Control and Data Edges

C programs (and pseudo-code) are often used as prototypes because they represent **high-level descriptions** of system behavior

However, C is sequential and cannot be directly mapped into parallel hardware

Nonetheless, codesigners must develop skills to carry out this task (it is listed as one of our course objectives)

A solid understanding of C program structure and the relationships that exist between C operations is foundational to this process

In this lecture, we consider two fundamental relationships between C operations

- **Data edge**: is a relationship between operations where data produced by one operation is consumed by another
- **Control edge**: is a relationship between operations that relates to the order in which the operations are performed

Control and Data Edges

Consider the following example that returns the max of *a* or *b*

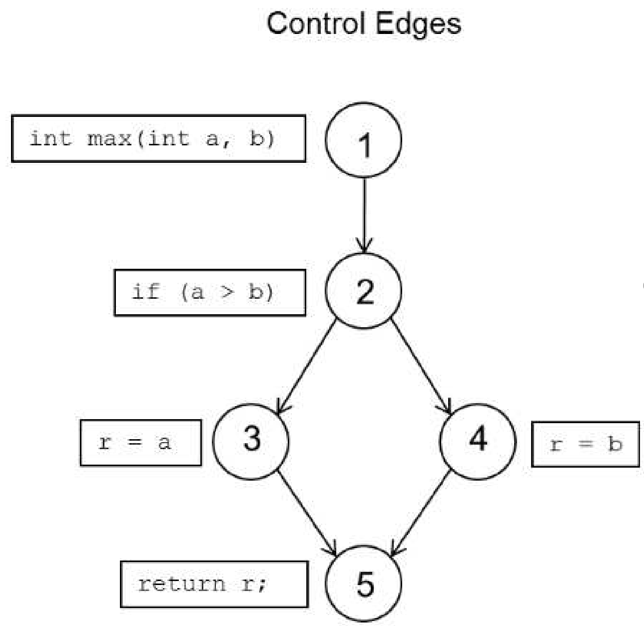
```
int max(int a, b) // operation 1 - enter the function
    {
    int r;
    if (a > b )      // operation 2 - if-then-else
       r = a;        // operation 3
    else
       r = b;        // operation 4
    return r;        // operation 5 - return max
    }
```

As you can see, our analysis treats each of the C statements as individual operations

To find the control edges in this program, we need to identify all possible paths through this program

Control and Data Edges

For example, operation 2 will always execute *after* operation 1



Control Flow Graph (CFG)

We can use a **control flow graph** (CFG) to capture this relationship, by adding a directed edge between these operations (which are represented as bubbles)

The *if-then-else* operation includes two out-going edges to represent each of the two execution paths

Control and Data Edges

The **data flow graph** (DFG) is constructed by analyzing the data production (writes) and consumption (reads) patterns for each of the variables

```
int max(int a, b) { // operation 1 - produce a, b
  int r;
  if (a > b )    // operation 2 - consume a, b
    r = a;       // operation 3 - consume a and (a>b),
                 //                produce r
  else
    r = b;       // operation 4 - consume b and (a>b),
                 //                produce r
  return r;      // operation 5 - consume r
}
```

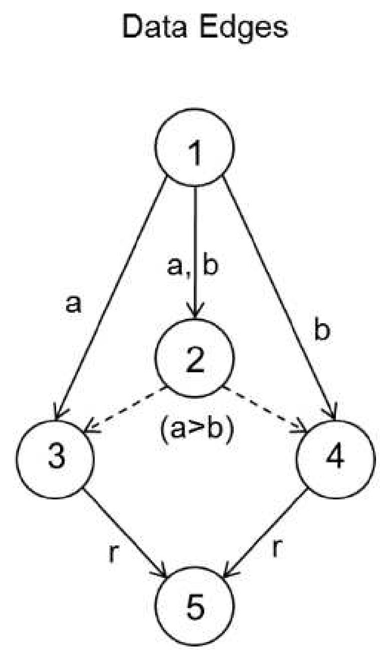
Data edges are added between operations which write and then read a variable

