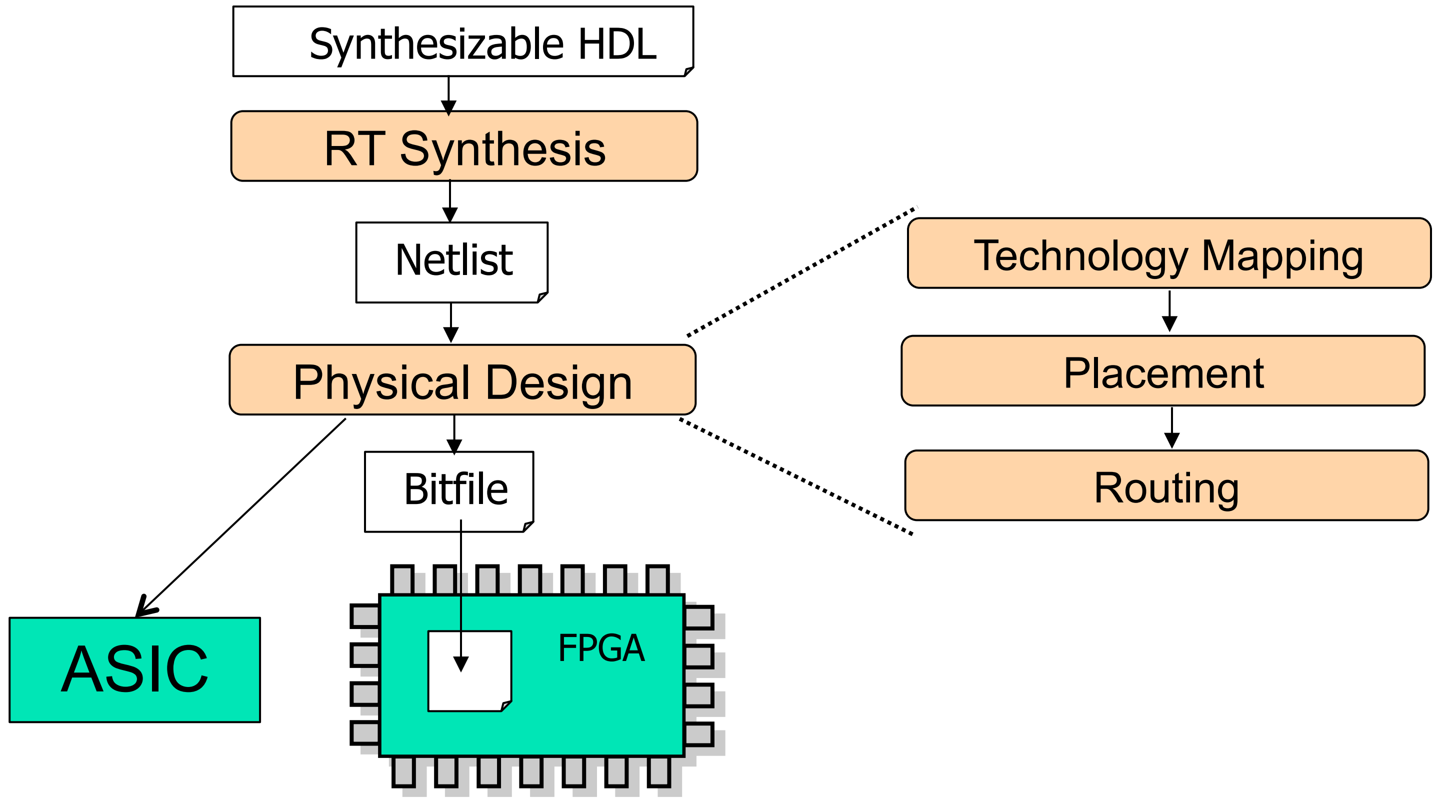
High-Level Synthesis

Creating Custom Circuits from High-Level Code

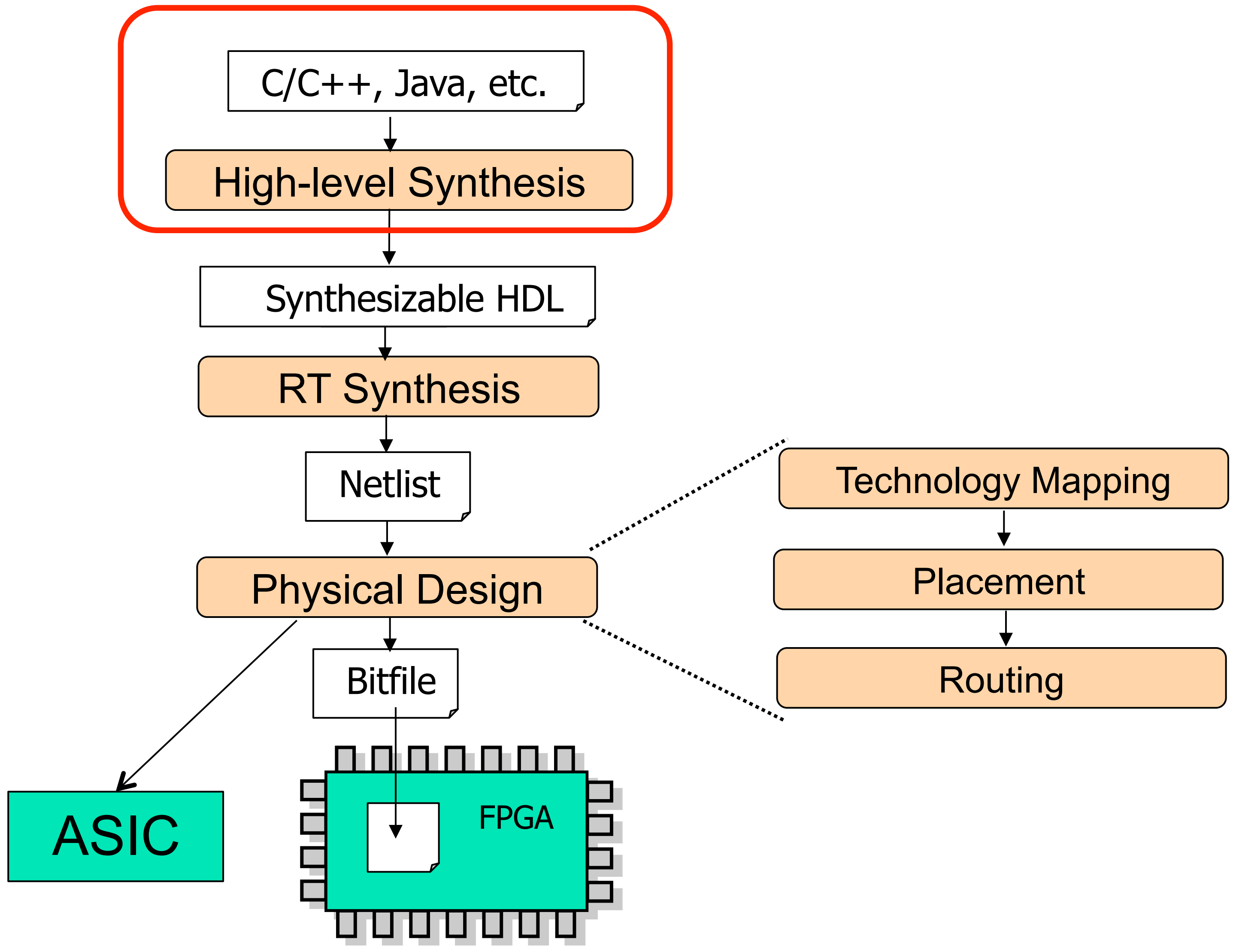
Hao Zheng
Comp Sci & Eng
University of South Florida

Existing Design Flow

- Register-transfer (RT) synthesis
  - Specify RT structure (muxes, registers, etc)
  - Allows precise specification
  - But, time consuming, difficult, error prone



Future Design Flow



Overview

- Input:
  - High-level languages (e.g., C)
  - Behavioral hardware description languages (e.g., VHDL)
  - State diagrams / logic networks
- Tools:
  - Parser
  - Library of modules
- Constraints:
  - Area constraints (e.g., # modules of a certain type)
  - Delay constraints (e.g., set of operations should finish in λ clock cycles)
- Output:
  - Operation scheduling (time) and binding (resource)
  - Control generation and detailed interconnections

High-level Synthesis - Benefits

- Ratio of C to VHDL developers (10000:1 ?)
- Easier to specify complex designs
- Technology/architecture independent designs
- Manual HW design potentially slower
  - Similar to assembly code era
  - Programmers could always beat compiler
  - But, no longer the case
- Ease of HW/SW partitioning
  - enhance overall system efficiency
- More efficient verification and validation
  - Easier to V & V of high-level code

High-level Synthesis

- More challenging than SW compilation
  - Compilation maps behavior into assembly instructions
  - Architecture is known to compiler
- HLS creates a custom architecture to execute specified behavior
  - Huge hardware exploration space
  - Best solution may include microprocessors
  - Ideally, should handle any high-level code
    - + But, not all code appropriate for hardware

