\begin{align*}
t &= a - b \\
u &= a - c \\
v &= t + u \\
a &= d \\
d &= v + u
\end{align*}
\]

Reconstruct Dragon Book Figure 8.16 from this.
Let’s translate

t = a - b
u = a - c
v = t + u
a = d
d = v + u

Reconstruct Dragon Book Figure 8.16 from this.
Register Allocation: *getReg*

We have value $V$ how do we get it in a register?

1. If we got it all is already well!
2. If we have an empty register then use that.
3. If there is no free register we have to make one—
   1. If we have a redundant one, use that;
   2. If we know our instruction will destroy a register, use it;
   3. If we have no next-use then it is as free;
   4. Otherwise we have to spill.
<table>
<thead>
<tr>
<th></th>
<th>Basic Block Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Basic Block DAG</td>
</tr>
<tr>
<td>3</td>
<td>Cross-BB Register Allocation</td>
</tr>
<tr>
<td>4</td>
<td>Interference Graphs</td>
</tr>
<tr>
<td>5</td>
<td>HACS &amp; Project Milestone 2 Part B</td>
</tr>
</tbody>
</table>
DAG – Directed Acyclic Graph

- One node per initial value.
- One node per statement, with an edge to last node observing every parameter.
- Node labeled by operator and list of variables not used further.
- Output nodes are those with live exit variables.
DAG uses

- Local common subexpressions.
- Dead code elimination.
- Apply algebraic simplifications.
- Reorder statements to reduce variable count.
- Register Allocation.
Local Common Subexpressions

1. \( a = b + c \)
2. \( b = a - d \)
3. \( c = b + c \)
4. \( d = a - d \)
Local Common Subexpressions

1. \( a = b + c \)
2. \( b = a - d \)
3. \( c = b + c \)
4. \( d = a - d \)
Dead Code Elimination

Inputs: \( b, c, d \)

1. \( a = b + c \)
2. \( b = b - c \)
3. \( c = c + d \)
4. \( e = b + c \)

Outputs: \( a, b \)
Dead Code Elimination

Inputs: b, c, d

1. a = b + c
2. b = b - c
3. c = c + d
4. e = b + c

Outputs: a, b
Algebraic Identities

\[ x + 0 = 0 + x = x \]
\[ x \times 1 = 1 \times x = x \]
\[ x - 0 = x \]
\[ x/1 = x \]

**EXPENSIVE**  **CHEAPER**

\[ x^2 = x \times x \]
\[ 2 \times x = x + x \]
\[ x/2 = x \times 0.5 \quad (= x \gg 1) \]
Algebraic Identities

\[ x + 0 = 0 + x = x \]
\[ x - 0 = x \]
\[ x \times 1 = 1 \times x = x \]
\[ x / 1 = x \]

**EXPENSIVE** **CHEAPER**

\[ x^2 = x \times x \]
\[ 2 \times x = x + x \]
\[ x / 2 = x \times 0.5 \quad (= x \gg 1) \]
Algebraic Identities

- **Constant folding**
- **Commutativity** with Local Common Subexpressions
- **Associativity** with composite expressions

\[
\begin{align*}
1. & \quad a = c + b \\
2. & \quad e = c + d + b
\end{align*}
\]
Algebraic Identities

- Constant folding
- Commutativity with Local Common Subexpressions
- Associativity with composite expressions

1. \( a = c + b \)
2. \( e = c + d + b \)
Algebraic Identities

- **Constant folding**
- **Commutativity** with Local Common Subexpressions
- **Associativity** with composite expressions

1. \( a = c + b \)
2. \( e = c + d + b \)
Array References

1. $x = a[i]$
2. $a[j] = y$
3. $z = a[i]$
Array References

1. \( x = a[i] \)
2. \( a[j] = y \)
3. \( z = a[i] \)

Unsafe!
Basic Block Code Generation

Basic Block DAG

Cross-BB Register Allocation

Interference Graphs

HACS & Project Milestone 2 Part B

No Pointing!

1 \[ x = *p \]
2 \[ *q = y \]

The entire memory is a single array! Needs pointer ("points-to") analysis!
No Pointing!