For example, operation 1 defines (writes) the values of *a* and *b*

Variable *a* is read by operation 2 and 3 while *b* is read by operation 2 and 4

This produces data edges from 1 to 2 for *a* and *b*, and to 3 for *a* and 4 for *b*

Control and Data Edges



Data Flow Graph (DFG)

Control statements in C also generate data edges

For example, the *if-then-else* statement evaluates a *flag* ($a > b$), which reads a and b

The boolean *flag carries* the value of ($a > b$) from operation 2 to operations 3 and 4

Note that unlike CFGs, edges in DFGs are labeled with a specific variable

Implementation Issues

CFGs and DFGs capture the behavior of the C program graphically

This leads naturally to the following question:

What are the important parts of a C program that MUST be preserved in ***any*** *implementation of that program?*

- Data edges reflect requirements on the flow of information

  Important note: If you change the flow of data, **you change the meaning of the algorithm**

- Control edges, on the other hand, provide a nice mechanism to break down the algorithm into a sequence of operations (a recipe)

  They are **not** fundamental to preserving correct functional behavior in an implementation

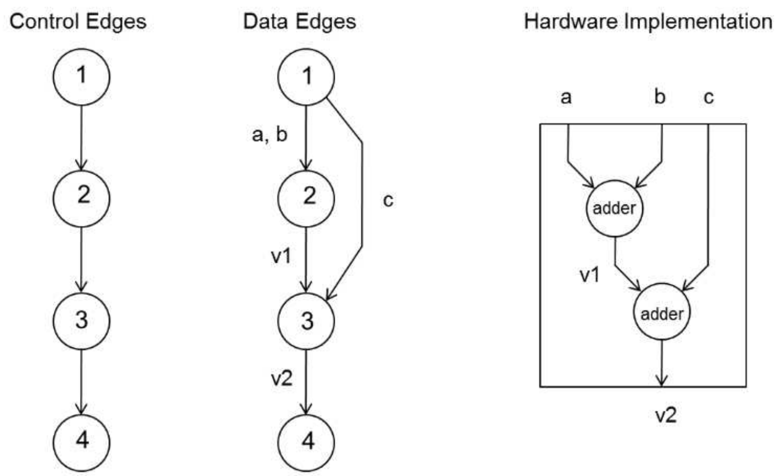
It follows then that **data edges** MUST be preserved while **control edges** can be removed and/or manipulated

Implementation Issues

Parallelism in the underlying architecture can be leveraged to remove control edges, e.g., superscalar processors can execute instructions *out-of-order*

On the other hand, parallel architectures MUST always preserve data dependencies otherwise, the results will be erroneous

```
int sum(int a, b, c) {          // operation 1
   int v1;
   v1 = a + b;                  // operation 2
   v2 = v1 + c;                 // operation 3
   return v2; }                 // operation 4
```



A fully parallel hardware implementation of this program can in fact carry out both additions in a *single clock cycle*

The sequential order specified by the CFG is eliminated in the hardware implementation

Construction of the Control Flow Graph

Let's define a systematic method to convert a C program to a CFG assuming:

- Each **node** in the graph represents a single operation (or C statement)
- Each **edge** of the graph represents an execution order for the two operations connected by that edge

Since C executes sequentially, this conversion is straightforward in most cases

The only exception occurs when multiple control edges originate from a single operation

Consider the *for loop* in C

```
for (i = 0; i < 20; i++) {
    // body of the loop
}
```

This statement includes four distinct parts:

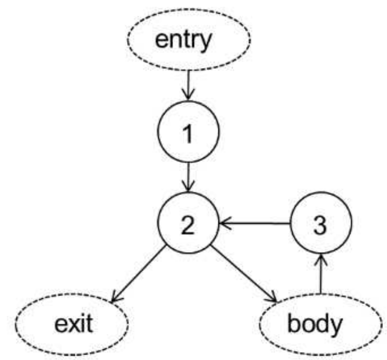
- loop initialization
- loop condition
- loop-counter increment operation
- body of the loop

Construction of the Control Flow Graph

The *for loop* introduces three nodes to the CFG

for loop contributes multiple operations

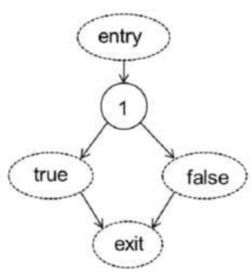
```
      (1)         (2)       (3)
for (i=0; i < 20; i++) {
   // body of the loop
}
```



