Compiler Optimization
Compiler optimization is

- to generate **better** code
- not to generate **optimal** code
  - it is an NP-complete problem

**What is a better version?**

- Same program output
- Better one or more of the followings
  - Execution time
  - Memory usage
  - Energy/power consumption
  - Security
  - Reliability
  - Debuggability
  - Other criteria
Compiler optimization is essentially a transformation

- Delete something
- Add something
- Move something
- Modify something

Transform code or layout?

- Code-related optimizations
  - Optimizes *what* code is generated
- Layout-related optimizations
  - Optimizes *where* in memory code and data is placed
Layout-Related Optimizations
Layout-Related Optimizations

- Seeks to improve caching (or paging) behavior by
  - changing the layout of data or code
  - exploiting knowledge of machine memory hierarchy

- Change code layout

```c
f() {
    ... call h();
}
g() {
    ...
}
h() {
    ...
}
```

OR

```
<table>
<thead>
<tr>
<th>code of f()</th>
<th>code of g()</th>
</tr>
</thead>
<tbody>
<tr>
<td>code of h()</td>
<td></td>
</tr>
</tbody>
</table>

OR

```
<table>
<thead>
<tr>
<th>code of f()</th>
<th>code of h()</th>
</tr>
</thead>
<tbody>
<tr>
<td>code of g()</td>
<td></td>
</tr>
</tbody>
</table>
```
Which Code Layout is Better?

Assume:
- data cache has one N-word line
- the size of each function is N/2-word long
- access sequence is “g, f, h, f, h, f, h”

<table>
<thead>
<tr>
<th>code of f()</th>
<th>code of g()</th>
</tr>
</thead>
<tbody>
<tr>
<td>code of h()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code of f()</th>
<th>code of h()</th>
</tr>
</thead>
<tbody>
<tr>
<td>code of g()</td>
<td></td>
</tr>
</tbody>
</table>

6 cache misses

\[ g, f, h, f, h, f, h \]

2 cache misses
Data Layout Optimization

- Change the variable declaration order

```
struct S {
    int x1;
    int x2[200];
    int x3;
} obj[100];
for(...) {
    ... = obj[i].x1 + obj[i].x3;
}
```

```
struct S {
    int x1;
    int x3;
    int x2[200];
} obj[100];
for(...) {
    ... = obj[i].x1 + obj[i].x3;
}
```

- Improved spatial locality
  - Now x1 and x3 likely reside in same cache line
  - Access to x3 will always hit in the cache
Data Layout Optimization

- Change AOS (array of structs) to SOA (struct of arrays)

  ```c
  struct S {
    int x1;
    int x2;
  } obj[100];
  for(...) {
    ... = obj[i].x1 * 2;
  }
  for(...) {
    ... = obj[i].x2 * 2;
  }

  struct S {
    int x1[100];
    int x2[100];
  } obj;
  for(...) {
    ... = obj.x1[i] * 2;
  }
  for(...) {
    ... = obj.x2[i] * 2;
  }
  ```

- Improved spatial locality for x1 and x2
- More efficient vectorization (no need to gather/scatter)
  - Gather: Load data from dispersed locations into vector unit
  - Scatter: Store data from dispersed locations into vector unit
Code-Related Optimizations
Code-Related Optimizations

- Modifying code  e.g. strength reduction
  \[ A = 2a; \quad \equiv \quad A = a \ll 1; \]

- Deleting code  e.g. dead code elimination
  \[ A = 2; \quad A = a; \quad \equiv \quad A = a; \]

- Moving code  e.g. code motion
  \[ A = x \times y; \quad B = A + 1; \quad C = y; \quad \equiv \quad A = x \times y; \quad C = y; \quad B = A + 1; \]

- Inserting code  e.g. data prefetching
  \[
  \text{while (p!=NULL)} \{ \quad \ldots \quad \text{p=p->next;} \quad \} \\
  \equiv \\
  \text{while (p!=NULL)} \{ \quad \text{prefetch(p->next);} \quad \ldots \quad \text{p=p->next;} \quad \} 
  \]
Optimization Categories

- Optimize at what representation level?
  - Source code level
  - IR level
  - Machine code level

- Optimize for specific machine?
  - Machine independent — typically at IR or source level
  - Machine dependent — typically at machine code level

- Optimize across control flow?
  - Local optimization — scope within straight line code
  - Global optimization — scope across control structures

- Optimize across procedures?
  - Intra-procedural — scope within individual procedure
  - Inter-procedural — scope across different procedures
    (Analyze callee to optimize caller and vice versa)
Local Optimizations
Local Optimizations

- Optimizations where the scope includes no control flow
  - Limited in scope but can still do useful things

- **Strength Reduction**
  - The idea is to replace expensive operations (multiplication, division) by less expensive operations (add, sub, shift, mov)
  - Some are redundant and thus can be deleted
    - e.g. \( x = x + 0; \ y = y \times 1; \)
  - Some can be simplified
    - e.g. \( x = x \times 0; \ y = y \times 8; \)
    - can be replaced by \( x = 0; \ y = y \ll 3; \)
  - Is also machine-dependent since it uses knowledge about the underlying machine (e.g. multiplication is expensive)
More Local Optimizations

Constant folding

- Operations on constants can be computed at compile time
- In general, if \( x = y \text{ op } z \) and \( y \) and \( z \) are constants then compute at compile time and replace

Example:

```c
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print(”error”);
```

Can be transformed to ...