1. \( x = *p \)
2. \( *q = y \)

The entire memory is a single array! Needs pointer ("points-to") analysis!
Respect ordering, pay special attention to overlapping side effects!
Back to Code

Respect ordering, pay special attention to overlapping side effects!

\[
\begin{align*}
1 & \quad a = b + c \\
2 & \quad b = a - d \\
3 & \quad c = b + c \\
4 & \quad d = b
\end{align*}
\]
Back to Code

Respect ordering, pay special attention to overlapping side effects!
1. Basic Block Code Generation
2. Basic Block DAG
3. Cross-BB Register Allocation
4. Interference Graphs
5. HACS & Project Milestone 2 Part B
Example

```c
int mult(int a, int b) {
    int i = a;
    int r = 0;
    while (i > 0) {
        r += b;
        i--;
    }
    return r;
}
```
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test

    Next:
    r = r + b
    i = i - 1

    Test:
    if i>0 goto Next
    return r;
}
ARM32 Instruction Subset

**MOV reg, arg**
\[ \text{reg} = \text{R0} \mid \text{R1} \mid \ldots \mid \text{R15} \mid \text{SP} \mid \text{LR} \mid \text{PC} \]

**ADD reg, reg1, arg2**
\[ \text{arg} = \#\text{imm8} \mid \text{reg} \mid \text{reg},\text{LSL} \#\text{imm5} \mid \text{reg},\text{LSR} \#\text{imm5} \]

**SUB reg, reg1, arg2**

**MUL reg, reg1, arg2**
\[ \text{immn} = \text{n-bit unsigned constant} \]

**AND reg, reg1, arg2**

**ORR reg, reg1, arg2**

**EOR reg, reg1, arg2**

**CMP reg, arg**

**B imm12**

**Bcd imm12**
\[ \text{cd} = \text{EQ} \mid \text{NE} \mid \text{GT} \mid \text{LT} \mid \text{GE} \mid \text{LE} \mid \text{CS} \mid \text{CC} \]

**BL imm12**

**LDRb reg, mem**
\[ \text{mem} = [\text{reg}, \text{arg}] \quad \text{b = B?} \]

**STRb reg, mem**

**LDMFD reg!, mreg**
\[ \text{mreg} = \{\text{reg}, \ldots, \text{reg}\} \]

**STMFD reg!, mreg**
## ARM32 Calling Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>On Entry</th>
<th>On Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0–3</td>
<td>parameter or unused</td>
<td>return value or unused</td>
</tr>
<tr>
<td>r4–11</td>
<td>preserved</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r12</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>r13 ‘sp’</td>
<td>stack pointer</td>
<td>same as on entry</td>
</tr>
<tr>
<td>r14 ‘lr’</td>
<td>return address</td>
<td>(unconstrained)</td>
</tr>
<tr>
<td>r15 ‘pc’</td>
<td>program counter</td>
<td>return address</td>
</tr>
</tbody>
</table>
Example (ARM32 Subset)

```
mult: ; "a"=r0, "b"=r1
   MOV r2, r0 ; "i"=r2
   MOV r3, #0 ; "r"=r3
   B Test

Next: ADD r3, r3, r1
    SUB r2, r2, #1

Test: CMP r2, #0
   BGT Next

   MOV r0, r3
   MOV pc, lr
```
Register Allocation: \textit{getReg} for One Basic Block

We have value $V$ how do we get it in a register?

1. If we got it all is already well!
2. If we have an empty register then use that.
3. If there is no free register we have to make one—
   1. If we have a redundant one, use that;
   2. If we know our instruction will destroy a register, use it;
   3. If we have no next-use then it is as free;
   4. Otherwise we have to \textit{spill}.
Register Allocation for Multiple Basic Blocks?

Determine which variables are required for each jump...
Example (ARM32 Subset, naïve forward register allocation)

# r0  r1  r2  r3  a  b  r  i

MOV r3,r0  # a, i  b  a, i  r0, r3  r1  r0, r3
MOV r2,#0  # a, i  b  r  a, i  r0, r3  r1  r2  r0, r3
B Test     # a, i  b  r  a, i  r0, r3  r1  r2  r0, r3

