Translating C to Hardware

As mentioned, control and data flow analysis can be helpful in translating C into hardware.

Translating data structures and pointers from C to hardware can get tricky.

For the purpose of this course, we restrict our analysis as follows:

• Only scalar C code is used (no pointers, arrays or other data structures)
• We assume each C statement executes in a single clock cycle

We first create the CFG and DFG for the C program.

The control edges translate to signals that control *datapath operations*.

The data edges define the *interconnection* of the datapath components.
Translating C to Hardware: Data Path Design

Data Path Design: The following process can be used to create the datapath

- Each variable in the C program is translated into a register and a multiplexer
  The multiplexer is used to allow the register to be written to from any one of multiple sources

  The select inputs of the multiplexor are connected to the controller

  The default multiplexer setting is to preserve the register contents, which means the output of the register is fed back to the input

- For each node (operation) in the DFG, create the corresponding combinational circuit from the C expression
  For example, the expression $b - a$ is used for the operation $a = b - a$; which is implemented using a subtractor

  Note that conditional expressions also generate datapath elements whose outputs define flags used by the hardware controller
Translating C to Hardware: Data Path Design

- The datapath and the registers are connected consistent with the DFG
  **Assignments** connect combinational circuit outputs to register inputs, while the **data edges** connect register outputs to combinational circuit inputs

System I/O is connected to datapath inputs and register outputs resp.

Let’s convert the GCD program to a hardware implementation
- The variables \(a\) and \(b\) are assigned to registers
- The conditional expressions for the *if* and *while* stmts require an equality- and greater-than comparator circuit
- The subtractions \(b - a\) and \(a - b\) are implemented using subtractors

The connectivity of the components is defined by the data edges of the DFG
The resulting datapath has **two data inputs** \((\text{in}_a\) and \(\text{in}_b)\), and one data output \((\text{out}_a)\)

The circuit needs **two control variables**, \(\text{upd}_a\) and \(\text{upd}_b\) (outputs of controller) and it produces two **flags**, \(\text{flag}_\text{while}\) and \(\text{flag}_\text{if}\) (inputs to controller)
Translating C to Hardware: Data Path Design

```c
1: int gcd(int a, int b) {
2:     while (a != b) {
3:         if (a > b)
4:             a = a - b;
5:     } else
6:         b = b - a;
7: } return a;
8: }
```

The directed edges in the DFG correspond to the connections in the schematic. Schematic representations of a circuit are **low-level representations** with lots of detail (similar to assembly code in programming languages).

We will learn how to create HLS and behavioral VHDL descriptions that synthesize to schematics similar to the one shown here.
Translating C to Hardware: Control Path Design

Controller Design: The design of the controller can be derived directly from the CFG and translated into a finite state machines (FSM)

A FSM is typically depicted using bubbles and directed edges, similar to CFGs.

Unlike CFGs, the edges in FSMs are labeled with condition/command tuples. The ‘_’ in _/run1 means don’t care, i.e., the transition is unconditional.

The command component is given by the symbol run1, which is used to control the data path and therefore represents an output of the FSM.
Translating C to Hardware: Control Path Design

Similarly, `flag_while/_` means the transition out of the current state is **conditional** on `flag_while`

And the command ` '_' ` is a **hold** operation, which means maintain the current state of the datapath and registers

The command set for this FSM includes ` _`, `run1`, `run4`, `run5`

These symbols will be used to create the `upd_a` and `upd_b` data path control signals
Translating C to Hardware: Control Path Design

Hardware implementation of the GCD controller with data path

The commands run1, run4 and run5 are decoded into upd_a and upd_b.