```c
x = 200;
if (false) print(”error”);
```

- Is machine-independent since it is beneficial regardless of machine
Global Optimizations and Control Flow Analysis
Global Optimizations and Control Flow

- Global optimization include more powerful optimizations
  - Can go across control structures (but not calls)
  - In effect, scope of optimization is one function
    (Not global as in entire program as the name implies)
  - E.g. Global Constant Propagation (GCP):
    - Replace variables with constants if value is known
      \[ X = 7; \]
      \[ \ldots \]
      \[ Y = X + 3; // \text{Can be replaced by } Y = 10; \]
    - Needs knowledge of control flow
      (Whether evaluation at point A happens before point B)

- Global optimization requires control flow analysis
  - Control flow analysis: Compiler analysis that determines
    flow of control during execution of a function
  - Constructs a control flow graph that describes the flow
Basic Block

- A **basic block** is a maximal sequence of instructions that
  - Except the first instruction, there are no other labels;
  - Except the last instruction, there are no jumps;

- Therefore,
  - Can only jump into the beginning of a block
  - Can only jump out at the end of a block

- Are units of control flow that cannot be divided further
  - All instructions in basic block execute or none at all
A control flow graph is a directed graph in which:

- Nodes are basic blocks
- Edges represent the flow of execution
  - Control statements such as if-then-else, while-loop, for-loop introduce control flow edges

CFG is widely used to represent a program.

CFG is widely used for program analysis, especially for global analysis/optimization.
Example

L1; t:= 2 * x;
   w:= t + y;
   if (w<0) goto L3
L2: ...
...
L3: w:= -w
...
Construction of CFG

- **Step 1:** partition code into basic blocks
  - Identify **leader** instructions that are
    - the first instruction of a program, or
    - target instructions of jump instructions, or
    - instructions immediately following jump instructions
  - A basic block consists of a leader instruction and subsequent instructions before the next leader

- **Step 2:** add an edge between basic blocks B1 and B2 if
  - there exist a jump from B1 to B2, or
  - B2 follows B1, and B1 does not end with unconditional jump
    - B1 ends with a conditional jump
    - B1 ends with a non-jump instruction (B2 is a target of a jump)
Example

01. A=4
02. T1=A*B
03. L1: T2=T1/C
04: if (T2<W) goto L2
05: M=T1*K
06: T3=M+1
07: L2: H=I
08: M=T3-H
09: if (T3>0) goto L3
10: goto L1
11: L3: halt
Example

01. A=4
02. T1=A*B
03. L1: T2=T1/C
04: if (T2<W) goto L2
05: M=T1*K
06: T3=M+1
07: L2: H=I
08: M=T3-H
09: if (T3>0) goto L3
10: goto L1
11: L3: halt
Global Optimizations

- Extend optimizations to flow of control, i.e. CFG

```
X := 3;
if (B > 0)
    Y := Z + W;
Y := 0;
A := 2 * X;

X := 3;
if (B > 0)
    Y := Z + W;
Y := 0;
A := 2 * 3;
```

- How do we know it is OK to globally propagate constants?
Correctness

In particular, there are situations that prohibit this optimization

To replace x by a constant C correctly, we must know

- Along all paths, the last assignment to X is “X:=C”
- All paths often include branches an even loops
  - Usually it is not trivial
Many compiler optimizations depend on knowing some property X at a particular point in program execution. Need to prove at that point property X holds along all paths. This ensures correctness by being conservative. If unsure, don’t do the optimization. While losing optimization opportunities, guarantees correctness. Examples include Global Constant Propagation (GCP): X = 7; ... Y = X + 3; // Replace by Y = 10, if X didn’t change. This involves data flow and control flow.
Optimizations Need to be Conservative

Many compiler optimizations depend on knowing some property $X$ at a particular point in program execution.

- Need to prove at that point property $X$ holds along all paths.
- Need to be **conservative** to ensure correctness.
  - An optimization is enabled only when $X$ is definitely true.
  - If not sure if it is true or not, it is safe to say don’t know.
  - If you don’t know, you don’t do the optimization.
  - May lose opt. opportunities but guarantees correctness.

E.g. Global Constant Propagation (GCP):