High-level Synthesis: An Example

- First, consider how to manually convert high-level code into circuit

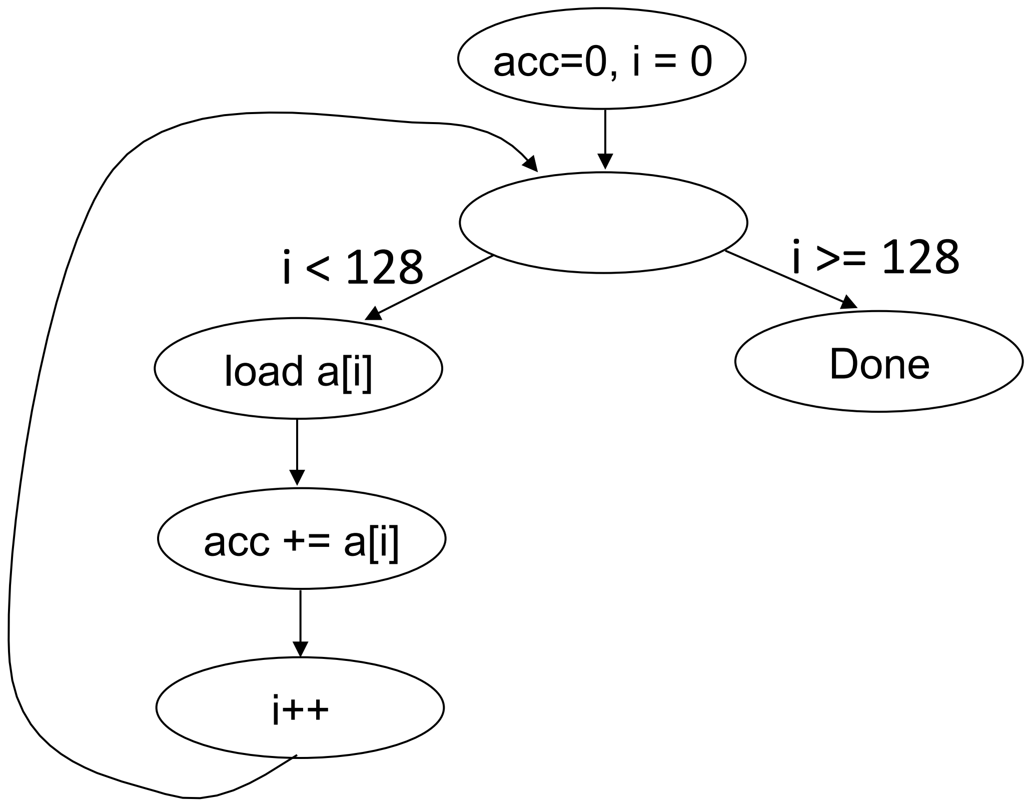
```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

- Steps
  1) Build FSM for controller
  2) Build datapath based on FSM

A Manual Example

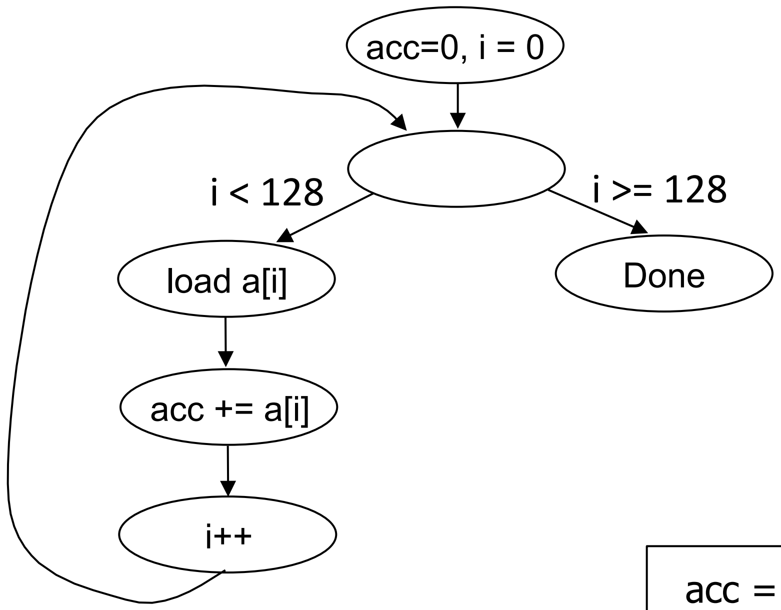
- Build a FSM (controller)
  - Decompose code into states

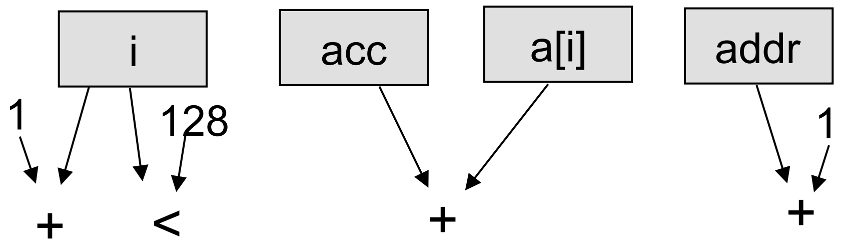
```
acc = 0;
for (i=0; i < 128; i++)
  acc += a[i];
```



A Manual Example

- Build a datapath
  - Allocate resources for each state

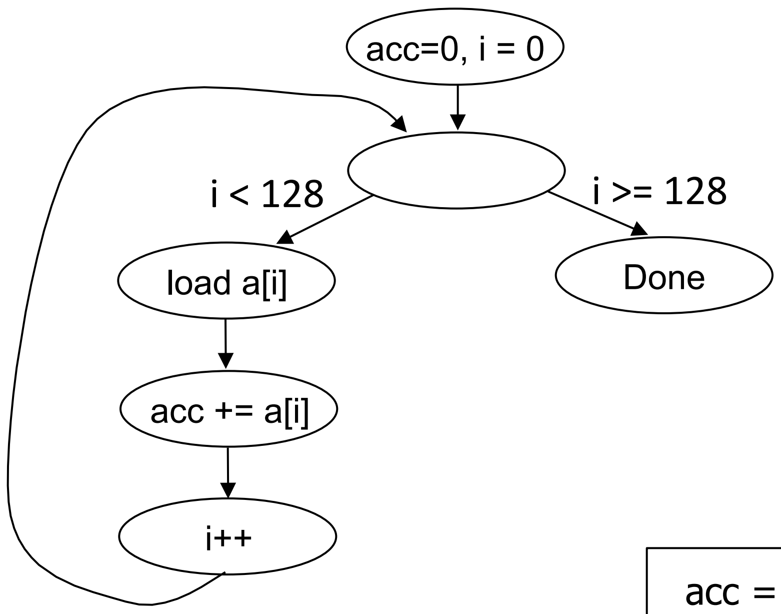


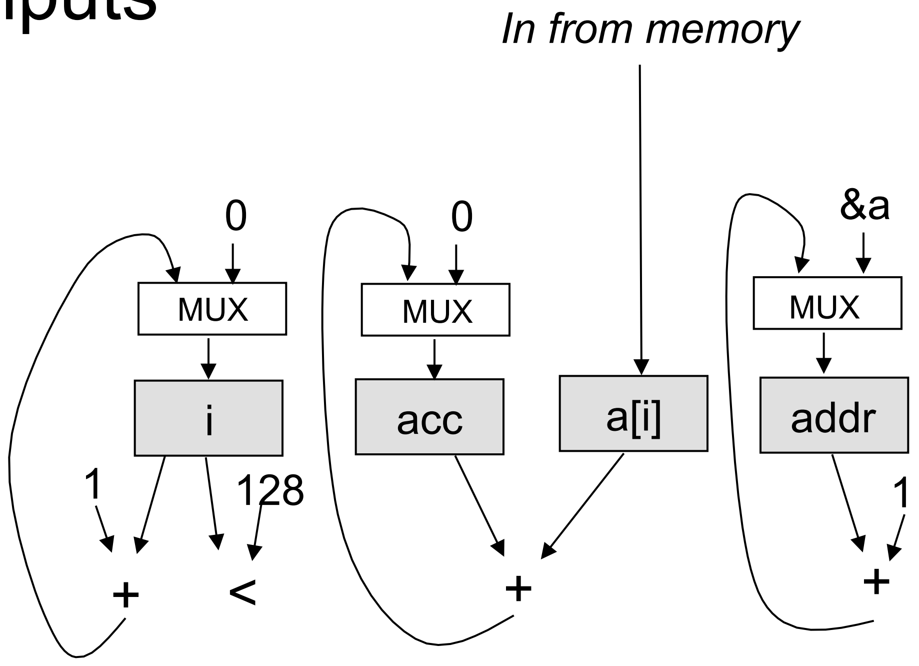


```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

A Manual Example

- Build a datapath
  - Determine register inputs

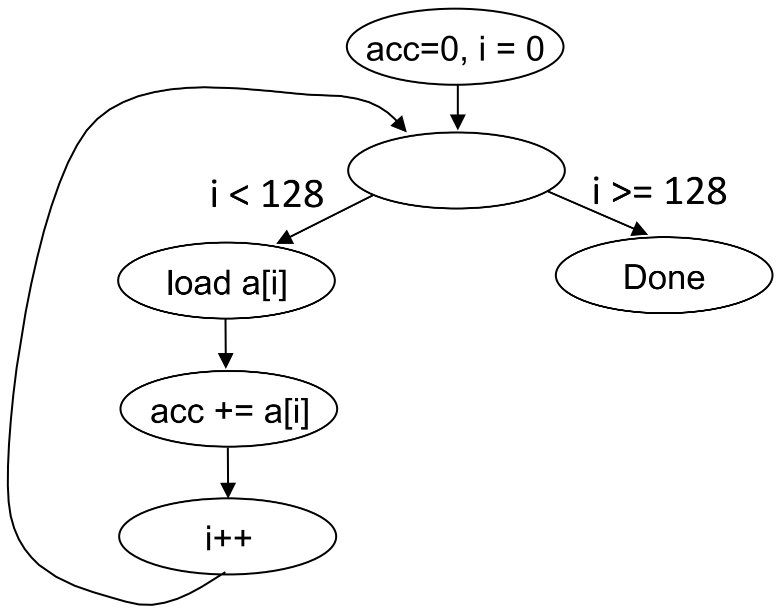


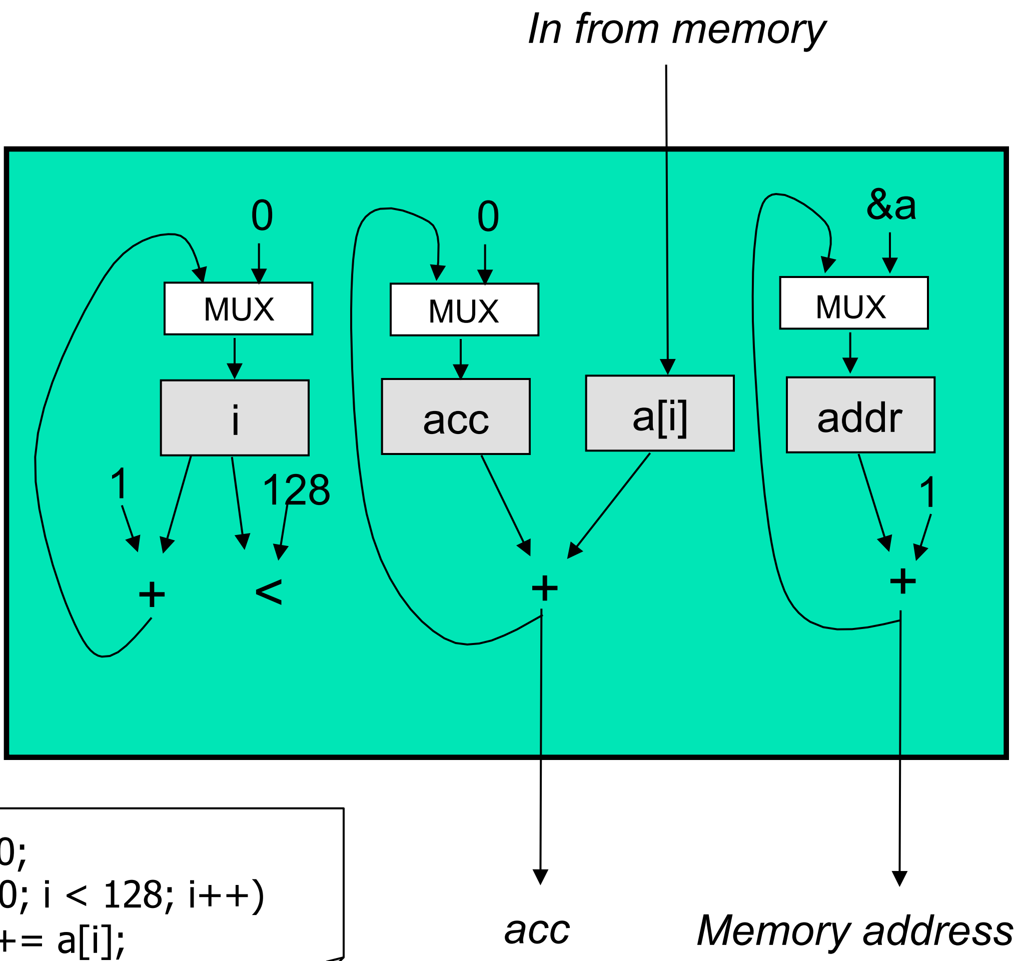


```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

A Manual Example

- Build a datapath
  - Add outputs

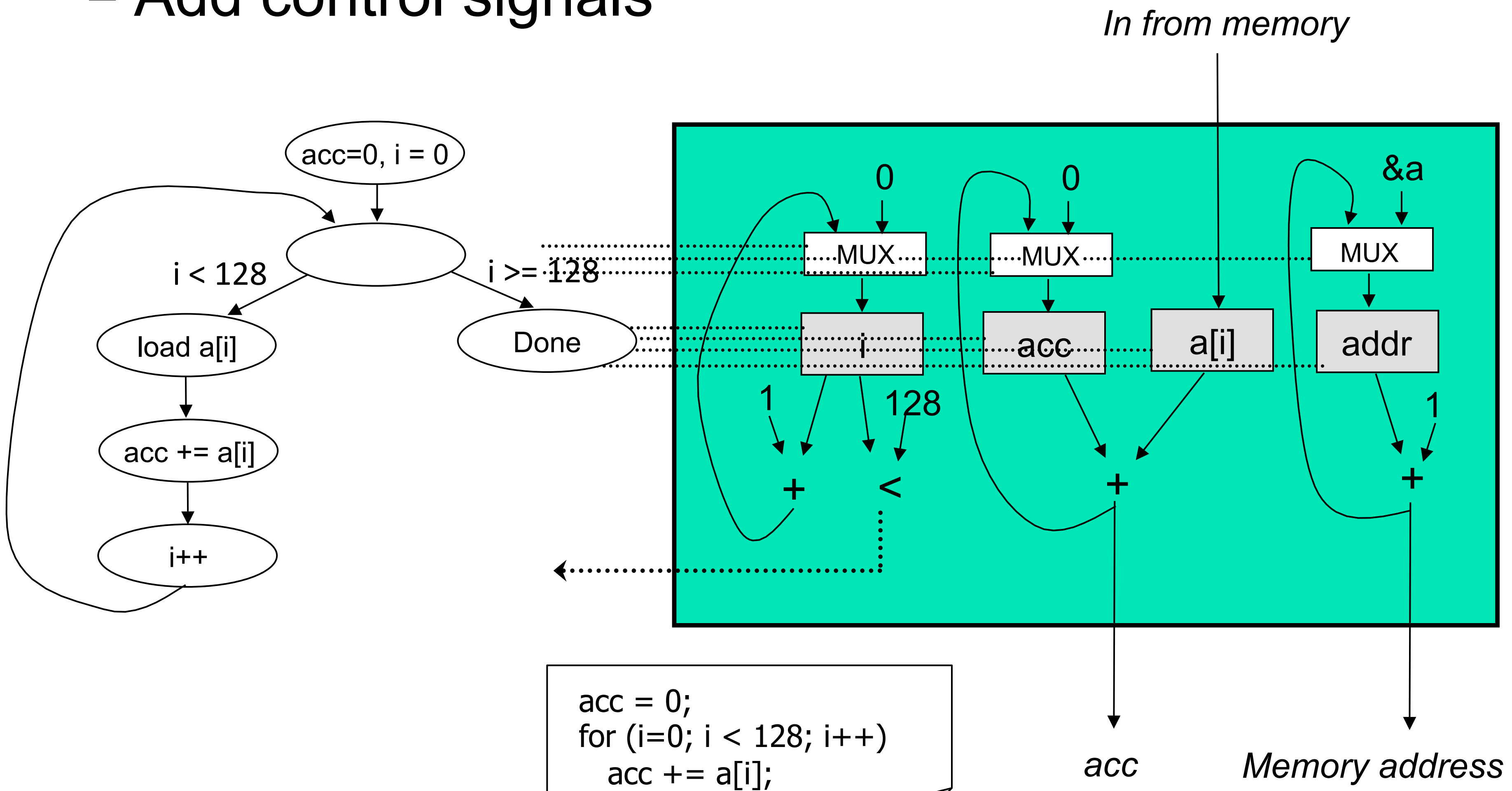