Next: ADD  r2, r2, r1
SUB  r3, r3, #1

Test: CMP  r3, #0
BGT Next

MOV r0, r2  # r  b  r  i  r0  r1  r2, r0  r3

Eva Rose, Kristoffer Rose
Example (ARM32 Subset, naïve forward register allocation)

```
# r0 r1 r2 r3 a b r i
mult:  STMFD sp!,{r4-r11,lr} # a b r0 r1
      MOV r3,r0 # a,i b a,i r0,r3 r1 r0,r3
      MOV r2,#0 # a,i b r a,i r0,r3 r1 r2 r0,r3
      B Test # a,i b r a,i r0,r3 r1 r2 r0,r3
Next:  ADD r2,r2,r1
      SUB r3,r3,#1
Test:  CMP r3,#0
      BGT Next
      MOV r0,r2 # r b r i r0 r1 r2,r0 r3
```
Example (ARM32 Subset, naïve forward register allocation)

```assembly
# r0  r1  r2  r3  a   b   r   i

mult:   STMFD sp!,{r4-r11,lr}  # a   b
        MOV       r3,r0  # a,i b   a,i
        MOV       r2,#0  # a,i b   r   a,i
        B         Test  # a,i b   r   a,i

Next:  ADD       r2,r2,r1
        SUB       r3,r3,#1

Test:   CMP       r3,#0
        BGT       Next

MOV       r0,r2  # r   b   r   i
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0 r1 r2 r3 a b r i

mult: STMFD sp!, {r4-r11, lr} # a b r0 r1

MOV r3, r0 # a, i b a, i r0, r3 r1 r0, r3
MOV r2, #0 # a, i b r a, i r0, r3 r1 r2 r0, r3
B Test # a, i b r a, i r0, r3 r1 r2 r0, r3

Next: ADD r2, r2, r1

SUB r3, r3, #1

Test: CMP r3, #0

BGT Next # a, i b r a, i r0, r3 r1 r2 r0, r3

MOV r0, r2 # r b r i r0 r1 r2, r0 r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0 r1 r2 r3 a b r i

mult: STMFD sp!,{r4-r11,lr} # a b r0 r1

MOV r3,r0 # a,ib a,i r0,r3 r1 r0,r3
MOV r2,#0 # a,ib r a,i r0,r3 r1 r2 r0,r3
B Test # a,ib r a,i r0,r3 r1 r2 r0,r3

Next: ADD r2,r2,r1
SUB r3,r3,#1

Test: CMP r3,#0
BGT Next

MOV r0,r2 # r b r i r0 r1 r2 r0 r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
 mult:   STMFD sp!,{r4-r11,lr} # a b r0 r1
         MOV  r3, r0        # a,i b a,i r0,r3 r1 r0,r3
         MOV  r2, #0        # a,i b r a,i r0,r3 r1 r2 r0,r3
         B Test          # a,i b r a,i r0,r3 r1 r2 r0,r3

 Next:  ADD  r2, r2, r1
         SUB  r3, r3, #1

 Test:  CMP  r3, #0    # a,i b r a,i r0,r3 r1 r2 r0,r3
         BGT Next
```

```
 Next:  ADD  r2, r2, r1
         SUB  r3, r3, #1

 Test:  CMP  r3, #0    # a,i b r a,i r0,r3 r1 r2 r0,r3
         BGT Next
```

```
         MOV  r0, r2        # r b r i r0 r1 r2, r0 r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0 r1 r2 r3 a b r i

mult: STMFD sp!, {r4-r11, lr} # a b r0 r1

MOV r3, r0 # a, i b a, i r0, r3 r1 r0, r3
MOV r2, #0 # a, i b r a, i r0, r3 r1 r2 r0, r3
B Test # a, i b r a, i r0, r3 r1 r2 r0, r3

Next: ADD r2, r2, r1
SUB r3, r3, #1

Test: CMP r3, #0 # a, i b r a, i r0, r3 r1 r2 r0, r3
BGT Next # a, i b r a, i r0, r3 r1 r2 r0, r3

MOV r0, r2 # r b r i r0 r1 r2 r0, r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0  r1  r2  r3  a   b   r   i

mult:  STMFD sp!,{r4-r11,lr}  # a  b  r0  r1

MOV   r3,r0  # a, i b  a, i  r0, r3  r1  r0, r3
MOV   r2,#0  # a, i b  r  a, i  r0, r3  r1  r2  r0, r3
B Test  # a, i b  r  a, i  r0, r3  r1  r2  r0, r3

Next:  ADD  r2, r2, r1  # a, i b  r  a, i  r0, r3  r1  r2  r0, r3
     SUB  r3, r3, #1

Test:  CMP  r3, #0  # a, i b  r  a, i  r0, r3  r1  r2  r0, r3
    BGT Next  # a, i b  r  a, i  r0, r3  r1  r2  r0, r3