```
X = 7;
...
Y = X + 3; // Replace by Y = 10, if X didn’t change
```

Needs knowledge of data flow, as well as control flow (Whether data flow is interrupted between points A and B).
Many compiler optimizations depend on knowing some property $X$ at a particular point in program execution:

- Need to prove at that point property $X$ holds along all paths.
- Need to be conservative to ensure correctness.
  - An optimization is enabled only when $X$ is definitely true.
  - If not sure if it is true or not, it is safe to say don’t know.
  - If you don’t know, you don’t do the optimization.
  - May lose opt. opportunities but guarantees correctness.

Property $X$ often involves data flow of program:

- E.g. Global Constant Propagation (GCP):
  
  ```
  X = 7;
  ...
  Y = X + 3; // Replace by Y = 10, if X didn’t change
  ```
  - Needs knowledge of data flow, as well as control flow.
  - (Whether data flow is interrupted between points A and B.)
Global Optimizations and Data Flow Analysis
Dataflow Analysis Framework

- **Dataflow analysis**: Compiler analysis that determines what values get propagated from point A to point B
  - Requires CFG since values flow through control flow edges
Dataflow Analysis Framework

Dataflow analysis: Compiler analysis that determines what values get propagated from point A to point B
- Requires CFG since values flow through control flow edges

Dataflow analysis framework: Framework for dataflow analysis that guarantees optimizations are conservative
- Defined by: \( \{ D, V, \wedge, F: V \rightarrow V \} \)
- \( D \): Direction of propagation (forwards or backwards)
- \( V \): Set of values (depends on analyzed property)
  - Value for GCP: set of variables with constant values
- \( \wedge \): Meet operator \( (V \wedge V \rightarrow V) \)
  - Defines behavior when values meet at control flow merges
- \( F \): Transfer function \( F: V \rightarrow V \)
  - Defines what happens to value within a basic block
Dataflow Analysis Framework

- **Dataflow analysis**: Compiler analysis that determines what values get propagated from point A to point B
  - Requires CFG since values flow through control flow edges

- **Dataflow analysis framework**: Framework for dataflow analysis that guarantees optimizations are **conservative**
  - Defined by: \{D, V, ∧, F: V → V \}
  - D: Direction of propagation (forwards or backwards)
  - V: Set of values (depends on analyzed property)
    - Value for GCP: set of variables with constant values
  - ∧: Meet operator (V ∧ V → V)
    - Defines behavior when values meet at control flow merges
  - F: Transfer function F: V → V
    - Defines what happens to value within a basic block

- **Goal**: To assign V for every point in program
  → aids optimization
Global Constant Propagation (GCP)

Rather than bore you with math, let’s learn by example: global constant propagation (GCP)

What is Global Constant Propagation?
- At compile time, if the value of a variable is a constant, replace the variable with the constant
- “Global” means we substitute across basic blocks and control flow
- At compile time, we don’t know which path is taken
Global Constant Propagation (GCP)

- Rather than bore you with math, let’s learn by example: **global constant propagation (GCP)**

- **What is Global Constant Propagation?**
  - At compile time, if the value of a variable is a constant, replace the variable with the constant
  - “Global” means we substitute across basic blocks and control flow
  - At compile time, we don’t know which path is taken

- Compiler can apply a dataflow analysis framework to the problem to ensure conservative optimization
  - What is $D$, $V$, $\land$, $F: V \rightarrow V$ in this context?
What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
- A given variable can be in one of following states:
  - \( x=1, x=2, \ldots \) // defined a constant
  - \( x=* \) // not defined yet
  - \( x=# \) // don’t know (not provably constant)
What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
- A given variable can be in one of following states:
  - \( x=1, x=2, \ldots \) // defined a constant
  - \( x=* \) // not defined yet
  - \( x=# \) // don’t know (not provably constant)
- \( V \) for GCP: Set of values where each value is the set of variables and their respective states.
- Examples of values in \( V \): \( \{x=*, y=10, z=#\}, \{x=1, y=#, z=5\} \)
- Goal for GCP is to assign a value to each point in program
What is $\wedge$?

- $\wedge$: Meet operator ($V \wedge V \rightarrow V$)
  - Defines behavior when values meet at control flow merges
  - Given
    - $V_{in}(B)$ — value at the entry of basic block $B$
    - $V_{out}(B)$ — value at the exit of basic block $B$
  - $V_{in}(B) = \wedge V_{out}(P)$ for each $P$, where $P$ is a predecessor of $B$

Example of $\wedge$ operator for GCP:

$\{x=*, y=2, z=3\} \wedge \{x=1, y=2, z=10\} = \{x=1, y=2, z=#\}$

- Why is $z=#$ after the meet?
- Why is $x=1$ after the meet?

Relationship between values in $V$ given by a meet operator is called a **Semilattice**
Semilattice

Semilattice for GCP (when there is one variable):

\{x=\ast\} \\
\{\ldots\} \quad \{x=-1\} \quad \{x=0\} \quad \{x=1\} \quad \{\ldots\} \\
\{x=\#\}

\wedge \text{ operator is defined by the } \textbf{Greatest Lower Bound (GLB)} \text{ between two values}

\begin{align*}
\{x=\ast\} \wedge \{x=1\} &= \{x=1\} \\
\{x=0\} \wedge \{x=1\} &= \{x=\#\}
\end{align*}

\text{Downward direction is always the conservative choice}

\text{In effect, GLB is the least } \textbf{conservative} \text{ but } \textbf{correct} \text{ choice}

\text{Semilattice essentially defines what meet operator means}
In a semilattice, there are two special values: $\top$ and $\bot$.
In a semilattice, there are two special values: \( \top \) and \( \bot \).

\( \top \): Called **Top Value**
- Initial value when analysis begins
- For GCP: \( \{x=*, y=*, z=*\} \)
- Value is refined in the course of analysis
In a semilattice, there are two special values: \( \top \) and \( \bot \).

- \( \top \): Called **Top Value**
  - Initial value when analysis begins
  - For GCP: \{\( x=\ast \), \( y=\ast \), \( z=\ast \)\}
  - Value is refined in the course of analysis

- \( \bot \): Called **Bottom Value**
  - Value which can be refined no further
  - For GCP: \{\( x=\# \), \( y=\# \), \( z=\# \)\}
  - Meaning: none of the variables are provably constant

Analysis iteratively refines values until they stabilize somewhere between \( \top \) and \( \bot \).
What is $F$?

- **$F$: Transfer function ($F: V \rightarrow V$)**
  - Defines what happens to value within a basic block
  - Given
    - $V_{in}(B)$ — value at the entry of basic block $B$
    - $V_{out}(B)$ — value at the exit of basic block $B$
  - $V_{out}(B) = F(V_{in}(B))$

- **$F$ for GCP:**
  $$V_{out}(B) = (V_{in}(B) - DEF_{v}(B)) \cup DEF_{c}(B)$$

  where $DEF_{v}(B)$ contains variable definitions in $B$
  $DEF_{c}(B)$ contains constant definitions in $B$

- Easier to reason about if you treat each individual statement as a basic block
There are two modes of propagation: \textbf{F} and \textcopyright{∧}.

- **Function \textbf{F}**— propagates values through basic blocks
  - Variables in \( \text{DEF}_v \) are set to \#.
  - Variables in \( \text{DEF}_c \) are set to constant value.

- **\textcopyright{∧} operator**— propagates values through CFG edges
  - Merges values from multiple predecessor blocks.
What is D?

- **D**: Direction of propagation (forwards or backwards)

Forward Analysis

Backward Analysis
What is D?

- Values are propagated forward: **Forward Analysis**
- Values are propagated backward: **Backward Analysis**
- GCP is an example of a Forward Analysis
  - Starting from a constant definition, the ‘constantness’ of a variable propagates forward through CFG
- We will see an example of Backward Analysis soon
In this example, constants can be propagated to $X+1$, $2X$.
In this example, constants can be propagated to \( X+1 \), \( 2*X \)
In this example, constants can be propagated to $X+1$, $2*X$.
In this example, constants can be propagated to $X+1$, $2*X$
In this example, constants can be propagated to \(X+1, 2*X\)
In this example, constants can be propagated to $X+1$, $2X$. 

```
X:=3;
if (B>0)
Y:=Z+W;
X:=4;
Y:=0;
X:=X+1
A:=2*X;
```
Example GCP without Loop

In this example, constants can be propagated to $X+1$, $2X$
In this example, constants can be propagated to $X+1$, $2X$
In this example, loop prevents any constant propagation

```
X := 3;
if (B > 0)
    Y := Z + W;
    X := 4;
Y := 0;
X := X + 1
A := 2 * X;
```
In this example, loop prevents any constant propagation
In this example, loop prevents any constant propagation

\[ X = * \]
\[ X = 3 \]
\[ X := 3; \]
\[ \text{if } (B > 0) \]
\[ Y := Z + W; \]
\[ X := 4; \]
\[ Y := 0; \]
\[ X := X + 1 \]
\[ A := 2 \times X; \]
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.

```plaintext
X := 3;
if (B > 0)
Y := Z + W;
X := 4;
Y := 0;
X := X + 1
A := 2 * X;
```

NOT SO FAST!
In this example, loop prevents any constant propagation.