```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

A Manual Example

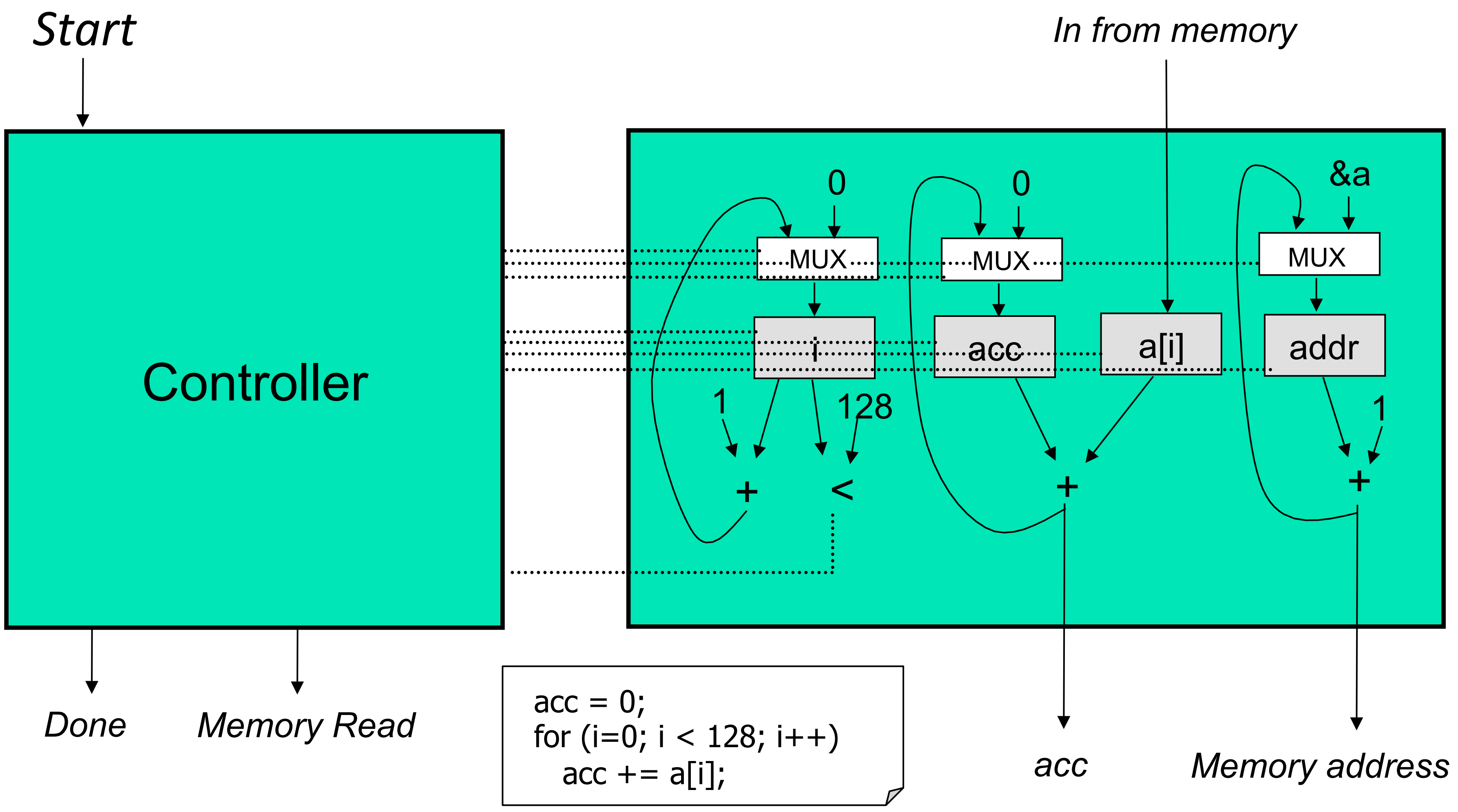
- Build a datapath
  - Add control signals



```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

A Manual Example

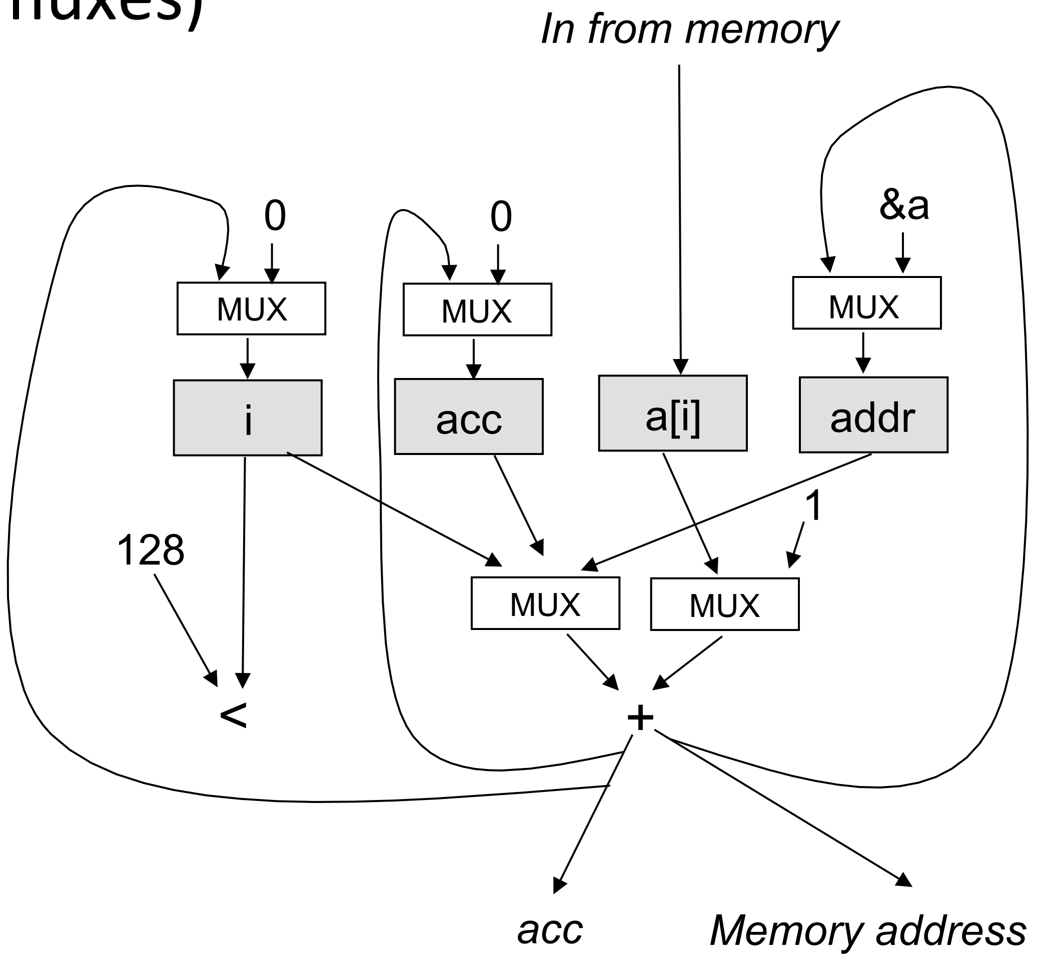
- Combine controller + datapath



```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```

A Manual Example - Optimization

- Alternatives
  - Use one adder (plus muxes)



A Manual Example – Summary

- Comparison with high-level synthesis
  - Determining when to perform each operation

    => Scheduling
  - Allocating resource for each operation

    => Resource allocation
  - Mapping operations to allocated resources

    => Binding

Another Example: Try it at home

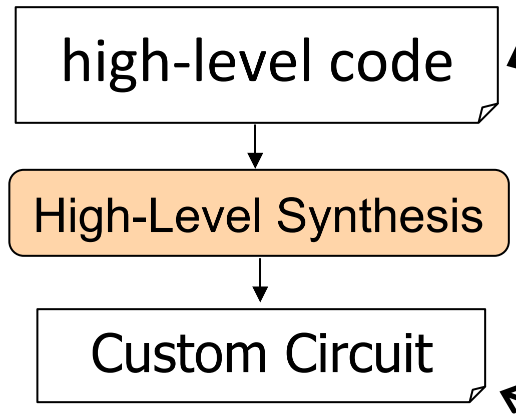
- Your turn

```
x=0;
for (i=0; i < 100; i++) {
   if (a[i] > 0)
      x ++;
   else
      x --;

   a[i] = x;
}
//output x
```