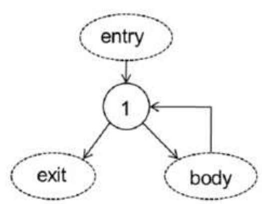
Dashed components, *entry*, *exit* and *body*, are other CFGs of the C program which have ***single-entry*** and ***single-exit*** points

The *do-while* loop and the *while-do* loop are similar iterative structures

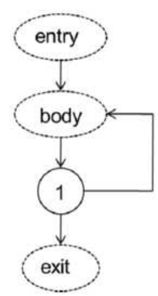
```
   (1)
if(a < b) {
   // true branch
} else {
   // false branch
}
```




```
        (1)
while (a < b) {
   // loop body
}
```



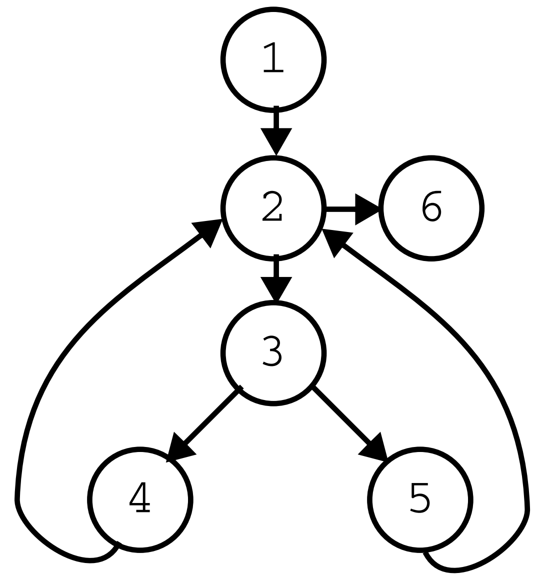

```
do {
   // loop body
} while (a<b)
          (1)
```



Construction of the Control Flow Graph

Consider the CFG for the GCD algorithm.

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



A **control path** is defined as a sequence of control edges that traverse the CFG

For example, each *non-terminating* iteration of the while loop will follow the path 2->3->4->2 or else 2->3->5->2

Control paths are useful in constructing the DFG

Construction of the Data Flow Graph

Let's also define a systematic method to convert a C program to a DFG assuming

- Each **node** in the graph represents a single operation (or C statement)
- Each **edge** of the graph represents a data dependency

Note that the CFG and the DFG will contain the **same set of nodes** -- only the edges will be different

While it is possible to derive the DFG directly from a C program, it is easier to create the CFG **first** and use it to derive the DFG

The method involves tracing control paths in the CFG while simultaneously identifying corresponding read and write operations of the variables

Our analysis focuses on C programs that do NOT have *arrays* or *pointers*

Text includes discussion and examples on how to handle these more complex data structures

Construction of the Data Flow Graph

Ad-hoc method:

- Start at the node where a variable is read (which is referred to as a *read-node*)
- Identify the CFG nodes that assign to that variable (referred to as *write-nodes*)
- Introduce a data edge between a read and write node under the condition that the *control path* does **NOT** pass through another *write-node* for that variable
- Repeat for all read nodes

This procedure identifies all data edges related to *assignment statements*, but **not** those originating from *conditional expressions* in control flow statements

However, these data edges are easy to find

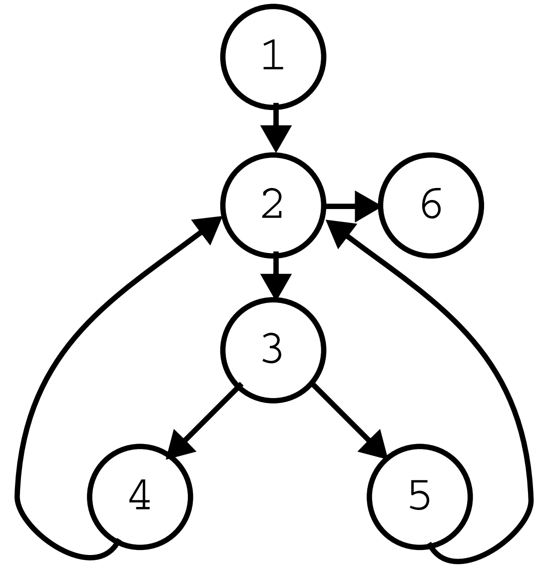
They originate from the condition evaluation and **affect all** the operations whose execution depends on that condition