```
X:=3;
if (B>0)
    Y:=Z+W;
X:=4;
Y:=0;
X:=X+1
```

```
A:=2*X;
X:=X+1
```
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
In this example, loop prevents any constant propagation.
Forward Analysis Algorithm

- Pseudocode for Forward Analysis

\[
\text{for (each basic block } B \text{) } V_{out}(B) = \top; \\
W = \{\text{all basic blocks}\}; \\
\text{while } (W \neq \emptyset) \{ \\
\quad B = \text{choose basic block from } W; \\
\quad V_{in}(B) = \bigwedge_{P \text{ is a predecessor of } B} V_{out}(P) \\
\quad V_{out}(B) = F(V_{in}(B)) \\
\quad \text{if } (V_{out}(B) \text{ is changed}) \ W = W \cup \{B\text{'s successors}\} \\
\}\n\]

- \(\bigwedge\) and \(F\) defined differently for each type of analysis

- Will it eventually stop?
  - If there are loops, we may go through the loop many times
  - Is there a possibility of an infinite loop?

- Will it give me an accurate solution?
Termination Problem

- Existence of \( \perp \) value ensures termination
  - Values start from \( \top \)
  - Values can only go down in the semilattice
  - Value for single variable can change at most twice
    ... from * to C, and from C to #

- Computational complexity (\( V = \text{vars}, N = \text{nodes} \))
  - Each node can only change value \( 2 \times V \) times
    (Twice for each variable)
  - There are N nodes in the CFG
  - Maximal complexity: \( O(2 \times V \times N) = O(V \times N) \)
  - Practical complexity: \( O(N) \)
    - With depth-first traversal of nodes in \( W \),
    Nodes in straight-line code change only once
    Nodes in loops change at most twice
    (Loops always converge on second traversal, as we saw)
Accuracy Problem

- A few different types of solutions:
  - IDEAL: Meet of all possible paths \( F_P \) to this point
    \[
    \text{IDEAL}(B) = \land_{P \text{ is possible path from ENTRY to } B} F_P(V_{ENTRY})
    \]
  - MOP (Meet-Over-Paths): Meet of all paths in CFG
    \[
    \text{MOP}(B) = \land_{P \text{ is path in } \text{CFG} \text{ from ENTRY to } B} F_P(V_{ENTRY})
    \]
  - MFP (Maximum Fixed Point): given iterative solution

- MFP \( \leq \) MOP \( \leq \) IDEAL (in semilattice)
  - MOP \( \leq \) IDEAL: Why?
    - Paths in CFG is a superset of all possible paths
    - \( MOP = \text{IDEAL} \land V_{\text{never taken paths}} \leq \text{IDEAL} \) (since GLB)
  - MFP \( \leq \) MOP: Why?
    - MFP stops only when fixed point is reached:
      Covers all paths in MOP, even for limitless iterations
    - For GCP: sometimes MFP < MOP (next slide)

- MFP is correct but not maximal (in short conservative)
When is \( \text{MFP} < \text{MOP} \)?

Assume \( V_{\text{ENTRY}} \equiv \{ A = *, B = *, C = * \} \):

- \( P_1: A=1; B=2; \)
- \( P_2: A=2; B=1; \)
- \( B: C=A+B; \)

\( \text{MOP} \equiv F_B(F_{P_1}(V_{\text{ENTRY}})) \land F_B(F_{P_2}(V_{\text{ENTRY}})) \)
\[ \equiv \{ A = 1, B = 2, C = 3 \} \land \equiv \{ A = 2, B = 1, C = 3 \} \]
\[ \equiv \{ A = \#, B = \#, C = 3 \} \]

\( \text{MFP} \equiv F_B(F_{P_1}(V_{\text{ENTRY}}) \land F_{P_2}(V_{\text{ENTRY}})) \)
\[ \equiv F_B(\{ A = \#, B = \#, C = * \}) \equiv \{ A = \#, B = \#, C = \# \} \]

Refer to Chapter 9.3 in textbook
Once constants have been globally propagated, we would like to eliminate the dead code.

\[
x := 3; \\
\text{if } (b > 0) \\
y := z + w; \\
y := 0; \\
z := 2 \times x;
\]
Once constants have been globally propagated, we would like to eliminate the dead code

```
x:=3;
if (b>0)
    y:=z+w;
    y:=0;
z:=2*3;
```
Once constants have been globally propagated, we would like to eliminate the dead code

\[
\begin{align*}
x &:= 3; \\
\text{if } (b > 0) & \\
y &:= z + w; \\
y &:= 0; \\
z &:= 2 \times 3;
\end{align*}
\]
A **dead statement** calculates a value that is not used later.

Otherwise, it is a **live statement**

In the example, the 1st statement is dead, the 2nd statement is live.
Global Liveness Analysis (GLA)

A variable $X$ is live at statement $S$ if
- There exists a statement $S_2$ after $S$ that uses $X$
- There is a path from $S$ to $S_2$
- There is no intervening assignment to $X$ between $S$ and $S_2$
Global Liveness Analysis (GLA)

- A variable X is live at statement S if:
  - There exists a statement S2 after S that uses X
  - There is a path from S to S2
  - There is no intervening assignment to X between S and S2

Again a dataflow analysis framework can be applied
- What is $D, V, \land, F: V \rightarrow V$ in this context?

What is $D$?
- Liveness Analysis is a Backward Analysis
  - Starting from a use, the ‘liveness’ of a variable propagates backward through CFG
- Changes direction of $\land$ operator and transfer function
Forward and Backward Analysis Again

Forward Analysis

Backward Analysis
What is \( V \)?

- Definition: Set of values in property under analysis
  - \( V \) for GLA: Each value is a set of live variables
  - Example values: \( \{x, y, z\}, \{y\} \)

- \( \top \): initial value at the beginning
  - \( \top \) for GLA = \( \{\} \)
  - Start with assumption that no variables are live

- \( \bot \): the don’t know value
  - \( \bot \) for GLA = \( \{\text{all variables in function}\} \)
  - Meaning: none of the variables are provably dead
What is $\land$?

$\land$: Meet operator ($V \land V \rightarrow V$) for backward analysis

- Defines behavior when values meet at control divergence
- Given
  - $V_{in}(B)$ — value at the entry of basic block $B$
  - $V_{out}(B)$ — value at the exit of basic block $B$

  - $V_{out}(B) = \land V_{in}(S)$ for each $S$, where $S$ is successor of $B$
  - Note the reversal in direction! GLA is a backward analysis.

$\land$ operator for GLA:

- Meet operator is a simple union $\cup$
- Example: $\{x, y\} \land \{y, z\} = \{x, y\} \cup \{y, z\} = \{x, y, z\}$
- Union operation monotonically increases set, hence values form a semilattice from $\top$ to $\bot$
What is F?

- **F**: Transfer function \( F: V \rightarrow V \) for backward analysis
- Defines what happens to value within a basic block
- Given
  - \( V_{in}(B) \) — value at the entry of basic block \( B \)
  - \( V_{out}(B) \) — value at the exit of basic block \( B \)
- \( V_{in}(B) = F( V_{out}(B) ) \)
- Again note the reversal in direction!

F for GLA:
\[
V_{in}(B) = ( V_{out}(B) - DEF(B) ) \cup USE(B)
\]
where \( DEF(B) \) contains variable definitions in \( B \)
\( USE(B) \) contains variable uses in \( B \)

- Easier to reason about if you treat each individual statement as a basic block
Liveness Example

\[ b = b + c \]

\[ a = d + 1; \]
Liveness Example

\[ b = b + c \]
\[ a = d + 1; \]
Liveness Example

\[
\begin{align*}
&\text{V}_{\text{in}}(B1) \\
&\text{V}_{\text{in}}(B2) \\
&b = b + c \\
&\text{V}_{\text{out}}(B2) \\
&\text{V}_{\text{out}}(B3) = \{a, b\} \\
&\text{V}_{\text{out}}(B3) \\
&\text{V}_{\text{in}}(B1) \\
&a = d + 1;
\end{align*}
\]
Liveness Example