1) Build FSM (do not perform if conversion)
2) Build datapath based on FSM

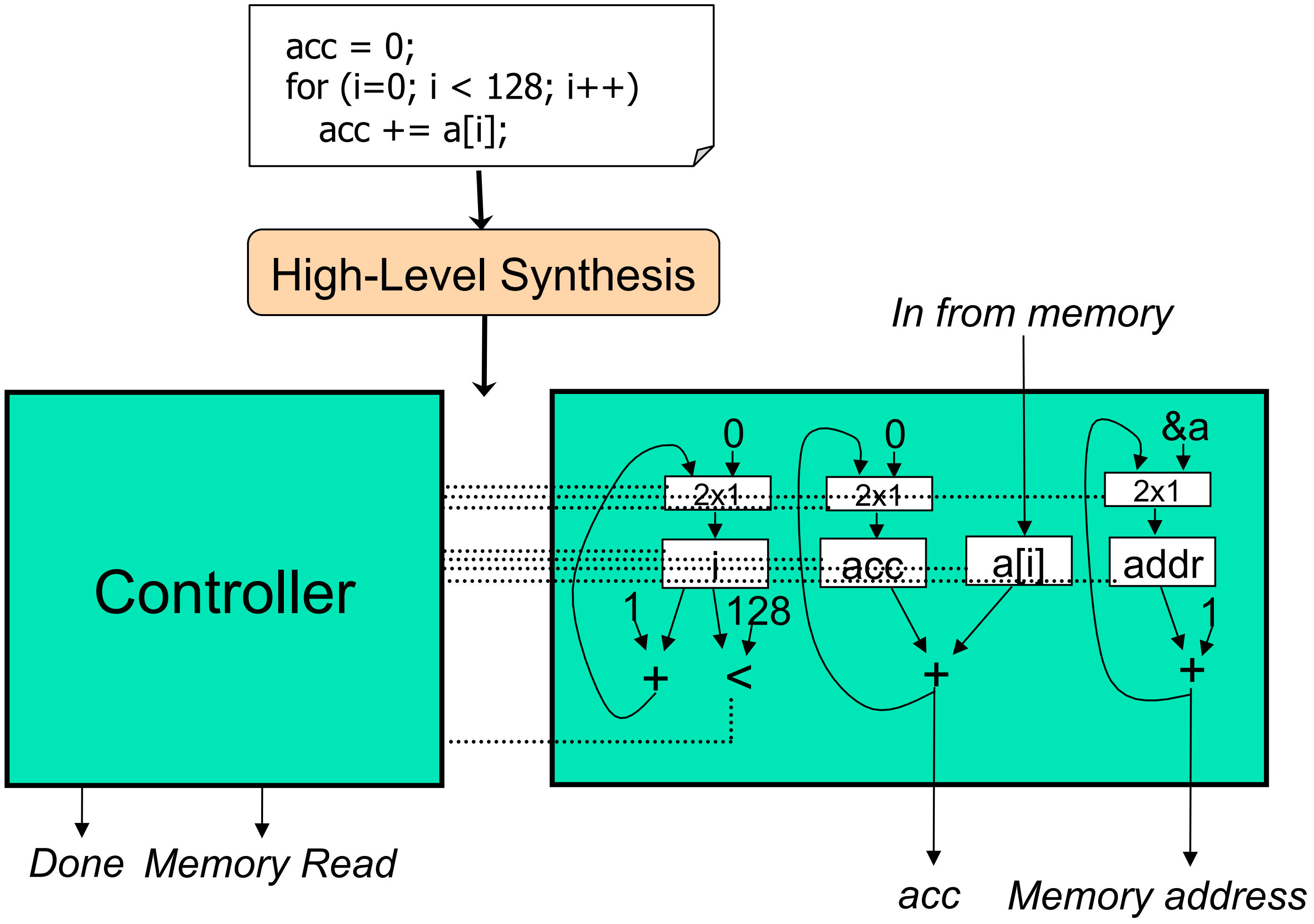
High-Level Synthesis



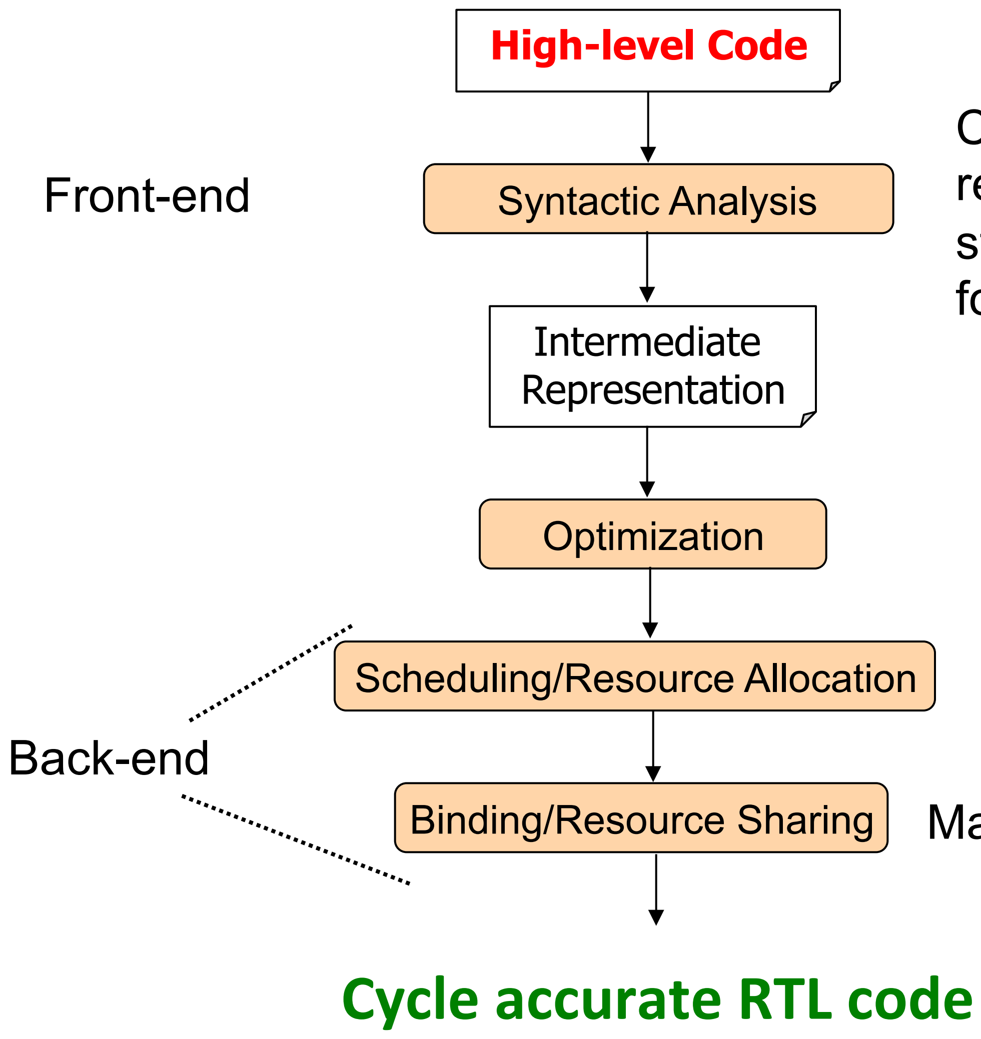
Could be C, C++, Java, Perl, Python, SystemC, ImpulseC, etc.

Usually a RT VHDL/Verilog description, but could as low level as a bit file for FPGA, or a gate netlist.

High-Level Synthesis – Overview



Main Steps



Converts code to intermediate representation - allows all following steps to use language independent format.

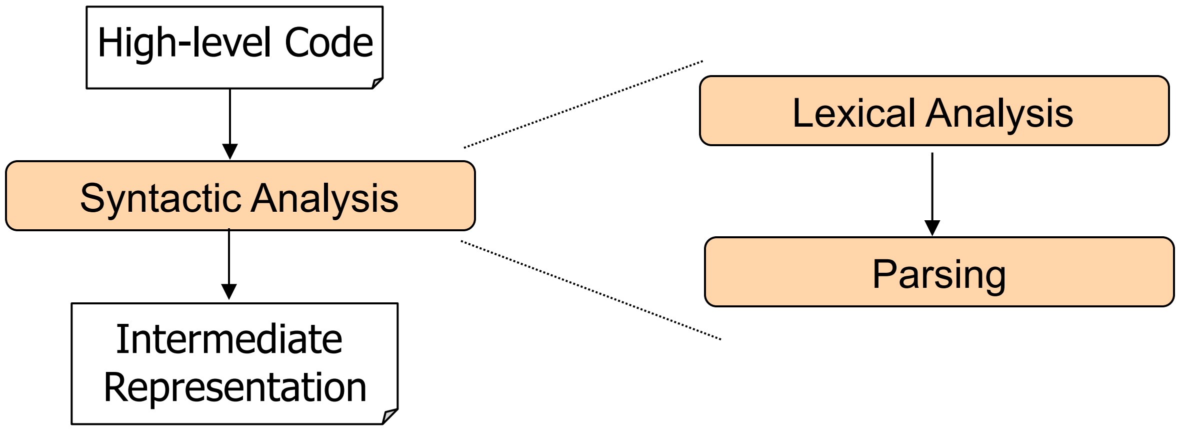
Determines when each operation will execute, and resources used

Maps operations onto physical resources

Parsing & Syntactic Analysis

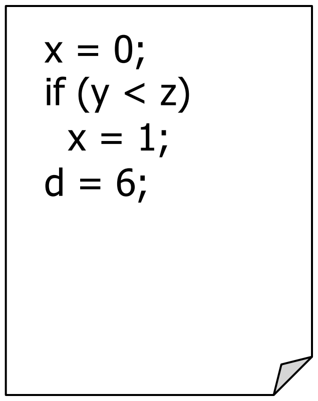
Syntactic Analysis

- Definition: Analysis of code to verify syntactic correctness
  - Converts code into intermediate representation
- Steps: similar to SW compilation
  1) Lexical analysis (Lexing)
  2) Parsing
  3) Code generation – intermediate representation

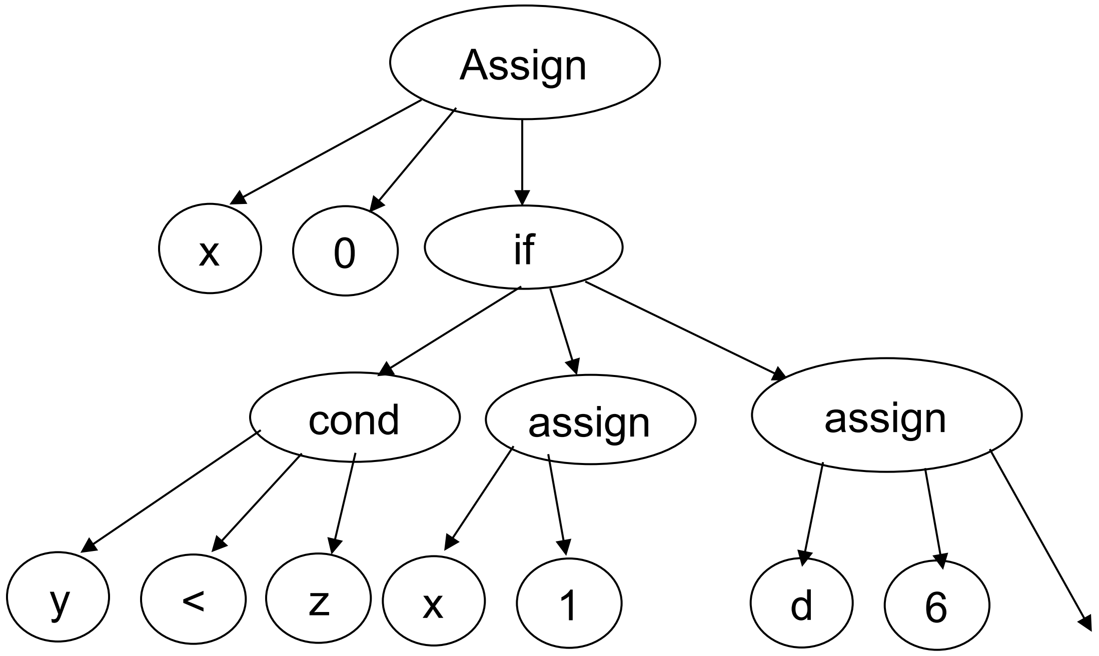


Intermediate Representation

- Parser converts tokens to intermediate representation
  - Usually, an abstract syntax tree