One each clock cycle, the controller generates a new command based on the current state and the current values of flag_while and flag_if

The combination of the data path and controller is referred to as a finite state machine with datapath (FSMD)
**Translating C to Hardware: Example**

FSMDs are central to custom hardware design, so we discuss them further in the next chapter (and throughout this course)

The table shows an example execution, where each row of the table corresponds to one clock cycle:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>a</th>
<th>b</th>
<th>State flag_if</th>
<th>flag_while</th>
<th>Next State</th>
<th>upd_a</th>
<th>upd_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>_</td>
<td>_</td>
<td>s1</td>
<td>_</td>
<td>s2</td>
<td>in_a</td>
<td>in_b</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
<td>s2</td>
<td>1</td>
<td>s3</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>s3</td>
<td>1</td>
<td>s4</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>s4</td>
<td>1</td>
<td>s2</td>
<td>a-b</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>s2</td>
<td>0</td>
<td>s3</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>s3</td>
<td>0</td>
<td>s5</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>4</td>
<td>s5</td>
<td>0</td>
<td>s2</td>
<td>a</td>
<td>b-a</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>s2</td>
<td>0</td>
<td>s6</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>s6</td>
<td>_</td>
<td>s6</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

Note that this solution is sub-optimal, in particular:

- The resulting implementation **limits parallelism** -- it executes a single C statement per clock cycle and does **not share** datapath operators

  For example, only one subtractor is needed in the implementation because only one is ever used in any given clock cycle
Single-Assignment Programs

Converting into hardware with one C-stmt/clock is not very efficient

This one cycle-per-statement is similar to what microprocessors do when they execute a program

A more lofty goal is to devise a translation strategy that allows the execution of multiple C stmts/clock

But our original variable-to-register mapping strategy creates a performance bottleneck

This is true because only one storage location exists for each variable and therefore, sequential updates to it will each require one clock cycle

We fix this problem by converting the C code to a single-assignment program

This is done by creating new variables for each sequential assignment stmt
Single-Assignment Programs

Consider a simple example:

\[
\begin{align*}
a &= a + 1; \\
a &= a \times 3; \\
a &= a - 2;
\end{align*}
\]

Our previous strategy requires 3 clock cycles to execute these statements.

Let’s re-write this as:

\[
\begin{align*}
a_2 &= a_1 + 1; \\
a_3 &= a_2 \times 3; \\
a_4 &= a_3 - 2;
\end{align*}
\]

This code allows \(a_2\) and \(a_3\) to be mapped to wires and \(a_4\) to a register, reducing the clock cycle count to 1.
Single-Assignment Programs

Note: care must be taken that all assignments are taken into account, which might be difficult to determine

\[ a = 0; \]
\[ \text{for } (i = 1; i < 6; i++) \]
\[ a = a + i; \]

After conversion to single-assignment, it remains unclear what version of \( a \) should be read inside of the loop

\[ a_1 = 0; \]
\[ \text{for } (i = 1; i < 6; i++) \]
\[ a_2 = a + i; // \text{which version of } a \text{ to read} \]

The answer is that both \( a_1 \) and \( a_2 \) are needed and it depends on the iteration, i.e., \( a_1 \) is needed on the first iteration and \( a_2 \) on subsequent iterations
Single-Assignment Programs

The solution is to introduce a new `merge` variable that selects from the two versions that are available.

```plaintext
a1 = 0;
for (i=0; i<5; i++) {
    a3 = merge(a1, a2);    // merge two instances.
    a2 = a3 + 1;
}
```

In a hardware implementation, the `merge` operation is mapped into a `multiplexer`, with the `selection signal` derived from the test of `(i == 0)`

Using these transformations, we can reformulate any program into single assignment form
Single-Assignment Programs
Consider the GCD program

```c
int gcd (int a, int b) {
    while (a != b)
    {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```
Single-Assignment Programs

The equivalent single-assignment form:

```c
int gcd (int a1, int b1)
{
    while ((a3 = merge(a1, a2)) != (b3 = merge(b1, b2)))
    {
        if (a3 > b3)
            a2 = a3 - b3;
        else
            b2 = b3 - a3;
    }
    return a2;
}
```
Single-Assignment Programs

The implementation of this single-assignment version might look like:

merge operations implemented as multiplexers

Here, $a2$ and $b2$ are mapped into registers while the other variables are replaced with wires

This type of manipulation allows multiple C statements to be executed per clock