```
b=b+c
a=d+1;
```

Two sets:
- DEF={a}
- USE={d}
Liveness Example

Two sets:
DEF={a}
USE={d}
Liveness Example

\[ b = b + c \]
\[ a = d + 1; \]

Two sets:
- DEF = \{a\}
- USE = \{d\}

\[ V_{\text{in}}(B1) \]
\[ V_{\text{out}}(B1) \]

\[ V_{\text{in}}(B2) = \{b, c\} \]
\[ V_{\text{out}}(B2) \]

\[ V_{\text{in}}(B3) = \{b, d\} \]
\[ V_{\text{out}}(B3) = \{a, b\} \]
Liveness Example

- $b = b + c$
- $a = d + 1$

**$V_{in}(B1)$:** $\{b, c, d\}$

**$V_{in}(B2)$:** $\{b, c\}$

**$V_{out}(B2)$**

**$V_{in}(B3)$:** $\{b, d\}$

**$V_{out}(B3)$:** $\{a, b\}$

**Two sets:**
- $DEF = \{a\}$
- $USE = \{d\}$
Backward Analysis Algorithm

- Pseudocode for Backward Analysis
  - for (each basic block B) $V_{in}(B) = \top$;
  - $W = \{\text{all basic blocks}\}$;
  - while ($W \neq \emptyset$) {
    - $B =$ choose basic block from $W$;
    - $V_{out}(B) = \bigwedge S \text{ is a successor of } B V_{in}(S)$
    - $V_{in}(B) = F(V_{out}(B))$
    - if ($V_{in}(B)$ is changed) $W = W \cup \{B\text{'s predecessors}\}$
  }

- Note the reversal in direction compared to forward analysis

- Will backward analysis for GLA eventually stop?
  - Again existence of $\bot$ value ensures termination
  - Node value can change $V$ times, where $V$ is number of vars
  - Maximal complexity: $O(V \times N)$
  - Practical complexity: $O(N)$, with in-depth traversal
Is GLA Accurate?

- For GLA, $MFP = MOP \leq IDEAL$
- $MOP \leq IDEAL$: CFG is a superset of all paths (like GCP)
- $MFP = MOP$: Why?
  - MFP emulates all paths in MOP (like GCP)
  - Unlike GCP, transfer function of GLA is **distributive**
    
    
    $$MOP \equiv F_B(F_{P1}(V)) \land F_B(F_{P2}(V))$$
    
    $$\equiv F_B(F_{P1}(V) \land F_{P2}(V)) \equiv MFP$$

- If all paths in CFG can be taken, GLA is maximal
- Refer to Chapter 9.3 in textbook for discussion on distributive transfer functions
Comparison of GCP and GLA

D: Direction of propagation
- GCP: Forward
- GLA: Backward

V: Set of values propagated
- GCP: Whether each variable is constant, and if so the value
- GLA: Set of live variables

∧: Meet operator
- GCP: Defined by semilattice (Top $\rightarrow$ Constant $\rightarrow$ Bottom)
- GLA: Simply the set union operator

F: Transfer function
- GCP: Subtract variable definitions, add constant definitions
- GLA: Add variable uses, subtract variable definitions
Global dead code elimination is based on global liveness analysis (GLA)

- Dead code detection
  - A statement $x = \ldots$ is dead code if $x$ is dead after this statement
  - Dead statement can be deleted from the program

Global register allocation is also based on GLA

- Live variables should be placed in registers
- Registers holding dead variables can be reused
Register Allocation
What is Register Allocation?

- Process of assigning (a large number of) variables to (a small number of) CPU registers

- Registers are fast
  - access to memory: 100s of cycles
  - access to cache: a few to 10s of cycles
  - access to registers: 1 cycle

- But registers are limited in number
  - x86: 8 regs, MIPS: 32 regs, ARM: 32 regs ...

- Goals of register allocation:
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live
Local Register Allocation

Allocate registers basic block by basic block
- Makes allocation decisions on a per-block basis
- Hence the prefix ‘local’
- Uses results of Global Liveness Analysis

Requires only a single scan through each basic block
- Keeps track of two tables:
  - Register table: which regs are currently allocated and where
  - Address table: location(s) where each variable is stored
    (locations can be: register, stack memory, global memory)
- For every use of variable:
  - If variable is already in reg, no action
  - If not, allocate reg to variable from available regs
  - If no available regs, select reg for displacement
Local Register Allocation

Which register should be displaced?
- Register whose value is no longer live (given by GLA)
- Register whose value has a copy in another location
- These registers can be safely recycled
- Otherwise the register needs to be spilled

Spill: storing variable in its own memory location
- Own memory location can be in
  - Stack memory: local variables, temporary variables
  - Global memory: global variables
- Generate store instruction to memory on assignment
- Generate load instruction from memory on use

At the end of basic block all live registers are spilled
- Makes all registers available for next basic block allocation
  (Gives allocator clean slate for next basic block)
- Can be source of inefficiency due to unnecessary spills
  → Addressed by Global Register Allocation
Global Register Allocation

- Allocates registers across basic blocks
- Relies on Global Liveness Analysis just like local register allocation
- Three popular register allocation algorithms
  1. Graph coloring allocator
  2. Linear scan allocator
  3. ILP (Integer Linear Programming) allocator
Algorithm steps:
1. Identify live range interference using GLA
2. Build register interference graph
3. Attempt K-coloring of the graph
   - K is the number of available registers
4. If none found, modify the program, rebuild graph until K-coloring can be obtained
   - Insert spill code to the program
**Live Range Interference**

- **Live Range**: Set of program points where a variable is live.
  - Two live ranges interfere if there is an overlap.
  - Vars with interfering ranges cannot reside in the same register.