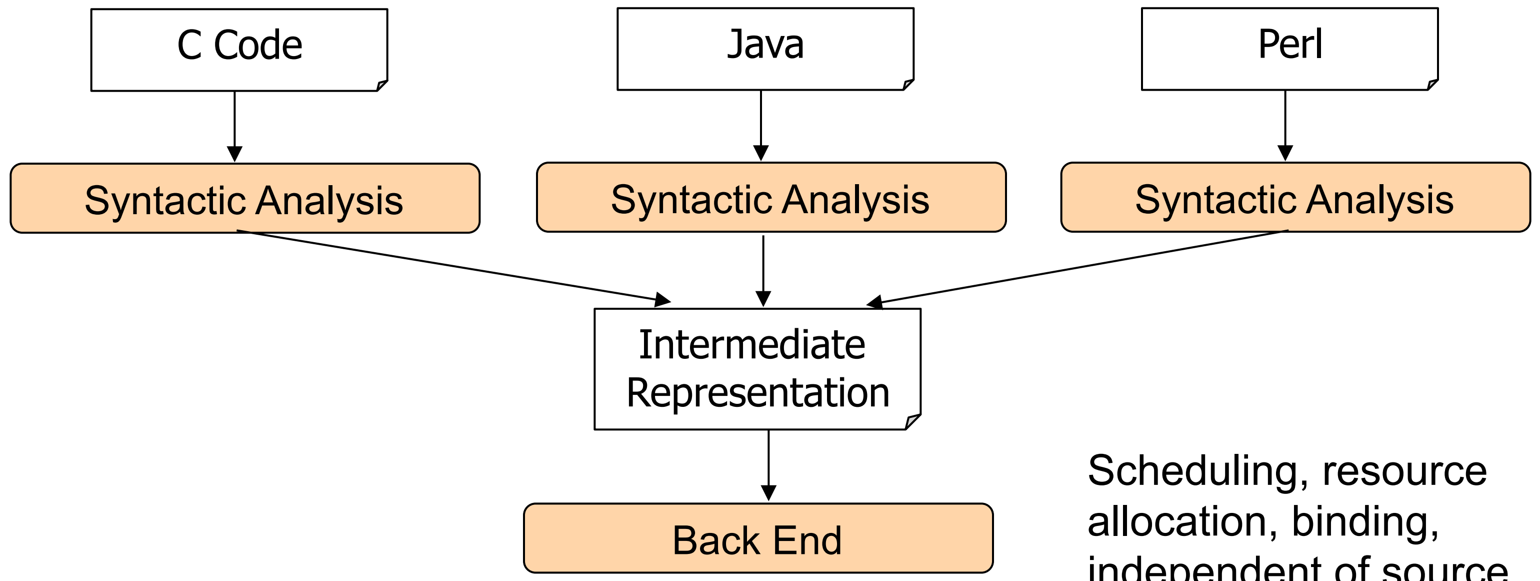


```
x = 0;
if (y < z)
  x = 1;
d = 6;
```



Intermediate Representation

- Why use intermediate representation?
  - Easier to analyze/optimize than source code
  - Theoretically can be used for all languages
    + Makes synthesis back end language independent



Scheduling, resource allocation, binding, independent of source language - sometimes optimizations too

Intermediate Representation

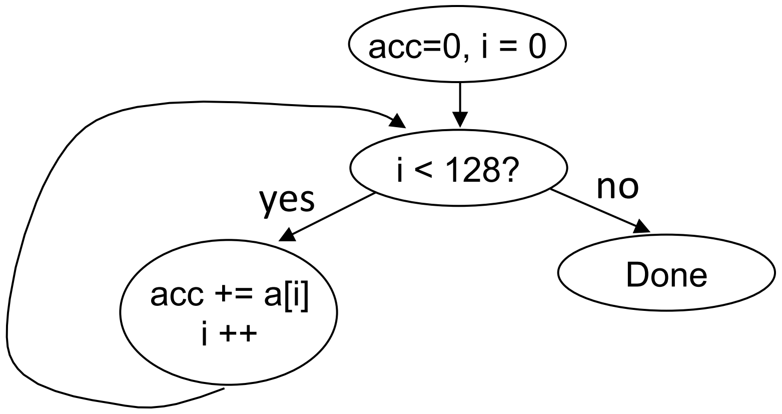
- Different Types
  - Abstract Syntax Tree
  - Control/Data Flow Graph (CDFG)
  - Sequencing Graph

    + ...
- We will focus on CDFG
  - Combines control flow graph (CFG) and data flow graph (DFG)

Control Flow Graphs (CFGs)

- Represents control flow dependencies of *basic blocks*
- A basic block is a section of code that always executes from beginning to end
  - I.e. no jumps into or out of block

```
acc = 0;
for (i=0; i < 128; i++)
   acc += a[i];
```



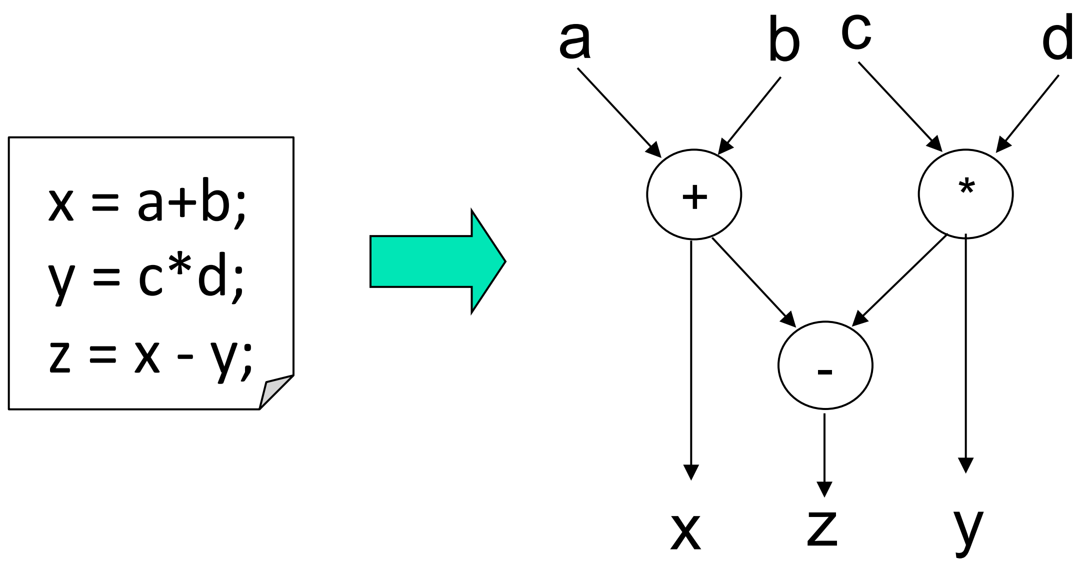
Control Flow Graphs: Your Turn

- Find a CFG for the following code.

```
i = 0;
while (i < 10) {
   if (x < 5)
      y = 2;
   else if (z < 10)
      y = 6;
   i++;
}
```

Data Flow Graphs

- Represents data dependencies between operations

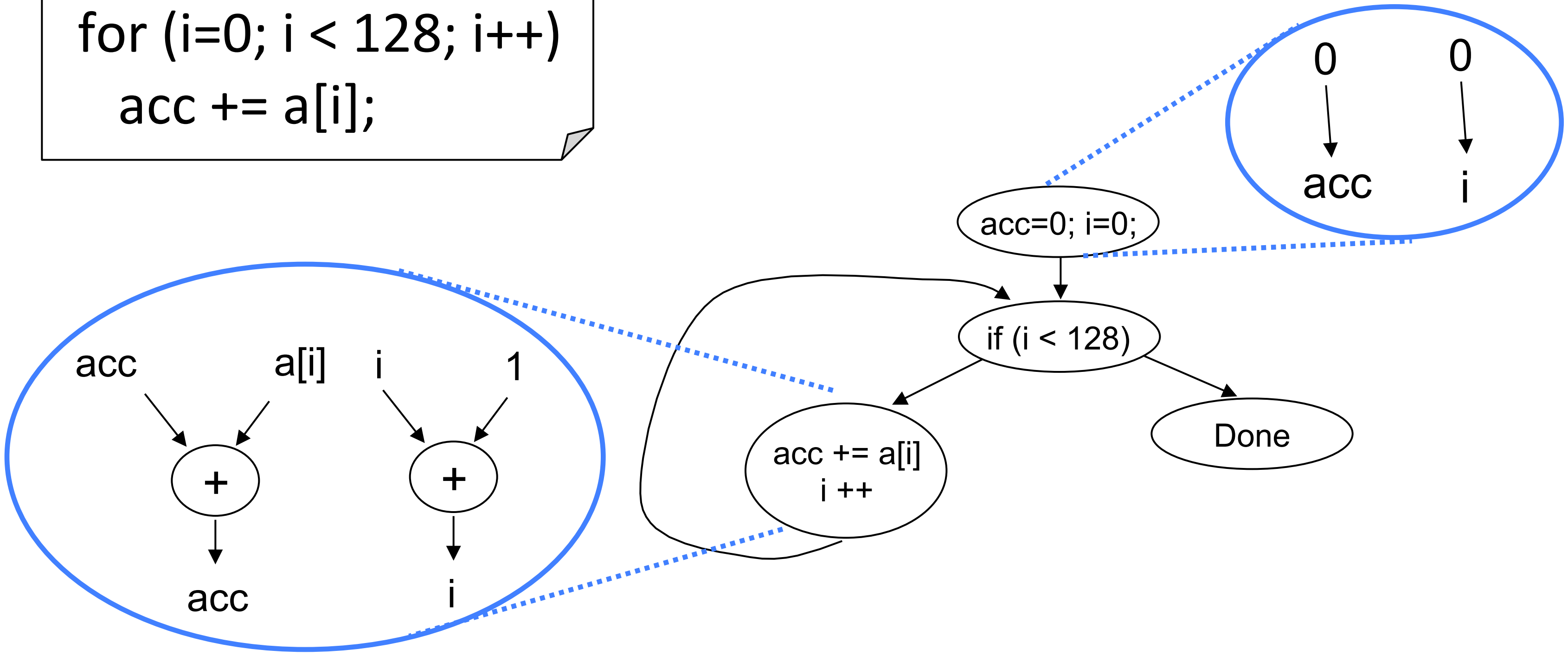


```
x = a+b;
y = c*d;
z = x - y;
```

Control/Data Flow Graph

- Combines CFG and DFG
  - Maintains DFG for each node of CFG

```
acc = 0;
for (i=0; i < 128; i++)
  acc += a[i];
```