     MOV  r0, r2  # r  b  r  i  r0  r1  r2, r0  r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0  r1  r2  r3  a   b   r   i

mult:  STMFD sp!,{r4-r11,lr}  # a   b
       MOV r3,r0               # a   i   b   a   i
       MOV r2,#0               # a   i   b   r   a   i
       B  Test                  # a   i   b   r   a   i

Next:  ADD  r2,r2,r1           # a   i   b   r   a   i
       SUB  r3,r3,#1           # a   b   r   i   r0  r1  r2  r3

Test:  CMP  r3,#0             # a   i   b   r   a   i
       BGT Next                # a   i   b   r   a   i

MOV  r0,r2                   # r   b   r   i   r0  r1  r2  r0  r3
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0  r1  r2  r3  a   b  r  i
mult:  STMFD sp!,{r4-r11,lr} # a  b  r0  r1
      MOV  r3, r0
      MOV  r2, #0
      B    Test

Next:  ADD  r2, r2, r1
      SUB  r3, r3, #1

Test:  CMP  r3, #0
      BGT  Next

MOV  r0, r2
```
Example (ARM32 Subset, naïve forward register allocation)

```
# r0  r1  r2  r3    a   b   r   i

mult:  STMFD sp!,{r4-r11,lr}  # a   b
        MOV r3, r0         # a   i   b a   i  r0   r3  r1  r0   r0   r3
        MOV r2, #0         # a   i   b r   a   i  r0   r3  r1  r2  r0   r0   r3
        B Test           # a   i   b r   a   i  r0   r3  r1  r2  r0   r0   r3

Next:  ADD  r2, r2, r1      # a   b   r   i  r0   r1  r2  r3
       SUB  r3, r3, #1      # a   b   r   i  r0   r1  r2  r3

Test:  CMP  r3, #0     # a   b   r   i  r0   r1  r2  r3
       BGT Next    # a   b   r   i  r0   r1  r2  r3

MOV  r0, r2     # r   b   r   i  r0   r1  r2  r0   r0   r3
```
### Example (ARM32 Subset, naïve forward register allocation)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult:</code></td>
<td><code>r0 r1 r2 r3 a b r i</code></td>
</tr>
<tr>
<td><code>STMFD sp!,{r4-r11,lr}</code></td>
<td><code># a b r0 r1</code></td>
</tr>
<tr>
<td><code>MOV r3,r0</code></td>
<td><code># a,i b a,i r0,r3 r1 r0,r3</code></td>
</tr>
<tr>
<td><code>MOV r2,#0</code></td>
<td><code># a,i b r a,i r0,r3 r1 r2 r0,r3</code></td>
</tr>
<tr>
<td><code>B Test</code></td>
<td><code># a,i b r a,i r0,r3 r1 r2 r0,r3</code></td>
</tr>
<tr>
<td><code>Next:</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>ADD r2,r2,r1</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>SUB r3,r3,#1</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>Test:</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>CMP r3,#0</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>BGT Next</code></td>
<td><code># a b r i r0 r1 r2 r3</code></td>
</tr>
<tr>
<td><code>MOV r0,r2</code></td>
<td><code># r b r i r0 r1 r2,r0 r3</code></td>
</tr>
</tbody>
</table>
Example (ARM32 Subset, naïve forward register allocation)

```assembly
# r0, r1, r2, r3, a, b, r, i

mult: STMFD sp!, {r4-r11, lr} # a, b, r0, r1

MOV r3, r0 # a, b, r0, r3, r1
MOV r2, #0 # a, b, r2, r0, r3
B Test # a, b, r2, r0, r3

Next: ADD r2, r2, r1 # a, b, r2, r3
SUB r3, r3, #1 # a, b, r2, r3

Test: CMP r3, #0 # a, b, r2, r3
BGT Next # a, b, r2, r0, r3

MOV r0, r2 # r, b, r2, r0, r3
LDMFD sp!, {r4-r11, pc} # r, b, r1, r0
```
Example (ARM32 Subset, naïve forward register allocation)