```
x := ...
y := ...
:= ...
x := ...
y := ...
```
Live Range Interference

**Live Range**: Set of program points where a variable is live

- Two live ranges interfere if there is an overlap
- Vars with interfering ranges cannot reside in same register

\[
x := ... \quad \text{x is live}
\]
\[
y := ... \quad \text{y is live}
\]

\[
x := ... \\
y := ...; \quad y := ...x; \\
\]

\[
:= y
\]
**Live Range Interference**

**Live Range**: Set of program points where a variable is live

- Two live ranges interfere if there is an overlap
- Vars with interfering ranges cannot reside in same register

We annotate each program point (between two statements) to explicitly show the interference.
Example of GLA and interfering live ranges

```
a := b + c;
d := -a;
e := d + f;
f := 2 * e;
b := f + c;
b := d + e;
e := e - 1;
```
Register Interference Graph

- Construct **Register Interference Graph (RIG)** such that
  - Each node represents a variable
  - An edge between two nodes $V_1$ and $V_2$ represents an interference in live ranges

- Based on RIG,
  - Two variables can be allocated in the same register if there is no edge between them
  - Otherwise, they cannot be allocated in the same register
In the RIG for our example:

- b, c cannot be in the same register
- a, b, d can be in the same register
Graph coloring is a theoretical problem where ...
- A coloring of a graph is an assignment of colors to nodes such that nodes connected by an edge have different colors
- A graph is k-colorable if it has a coloring with k colors

Problem of register allocation in RIG maps to graph coloring problem
- Instead of assigning k-colors, we need to assign k registers
- K is the number of available machine registers
- If the graph is k-colorable, we have a register assignment that uses no more than k registers
This is an coloring of our example RIG using 4 colors

There is no solution with less than 4 colors
Using the coloring result, map it back to the code

a:=b+c;
d:=-a;
e:=d+f;

f:=2*e;

b:=d+e;
e:=e-1;

b:=f+c;
Using the coloring result, map it back to the code

\[ a \rightarrow R1 \]
\[ b \rightarrow R2 \]
\[ c \rightarrow R3 \]
\[ d \rightarrow R2 \]
\[ e \rightarrow R1 \]
\[ f \rightarrow R4 \]

\[ f := 2 \times e; \]
\[ b := d + e; \]
\[ e := e - 1; \]
\[ b := f + c; \]
\[ R1 := R2 + R3; \]
\[ R2 := -R1; \]
\[ R1 := R2 + R4; \]
After Register Allocation

Using the coloring result, map it back to the code

\begin{align*}
    a &\rightarrow R1 \\
    b &\rightarrow R2 \\
    c &\rightarrow R3 \\
    d &\rightarrow R2 \\
    e &\rightarrow R1 \\
    f &\rightarrow R4
\end{align*}

\begin{align*}
    b := d + e; \\
    e := e - 1; \\
    b := f + c; \\
    R1 := R2 + R3; \\
    R2 := -R1; \\
    R1 := R2 + R4; \\
    R4 := 2 * R1;
\end{align*}
Using the coloring result, map it back to the code:

- \( a \rightarrow R1 \)
- \( b \rightarrow R2 \)
- \( c \rightarrow R3 \)
- \( d \rightarrow R2 \)
- \( e \rightarrow R1 \)
- \( f \rightarrow R4 \)

\[
\begin{align*}
R1 &= R2 + R3; \\
R2 &= -R1; \\
R1 &= R2 + R4; \\
R4 &= 2 \times R1; \\
R2 &= R2 + R1; \\
R1 &= R1 - 1; \\
b &= f + c;
\end{align*}
\]
After Register Allocation

Using the coloring result, map it back to the code

\[
R1 := R2 + R3;  \\
R2 := -R1;  \\
R1 := R2 + R4;  \\
R4 := 2*R1;  \\
R2 := R4 + R3;  \\
R2 := R2 + R1;  \\
R1 := R1 - 1;  \\
R2 := R2 + R4;  \\
R1 := R1 - 1;
\]
How is Graph Coloring Performed?

For graph G and k>2, determining whether G is k-colorable is NP complete

- Problem of k-register allocation is NP complete
- In practice: use heuristic polynomial algorithm that gives close to optimal allocations most of the time
- Chaitin’s graph coloring is a popular heuristic algorithm
  - Most backends of GCC use Chaitin’s algorithm by default

What if k-register allocation does not exist?

- Spill a register to memory to reduce RIG and try again
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what...
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what...
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what
Chaitin’s Graph Coloring

**Corollary**: Given graph $G$ for a $k$-coloring problem
- Let $G'$ be the graph after removing a node with fewer than $k$ neighbors
- If $G'$ can be $k$-colored then $G$ can be $k$-colored

**Insight**: Solving for $G'$ is easier than solving for $G$, so solve for $G'$ instead of $G$

**Algorithm**
- Phase 1: Repeat until there are no nodes left
  - Pick a node $V$ with fewer than $k$ neighbors
  - Put $V$ on a stack and remove it and its associated edges from the graph
- Phase 2: Assign colors to nodes on the stack in LIFO order
  - Pick a color that is different from its neighbors
  - Such a color is guaranteed to exist due to corollary (Analogous to coloring $G$ after adding removed node to $G'$)
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{\}$
Chaitin’s algorithm applied to our example where $k=4$
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where k=4

Stack={a}
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d\}$
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d\}$
Chaitin’s algorithm applied to our example where $k=4$
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b\}$
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b,c\}$
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b,c\}$
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where k=4

Stack={a,d,b,c,e}
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b,c,e\}$
Coloring Result

Starting assigning colors to f, e, b, c, d, a

Diagram showing a graph with nodes labeled f, e, b, c, d, a and colors R1, R2, R3, R4.
Is Chaitin’s Graph Coloring Optimal?

According to Chaitin’s algorithm:
Every node has 3 outgoing edges, thus it is not 3-colorable
Is Chaitin’s Graph Coloring Optimal?

- According to Chaitin’s algorithm: Every node has 3 outgoing edges, thus it is not 3-colorable.
- However, it is 3-colorable as you can see above.
- Chaitin’s algorithm is not optimal.
What if Coloring Fails?

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory

```
\begin{center}
\begin{tikzpicture}
  \node (a) at (0,0) {a};
  \node (b) at (2,0) {b};
  \node (c) at (2,2) {c};
  \node (d) at (0,2) {d};
  \node (e) at (0,1) {e};
  \node (f) at (1,1) {f};
  \draw (a) -- (b) -- (c) -- (d) -- (e) -- (f) -- (a);
\end{tikzpicture}
\end{center}
```
What if Coloring Fails?

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory
What if Coloring Fails?

- Spill the variable to memory
  - a spilled variable temporarily lives in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory

Graph:
- Nodes: f, e, c, b, d
- Edges: f <-> e, e <-> c, c <-> d, d <-> b, b <-> f
What if Coloring Fails?

- Spill the variable to memory
  - a spilled variable temporarily lives in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory
What if Coloring Fails?

- Spill the variable to memory
  - a spilled variable temporarily **lives** in memory
  - e.g. to color the previous graph using 3 colors
    - spill “f” into memory

![Graph Diagram]

- f
- e
- c
- d
On-line compilers need to generate binary code quickly
  ➢ Just-in-time compilation
  ➢ Interactive environments e.g. IDE

In these cases, it is beneficial to sacrifice code performance a bit for quicker compilation
  ➢ A faster allocation algorithm
  ➢ Not sacrificing too much in code quality

Proposed in following publication:
  ➢ Poletto, M., Sarkar, V., "Linear scan register allocation", in ACM Transactions on Programming Languages and Systems (TOPLAS), 1999
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each live interval

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
</table>

Scan order

Code
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each live interval

Diagram:
- Code layout with scan order
- Live intervals marked for allocation
- A and D may be allocated to the same register
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each live interval

Allocate greedily at each numbered point in program

- A and D may be allocated to same register
**Live Interval**: Smallest interval of code containing all live ranges in the given linear code layout

- Live range of $a = \{B1, B3\}$, $b = \{B2, B4\}$
- If code layout is “B1,B3,B2,B4”, only 1 register is enough
  - Live interval of $a = \{B1, B3\}$, $b = \{B2, B4\}$
- If code layout is “B1,B2,B3,B4”, then need 2 registers
  - Live interval of $a = \{B1, B2, B3\}$, $b = \{B2, B3, B4\}$

```
B1
 a= ...

B2
 b= ...