Transformation/Optimization

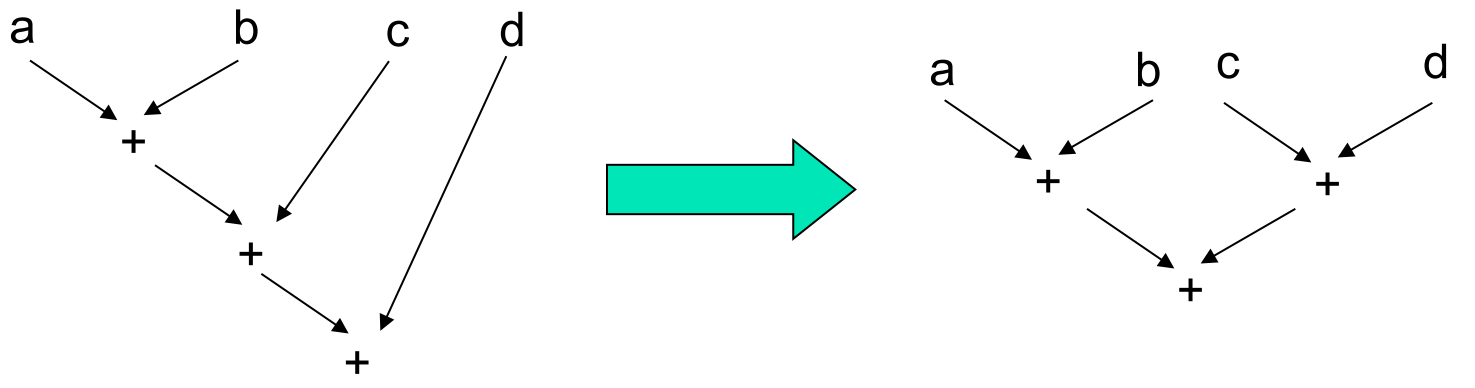
Synthesis Optimizations

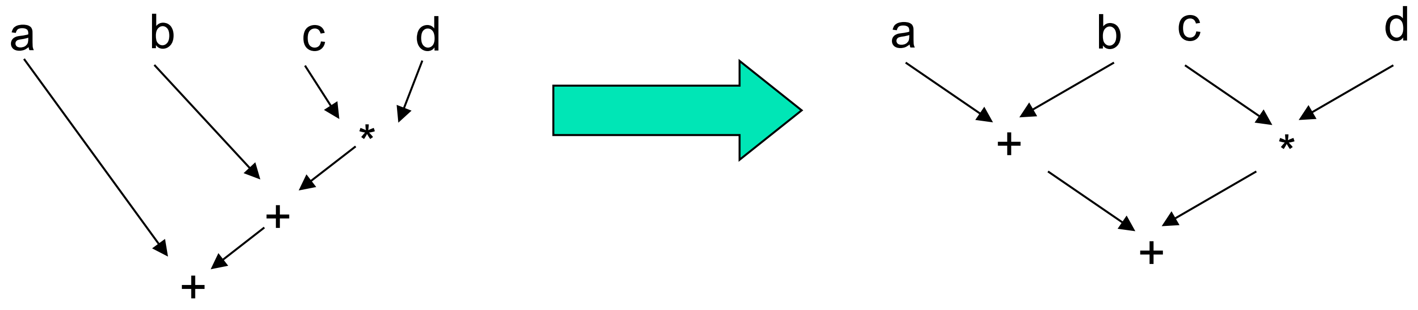
- After creating CDFG, high-level synthesis optimizes it with the following goals
  - Reduce area
  - Improve latency
  - Increase parallelism
  - Reduce power/energy
- 2 types of optimizations
  - Data flow optimizations
  - Control flow optimizations

Data Flow Optimizations

- Tree-height reduction
  - Generally made possible from commutativity, associativity, and distributivity

$$x = a + b + c + d$$





Data Flow Optimizations

- Operator Strength Reduction
  - Replacing an expensive (“strong”) operation with a faster one
  - Common example: replacing multiply/divide with shift

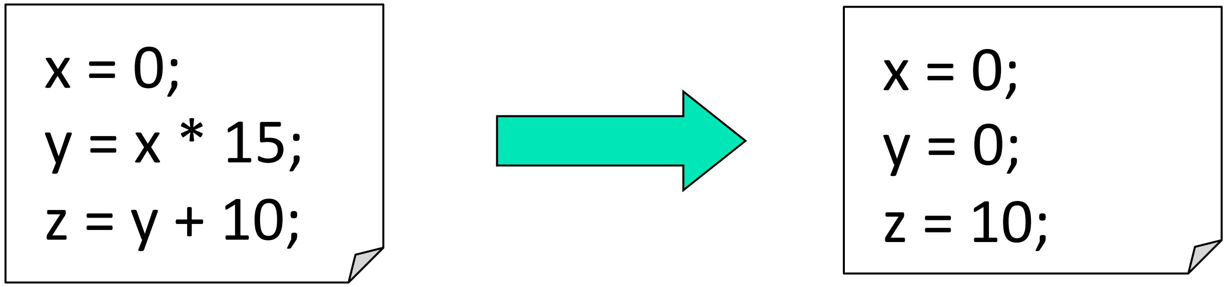
1 multiplication		0 multiplications
`b[i] = a[i] * 8;`	⇒	`b[i] = a[i] << 3;`

`a = b * 5;`	⇒	`c = b << 2;`
		`a = b + c;`

`a = b * 13;`	⇒	`c = b << 2;`
		`d = b << 3;`
		`a = c + d + b;`

Data Flow Optimizations

- Constant propagation
  - Statically evaluate expressions with constants

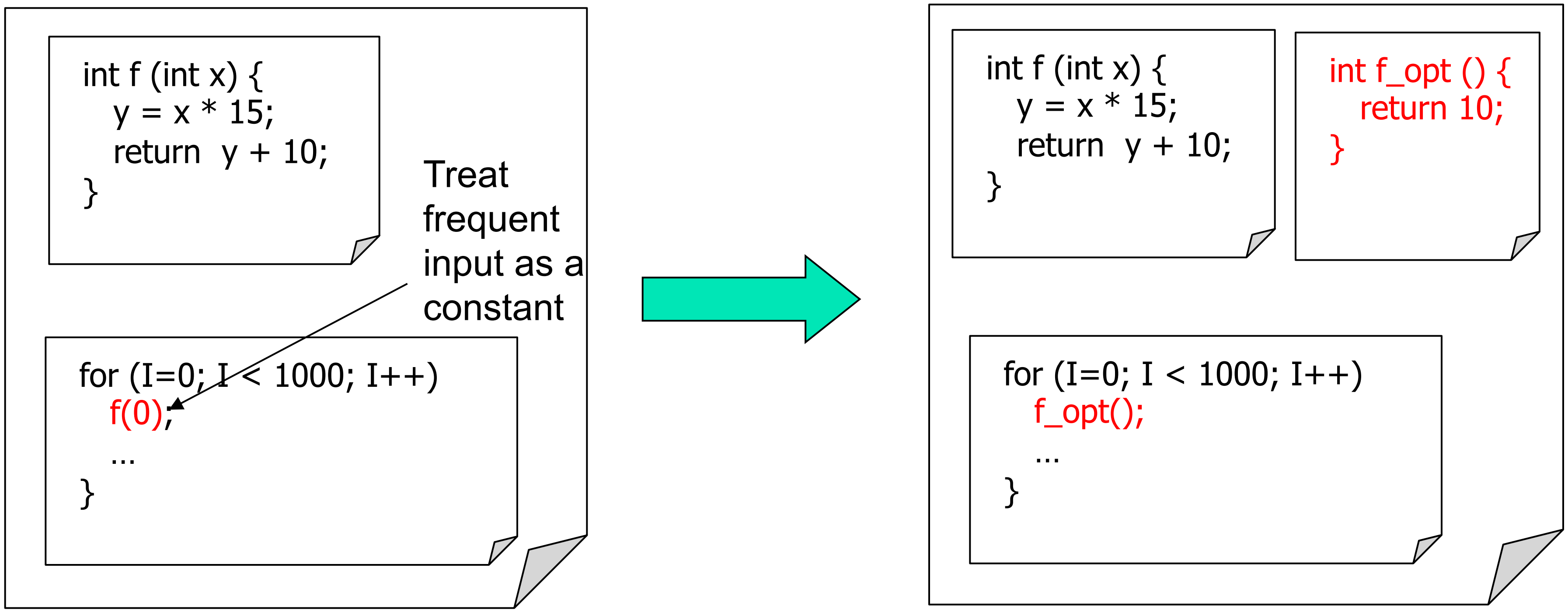


```
x = 0;
y = x * 15;
z = y + 10;
```

```
x = 0;
y = 0;
z = 10;
```

Data Flow Optimizations

- Function Specialization
  - Create specialized code for common inputs
    - + Treat common inputs as constants
    - + If inputs not known statically, must include if statement for each call to specialized function



```
int f (int x) {
   y = x * 15;
   return  y + 10;
}
```

Treat frequent input as a constant

```
for (I=0; I < 1000; I++)
   f(0);
   …
}
```

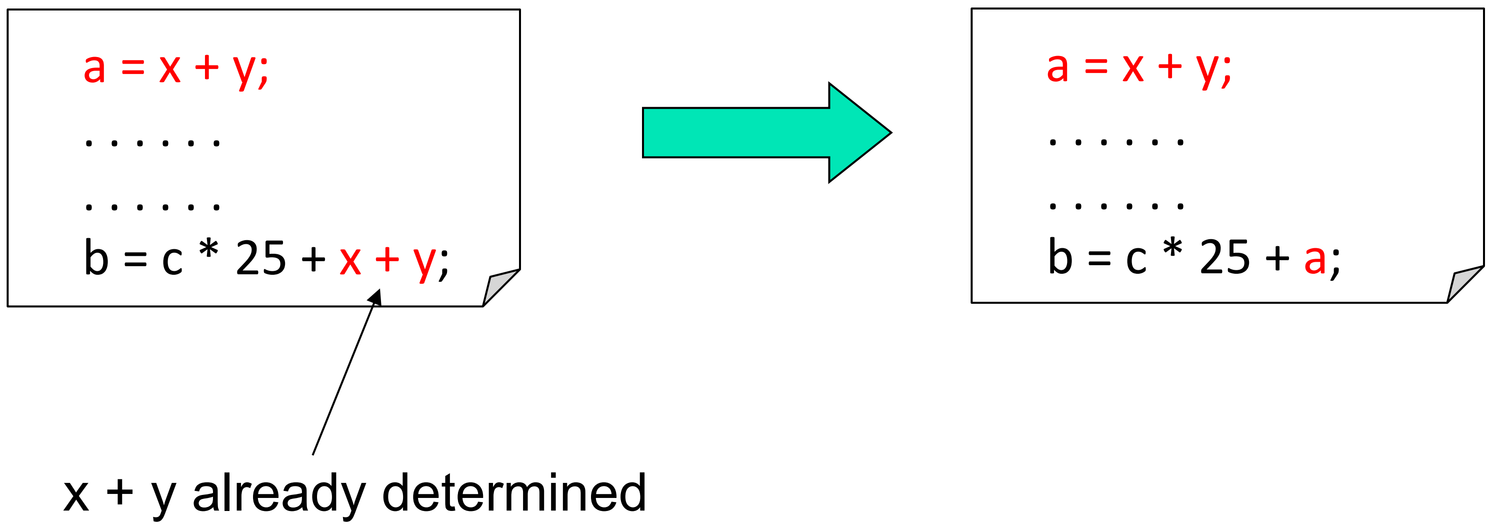
```
int f (int x) {
   y = x * 15;
   return  y + 10;
}
```

```
int f_opt () {
   return 10;
}
```

```
for (I=0; I < 1000; I++)
   f_opt();
   …
}
```

Data Flow Optimizations

- Common sub-expression elimination
  - If expression appears more than once, repetitions can be replaced



Data Flow Optimizations

- Dead code elimination
  - Remove code that is never executed
    + May seem like stupid code, but often comes from constant propagation or function specialization

```
int f (int x) {
   if (x > 0 )
      a = b * 15;
   else
      a = b / 4;
   return  a;
}
```

```
int f_opt () {
   a = b * 15;
   return  a;
}
```

Specialized version for x > 0 does not need else branch - “dead code”

Data Flow Optimizations

- Code motion (hoisting/sinking)
  - Avoid repeated computation

```
for (I=0; I < 100; I++) {
  z = x + y;
  b[i] = a[i] + z ;
}
```

$\Rightarrow$

```
z = x + y;
for (I=0; I < 100; I++) {
  b[i] = a[i] + z ;
}
```

Control Flow Optimizations

- Loop Unrolling
  - Replicate body of loop
    + May increase parallelism

```
for (i=0; i < 128; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < 128; i+=2) {
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1]
}
```