Let's derive the DFG of the GCD program

We first pick a node where a variable is read

Construction of the Data Flow Graph

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



Consider stmt 5:

There are two variable-reads in this statement, one for *a* and one for *b*

Consider *b* first

Find all nodes that reference *b* by **tracing backwards** through predecessors of node 5 in the CFG -- this produces the ordered sequence 3, 2, 1, 4, and 5

Both nodes 1 and 5 write *b* and there is a *direct path* from 1 to 5 (e.g. 1, 2, 3, 5), and from 5 to 5 (e.g. 5, 2, 3, 5)

Therefore, we need to add data edges for *b* from 1 to 5 and from 5 to 5

Construction of the Data Flow Graph

A similar process can be carried out for variable-read of *a* in node 5

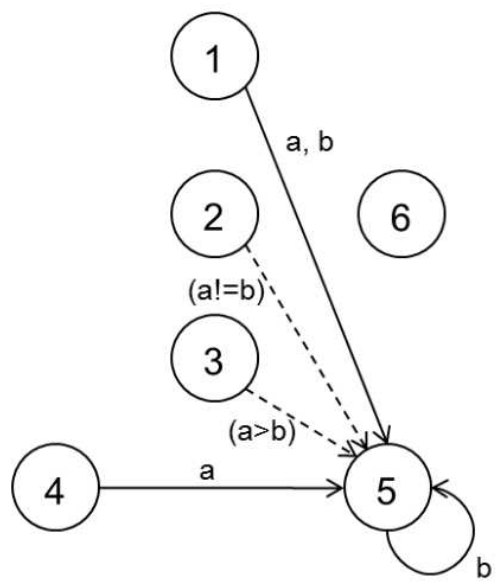
Nodes 1 and 4 write into *a* and there is a direct control path from 1 to 5 and from 4 to 5

Hence, data edges are added for *a* from 1 to 5 and from 4 to 5

To complete the set of data edges into node 5, we also need to identify all **conditional expressions** that affect the outcome of node 5

From the CFG, node 5 depends on the condition evaluated in node 3 ($a > b$) AND the condition evaluated in node 2 (a != b)

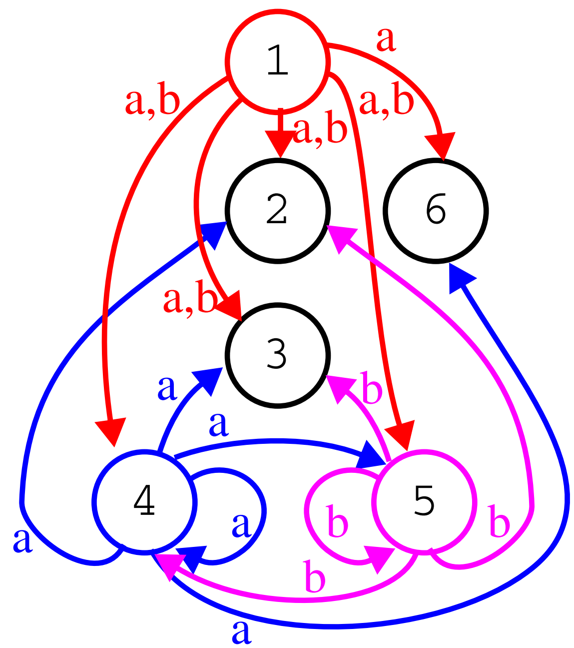
Partial DFG for node 5



Construction of the Data Flow Graph

The final DFG for all nodes and all *variable-reads* for GCD is shown below.

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



Note: this DFG leaves out the data edges originating from conditional expressions

Being able to abstract a complex C program to a DFG is essential for codesign