B3
 ...=a

B4
 ...=b
```
Linear Scan Algorithm

- Linear scan RA consists of four steps
  - S1. Order all instructions in linear fashion
    - Order affects quality of allocation but not correctness
  - S2. Calculate the set of live intervals
    - Each variable is given a live interval
  - S3. Greedily allocate register to each interval in order
    - If a register is available then allocation is possible
    - If a register is not available then an already allocated register is chosen (register spill occurs)
  - S4. Rewrite the code according to the allocation
    - CPU registers replace temporary or program variables
    - Spill code is generated
Register Allocation Time Comparison

- Usage Counts, Linear Scan, and Graph Coloring shown
- Linear Scan allocation is always faster than Graph Coloring
ILP-based Register Allocation

- Idea and steps:
  1. Convert RA problem to an ILP problem
  2. Solve ILP problem using widely known ILP solvers
  3. Map the ILP solution back to register assignment

- Goal: find “optimal” allocation
  - Chaitin graph coloring is a heuristic algorithm
  - Optimal (NP-complete) graph coloring algorithms exist, but still use heuristics for spilling
  - ILP finds optimal allocation and placement of spill code

- Complexity restricts adoption by industrial compilers
  - Optimal ILP solution is NP-hard (similar to graph coloring)
  - ILP allocation is slow → does not scale to large programs
What is Integer Linear Programming (ILP)?

- Integer Linear Programming (ILP)
  - Variables: a, b
  - Constraints:
    - \(0 \leq a \leq 10\)
    - \(0 \leq b \leq 29\)
    - \(a + b \leq 36\)
  - Goal function
    - minimize \(f(a,b) = 3a + 4b\)

- It is trivial if a and b can take real values
- It is NP hard if a and b can only take integer values
How to Convert Register Allocation to ILP?

An example

(9) ...
(10) ... = b + a ;
(11) ...

- Want to know to which register b should be allocated i.e. load Rx, addr(b)

Convert to an ILP problem
- Assume there are four free registers R1, R2, R3, R4

S1: Define the variables in ILP

$V_{var(location)}^{R_i}$ — Whether var at location is allocated to Ri

$V_{b(10)}^{R_1}$, $V_{b(10)}^{R_2}$, $V_{b(10)}^{R_3}$, $V_{b(10)}^{R_4}$

Value of 0 — not allocate to that register at the place
Value of 1 — is allocated to that register at the place
Converting Register Allocation to ILP

**S2:** Define constraints. E.g. for code (10) ... = b + a,

- A register can hold at most one variable per place
  \[ V_{R1}^{b(10)} + V_{R1}^{a(10)} \leq 1, \quad V_{R2}^{b(10)} + V_{R2}^{a(10)} \leq 1, \ldots \]
- A variable is allocated to exactly one register per place
  \[ V_{R1}^{b(10)} + V_{R2}^{b(10)} + V_{R3}^{b(10)} + V_{R4}^{b(10)} = 1 \]
  \[ V_{R1}^{a(10)} + V_{R2}^{a(10)} + V_{R3}^{a(10)} + V_{R4}^{a(10)} = 1 \]
  and many more ...

**S3:** Define goal function

- To minimize cost of memory operations for spilling:
  \[ f_{cost} = \sum V_{v(loc)}^{stack} \times U_{v(loc)} \times \text{exec\_count}(loc) \times LOAD_{cost} + \ldots \]
  \[ V_{v(loc)}^{stack} \]: Whether \( v \) at \( loc \) is allocated to stack (spilled)
  \[ U_{v(loc)} \]: Whether variable \( v \) is used right after \( loc \)
  \[ \text{exec\_count}(loc) \]: Expected runtime execution count of \( loc \)
  \[ LOAD_{cost} \]: Cost of load instruction in given machine
Conclusion

- Good Register Allocation is crucial to code quality
  - Accesses to memory are costly, even with caches
  - Even with few program variables, intermediate values introduce many more temporary variables, adding to register pressure

- Different algorithms make different trade-offs between allocation time and code quality
Instruction Selection
Instruction Selection

- Instruction selection is the task to select appropriate machine instructions to implement the operations in the intermediate representation (IR).
  - Very important for CISC machines, and machines with special purpose instructions (MMX)
  - X86, ARM, DSP, ...

- There are many semantically equivalent instruction sequences
  - How to find the “minimal cost” sequence?
## Some Instruction Patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( d_i \leftarrow d_j + d_k )</td>
<td>( d + )</td>
</tr>
<tr>
<td>MUL</td>
<td>( d_i \leftarrow d_j \times d_k )</td>
<td>( d \times d )</td>
</tr>
<tr>
<td>SUB</td>
<td>( d_i \leftarrow d_j - d_k )</td>
<td>( d - )</td>
</tr>
<tr>
<td>DIV</td>
<td>( d_i \leftarrow d_j / d_k )</td>
<td>( d \div d )</td>
</tr>
<tr>
<td>ADDI</td>
<td>( d_i \leftarrow d_j + \text{const} )</td>
<td>( d + \text{CONST} )</td>
</tr>
<tr>
<td>SUBI</td>
<td>( d_i \leftarrow d_j - \text{const} )</td>
<td>( d \text{CONST} )</td>
</tr>
<tr>
<td>MOVEA</td>
<td>( d_j \leftarrow a_i )</td>
<td>( d a )</td>
</tr>
<tr>
<td>MOVED</td>
<td>( a_j \leftarrow d_i )</td>
<td>( a d )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( d_i \leftarrow M[a_j + \text{const}] )</td>
<td>( d \text{MEM} \text{MEM} \text{CONST} )</td>
</tr>
<tr>
<td>STORE</td>
<td>( M[a_j + \text{const}] \leftarrow d_i )</td>
<td>( d \text{MEM} \text{MEM} \text{CONST} )</td>
</tr>
<tr>
<td>MOVEM</td>
<td>( M[a_j] \leftarrow M[a_i] )</td>
<td>( d \text{MEM} \text{MEM} )</td>
</tr>
</tbody>
</table>
A Parse Tree to be Tiled

MOVE

MEM [MEM [MEM [FP const] + temp_i] * const] + MEM [FP const]
A Parse Tree to be Tiled

```
MOVE
  | MEM  | MEM |
  | +    | +   |
  | *    |     |
  |      | FP  |
  |      | const |
  | MEM |
  | temp_i |
  | 2: load R, M[fp+a] |
  | const |
  | const |
```
A Parse Tree to be Tiled

```
MOVE
  MEM       MEM
  +
  +
  *
FP   const

1: const
2: load R, M[fp+a]
3: temp;i
```
A Parse Tree to be Tiled

1: const
2: load R, M[fp+a]
3: temp_
4: addi R2, R0+4

MOVE

MEM
  | +
  | |
MEM

MEM
  | *
  | |
FP
  |
const
A Parse Tree to be Tiled

1: const
2: load R, M[fp+a]
3: temp
4: addi R2, R0+4
5: mul R2, Ri × R2
A Parse Tree to be Tiled

1: const

2: load R, M[fp+a]

3: const

4: addi R2, R0+4

5: mul R2, Ri × R2

6: add R1, R1+R2

MOVE

MEM

MEM

FP

const

temp

const
A Parse Tree to be Tiled

1: M
2: load R, M[fp+a]
3: temp;
4: addi R2, R0+4
5: mul R2, Ri × R2
6: add R1, R1+R2
7: +

MEM MOVE MEM

FP const

const
A Parse Tree to be Tiled

```
MOVE

MEM

MEM

6: add R1, R1+R2

5: mul R2, Ri × R2

8: addi R2, fp+x

2: load R, M[fp+a]

temp;

3:

4: addi R2, R0+4

1:

FP

const

FP

const

```
A Parse Tree to be Tiled

1: const
2: load R, M[fp+a]
3: temp
4: addi R2, R0+4
5: mul R2, Ri × R2
6: add R1, R1+R2
7: temp
8: addi R2, fp+x
9: MoveMem M[R1], M[R2]

MEM

MOVE

MEM

FP

const

FP const

const

const
A Parse Tree to be Tiled

1: const
2: load R, M[fp+a]
3: temp
4: addi R2, R0+4
5: mul R2, Ri \times R2
6: add R1, R1+R2
7: const

MOVE

MEM

MEM

MEM

FP

const
A Parse Tree to be Tiled

1: const

2: load R, M[fp+a]

3: temp

4: addi R2, R0+4

5: mul R2, Ri \times R2

6: add R1, R1+R2

7: 

8: load R2, M[fp+x]

MOVE

MEM

MEM

MEM

FP

const

temp

const
A Parse Tree to be Tiled

1: \( \text{const} \)

2: load \( R, M[fp+a] \)

3: \( \text{addi} R2, R0+4 \)

4: \( \text{mul} R2, \text{const} \)

5: \( \text{mul} R2, \text{Ri} \times \text{R2} \)

6: \( \text{add} R1, R1+R2 \)

7: \( \text{load} R2, M[fp+x] \)

8: load \( R2, M[fp+x] \)

9: store \( M[R1+0], R2 \)

\[ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \]
The END !