Control Flow Optimizations

- Function inlining – replace function call with body of function
  - Common for both SW and HW
  - SW: Eliminates function call instructions
  - HW: Eliminates unnecessary control states

```
for (i=0; i < 128; i++)
   a[i] = f( b[i], c[i] );
. . . .
int f (int a, int b) {
  return a + b * 15;
}
```

```
for (i=0; i < 128; i++)
   a[i] = b[i] + c[i] * 15;
```

Control Flow Optimizations

- Conditional Expansion – replace **if** with logic expression
  - Execute **if/else** bodies in parallel

```
y = ab
if (a)
   x = b+d
else
   x = bd
```

[DeMicheli]

```
y = ab
x = a(b+d) + a ‘bd
```

Can be further optimized to:

```
y = ab
x = y + d(a+b)
```

Example

- Optimize this

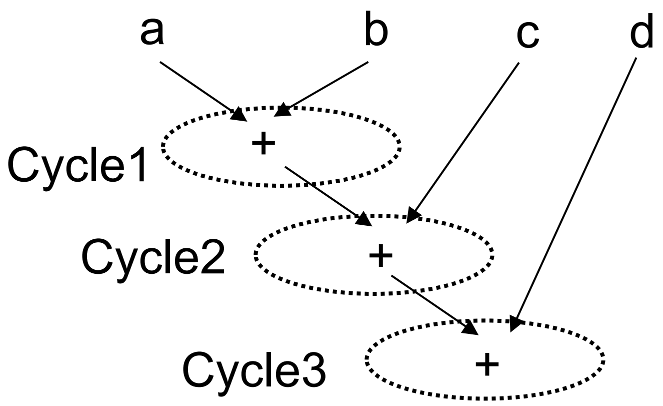
```
x = 0;
y = a + b;
if (x < 15)
   z = a + b - c;
else
   z = x + 12;
output = z * 12;
```

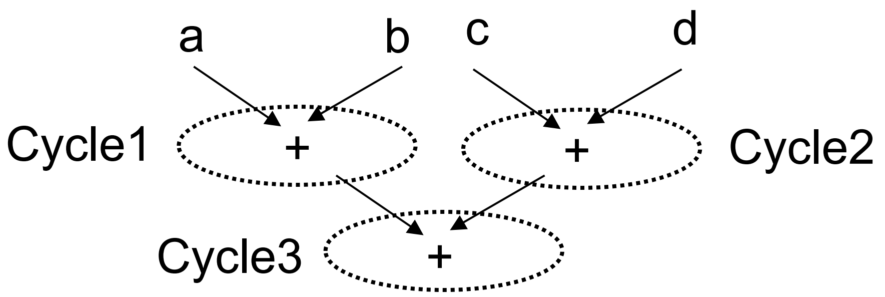
Scheduling/Resource Allocation

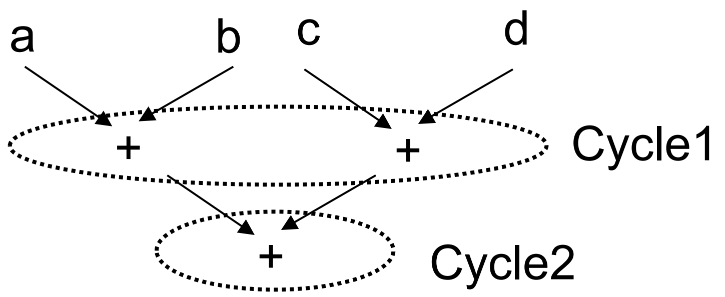
Scheduling

- Scheduling assigns a start time to each operation in DFG
  - Start times must not violate dependencies in DFG
  - Start times must meet performance constraints
    - Alternatively, resource constraints
- Performed on the DFG of each CFG node
  - Cannot execute multiple CFG nodes in parallel

Examples


a
b
c
d
Cycle1
+
Cycle2
+
Cycle3
+


a
b
c
d
Cycle1
+
+
Cycle2
Cycle3
+

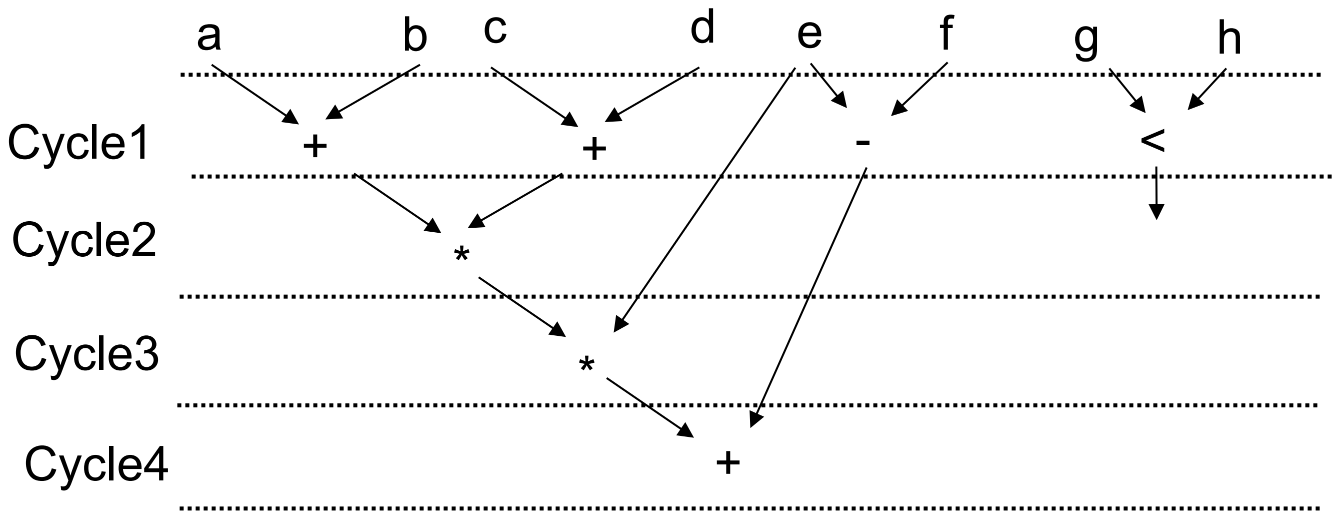

a
b
c
d
+
+
Cycle1
+
Cycle2

Scheduling Problems

- Several types of scheduling problems
  - Usually some combination of performance and resource constraints
- Problems:
  - Unconstrained
    - + Not very useful, every schedule is valid
  - Minimum latency
  - Latency constrained
  - Mininum-latency, resource constrained
    - + i.e. find the schedule with the shortest latency, that uses less than a specified # of resources
    - + NP-Complete
  - Mininum-resource, latency constrained
    - + i.e. find the schedule that meets the latency constraint (which may be anything), and uses the minimum # of resources
    - + NP-Complete

Minimum Latency Scheduling

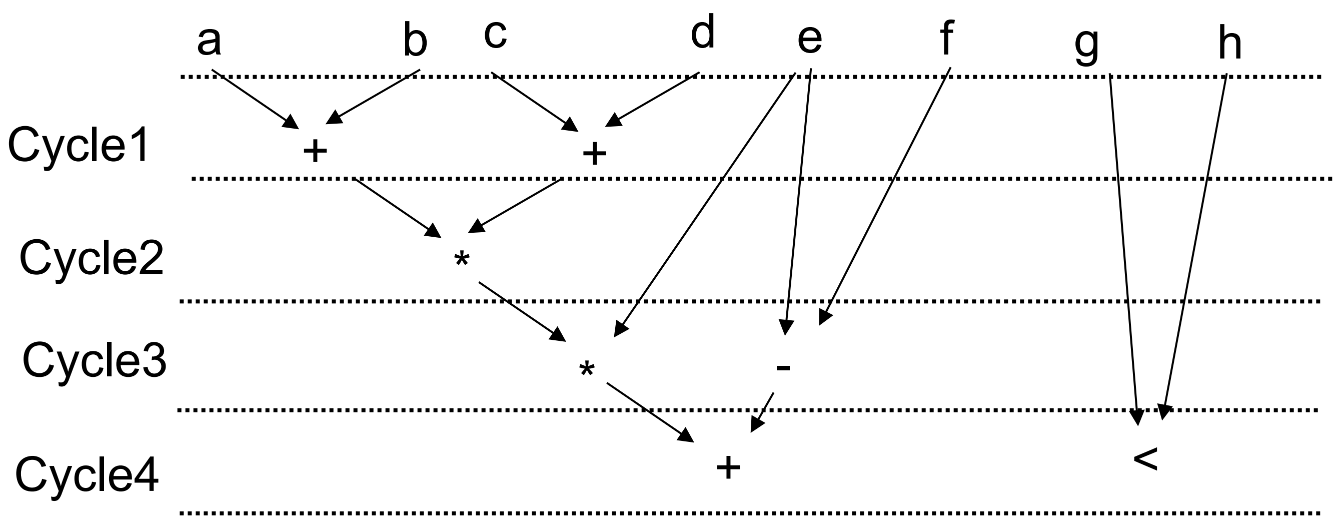
- ASAP (as soon as possible) algorithm
  - Find a candidate node
    - + Candidate is a node whose predecessors have been scheduled and completed (or has no predecessors)
  - Schedule node one cycle later than max cycle of predecessor
  - Repeat until all nodes scheduled



Minimum possible latency - 4 cycles

Minimum Latency Scheduling

- ALAP (as late as possible) algorithm
  - Run ASAP, get minimum latency L
  - Find a candidate
    - Candidate is node whose successors are scheduled (or has none)
  - Schedule node one cycle *before min* cycle of successor
    - Nodes with no successors scheduled to cycle L
  - Repeat until all nodes scheduled



L = 4 cycles

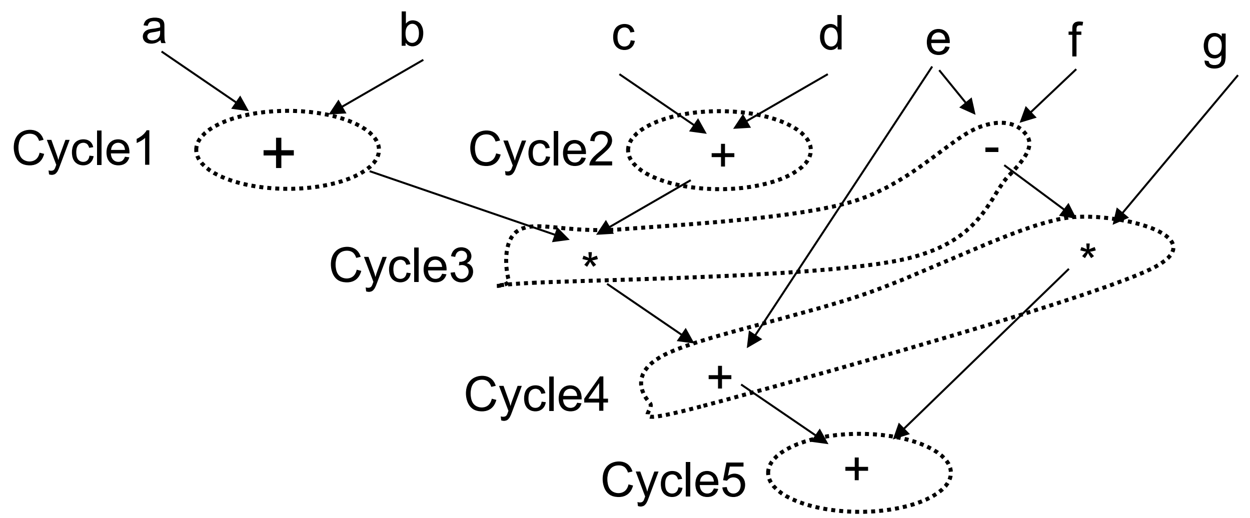
Latency-Constrained Scheduling

- Instead of finding the minimum latency, find latency less than L
- Solutions:
  - Use ASAP, verify that minimum latency less than *L*.
  - Use ALAP starting with cycle *L* instead of minimum latency (don’t need ASAP)