# r0  r1  r2  r3  a  b  r  i

**mult:**

MOV r3, r0
# a, i  b  a, i  r0, r3  r1  r0, r3
MOV r2, #0
# a, i  b  r  a, i  r0, r3  r1  r2  r0, r3
B Test
# a, i  b  r  a, i  r0, r3  r1  r2  r0, r3

**Next:**

ADD r2, r2, r1
# a  b  r  i  r0  r1  r2  r3
SUB r3, r3, #1
# a  b  r  i  r0  r1  r2  r3

**Test:**

CMP r3, #0
# a  b  r  i  r0  r1  r2  r3
BGT Next
# a  b  r  i  r0  r1  r2  r3

MOV r0, r2
# r  b  r  i  r0  r1  r2, r0  r3
MOV pc, lr
# r  b  r  i  r0  r1  r0
Basic Block Code Generation

Basic Block DAG

Cross-BB Register Allocation

Interference Graphs

HACS & Project Milestone 2 Part B
Interference Graph

Track variables that are live at the same time

Analyze backwards, one 3-address instruction at the time:

- When a variable is set then it is removed.
- When a variable is used then it is added.
Interference Graph

Track variables that are **live** at the same time

Analyze **backwards**, one 3-address instruction at the time:

1. When a variable is **set** then it is **removed**.
2. When a variable is **used** then it is **added**.
```c
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test

Next:
    r = r + b
    i = i - 1

Test:
    if i>0 goto Next

    return r;
}
```
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test

    Next:
    r = r + b
    i = i - 1

    Test:
    if i>0 goto Next  {r}

    return r;
}
```c
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test

Next:
    r = r + b
    i = i - 1

Test:
    if i>0 goto Next {r}

    return r;
}
```
```c
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test {i,r}

    Next:
    r = r + b
    i = i - 1 {i,r}

    Test: {i,r}
    if i>0 goto Next {r}

    return r;
}
```
Basic Block Code Generation

Basic Block DAG

Cross-BB Register Allocation

Interference Graphs

HACS & Project Milestone 2 Part B

**Interference Graph**

```c
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test
}

Next:
    r = r + b
    i = i - 1

Test:
    if i>0 goto Next
    return r;

Interference Graph
```

Eva Rose, Kristoffer Rose

Compiler Construction (CSCI-GA.2130-001) 10. Register Allocation

November 13, 2014 30/37
int mult(int a, int b)
{
    {a}
    i = a
    r = 0
    goto Test
    {i,r}

    Next: {b,i,r}
    r = r + b
    i = i - 1
    {i,r}

    Test: {i,r}
    if i>0 goto Next
    {r}

    return r;
}
Interference Graph

```c
int mult(int a, int b)
{
    {a}
    i = a
    r = 0
    goto Test {i,r}

    Next: {b,i,r}
    r = r + b
    i = i - 1 {i,r}

    Test: {i,r}
    if i>0 goto Next {b,i,r}

    return r;
}
```
```c
int mult(int a, int b)
{
    i = a
    r = 0
    goto Test ;
}

Next:  {b,i,r}
    r = r + b
    i = i - 1

Test:   {b,i,r}
    if i>0 goto Next  {b,i,r}

    return r;
}
```
### Interference Graph

```c
int mult(int a, int b)
{
    {a}
    i = a
    r = 0
    goto Test {b,i,r}

Next: {b,i,r}
    r = r + b
    i = i - 1 {b,i,r}

Test: {b,i,r}
    if i>0 goto Next {b,i,r}

return r;
}
```
```c
int mult(int a, int b)
{
    {a, b}
    i = a
    r = 0
    goto Test {b, i, r}

    Next: {b, i, r}
    r = r + b
    i = i - 1 {b, i, r}

    Test: {b, i, r}
    if i>0 goto Next {b, i, r}

    return r;
}
```
```c
int mult(int a, int b)
{
    {a, b}
    i = a
    r = 0
    goto Test {b, i, r}

    Next: {b, i, r}
    r = r + b
    i = i - 1 {b, i, r}

    Test: {b, i, r}
    if i>0 goto Next {b, i, r}

    return r;
}
```

Interference Graph
Example (ARM32 Subset, finished)

```
# a,r=r0  b=r1  i=r2

mult:  MOV  r2,r0
       MOV  r0,#0
       B  Test

Next:  ADD  r0,r0,r1
       SUB  r2,r2,#1

Test:  CMP  r2,#0
       BGT  Next
       MOV  pc,lr
```
Example (Full ARM32)

```plaintext
# a, i = r0, b = r1, r = r2

mult: MOVS r2, r0
      MOV  r0, #0
      B   Test
Next: ADD  r0, r0, r1
      SUBS r2, r2, #1
Test: BPL  Next
      MOV  pc, lr
```
**Spill Example (IR)**

**Start:**

\(i = a\)

\(r = 0\)

goto Test

**Next:**

\(r = r + b\)

\(i = i - 1\)

**Test:**

if \(i > 0\) goto Next

**End:**

return \(r\);
Questions?
Bonus Division

.global udiv64
udiv64:
  adds r0, r0, r0
  adc r1, r1, r1

.rept 31
  cmp r1, r2
  subcs r1, r1, r2
  adcs r0, r0, r0
  adc r1, r1, r1
.endr

cmp r1, r2
subcs r1, r1, r2
adcs r0, r0, r0
bx lr
1. Basic Block Code Generation
2. Basic Block DAG
3. Cross-BB Register Allocation
4. Interference Graphs
5. HACS & Project Milestone 2 Part B
Questions?