Scheduling with Resource Constraints

- Schedule must use less than specified number of resources

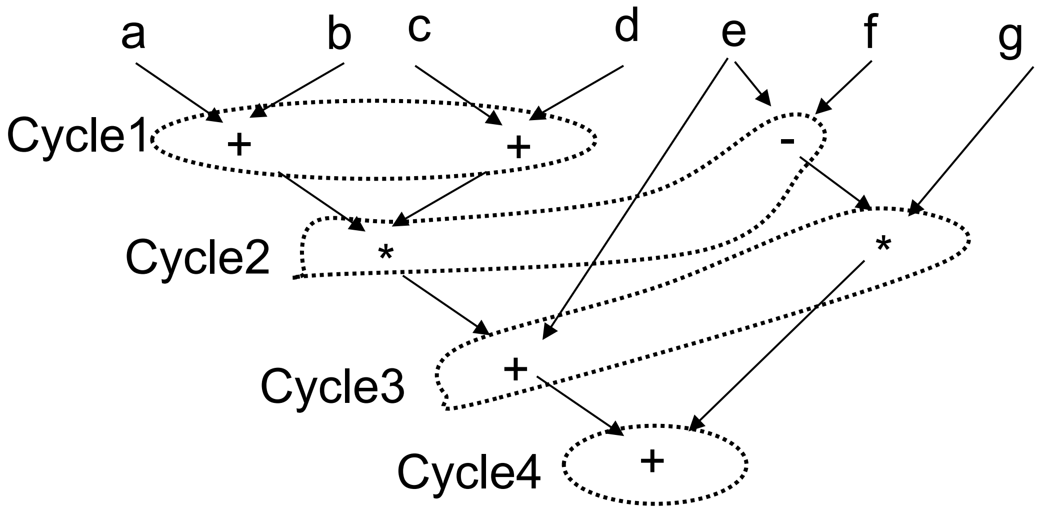
Constraints: 1 ALU (+/-), 1 Multiplier



Scheduling with Resource Constraints

- Schedule must use less than specified number of resources

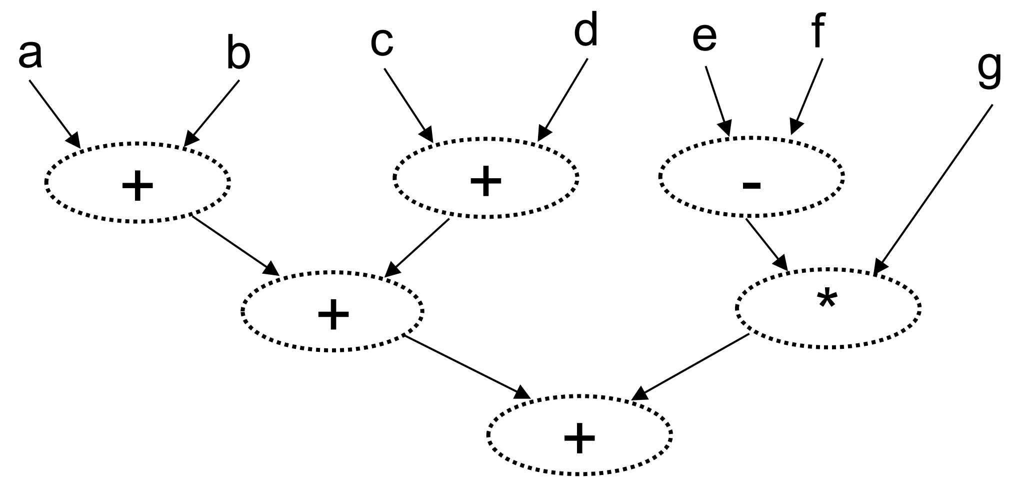
Constraints: 2 ALU (+/-), 1 Multiplier



Mininum-Latency, Resource-Constrained Scheduling

- Definition: Given resource constraints, find schedule that has the minimum latency
  - Example:

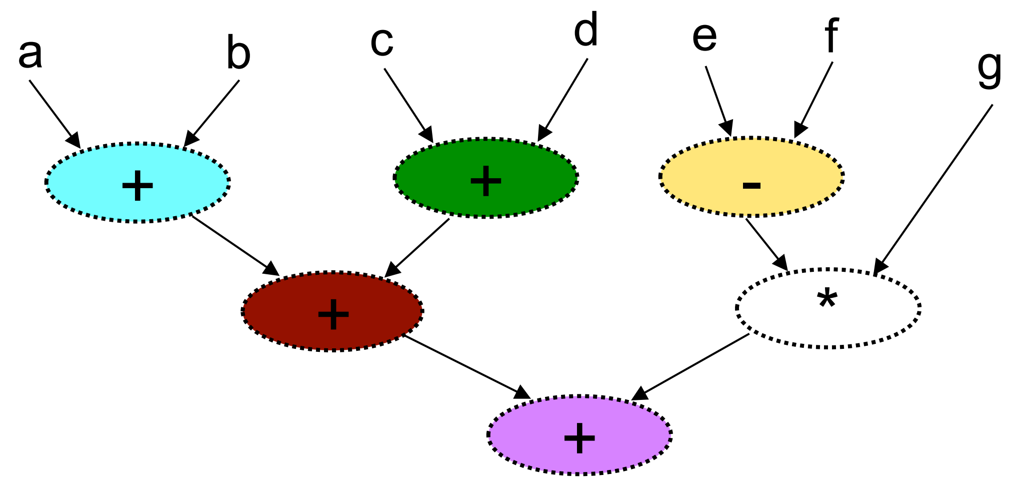
Constraints: 1 ALU (+/-), 1 Multiplier



Mininum-Latency, Resource-Constrained Scheduling

- Definition: Given resource constraints, find schedule that has the minimum latency
  - Example:

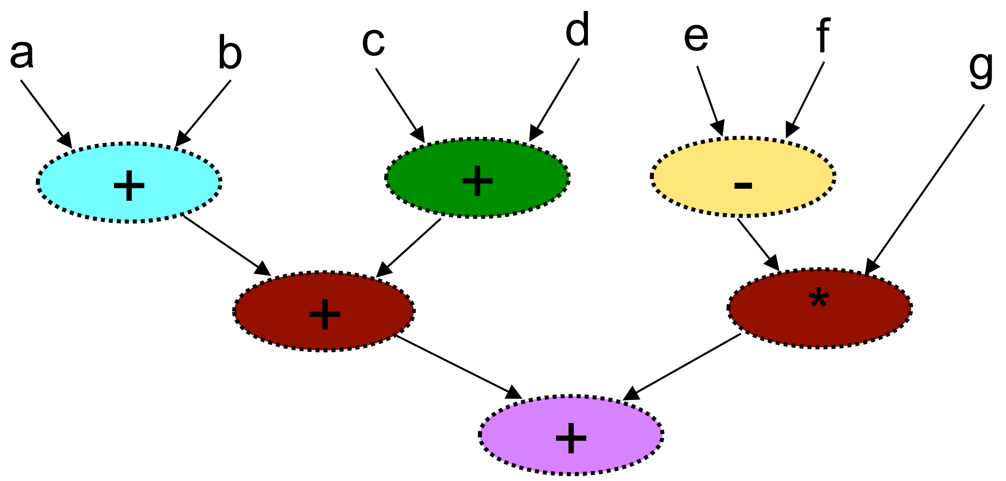
Constraints: 1 ALU (+/-), 1 Multiplier



Mininum-Latency, Resource-Constrained Scheduling

- Definition: Given resource constraints, find schedule that has the minimum latency
  - Example:

Constraints: 1 ALU (+/-), 1 Multiplier



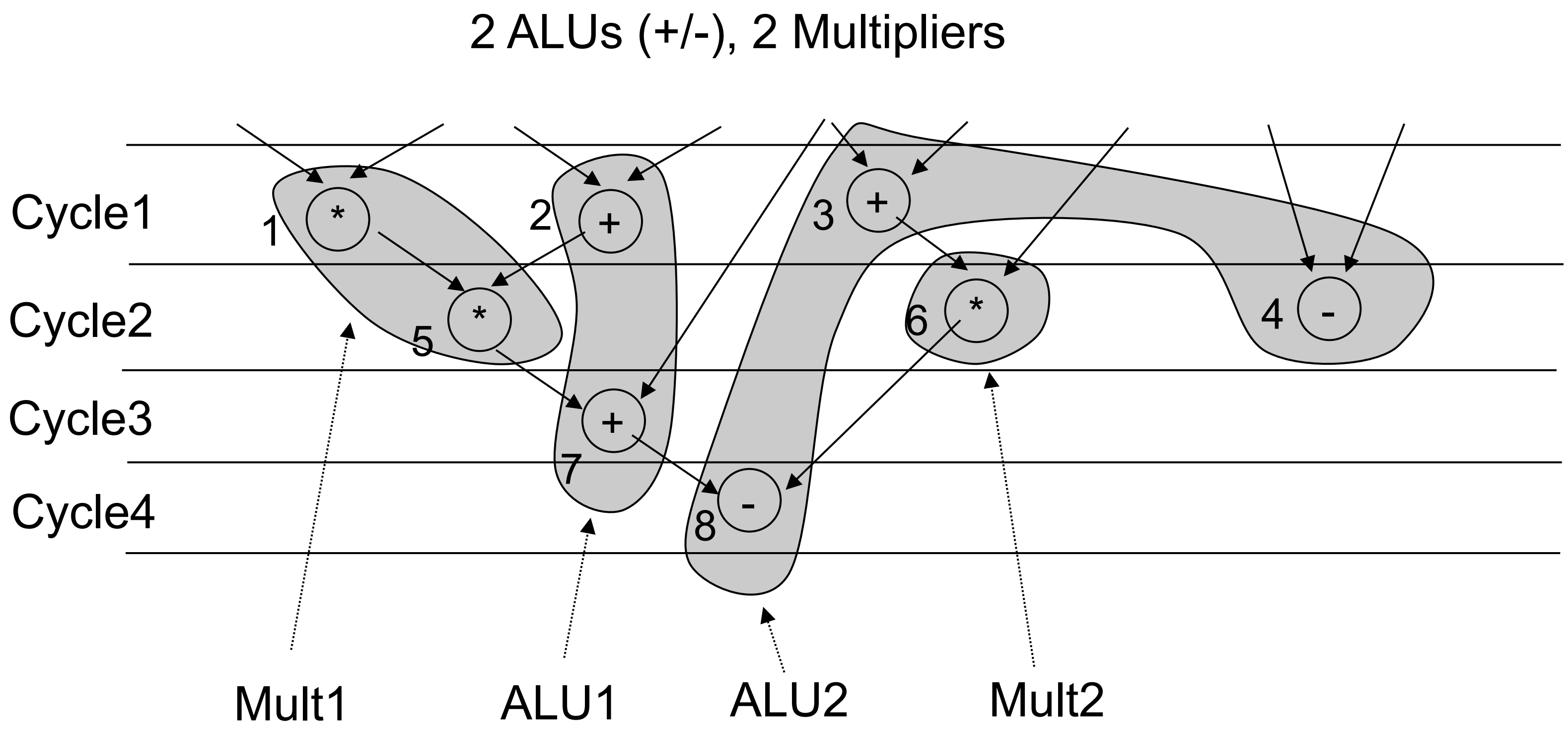
Binding/Resource Sharing

Binding

- During scheduling, we determined:
  - When operations will execute
  - How many resources are needed
- We still need to decide which operations execute on which resources – binding
  - If multiple operations use the same resource, we need to decide how resources are shared - resource sharing.

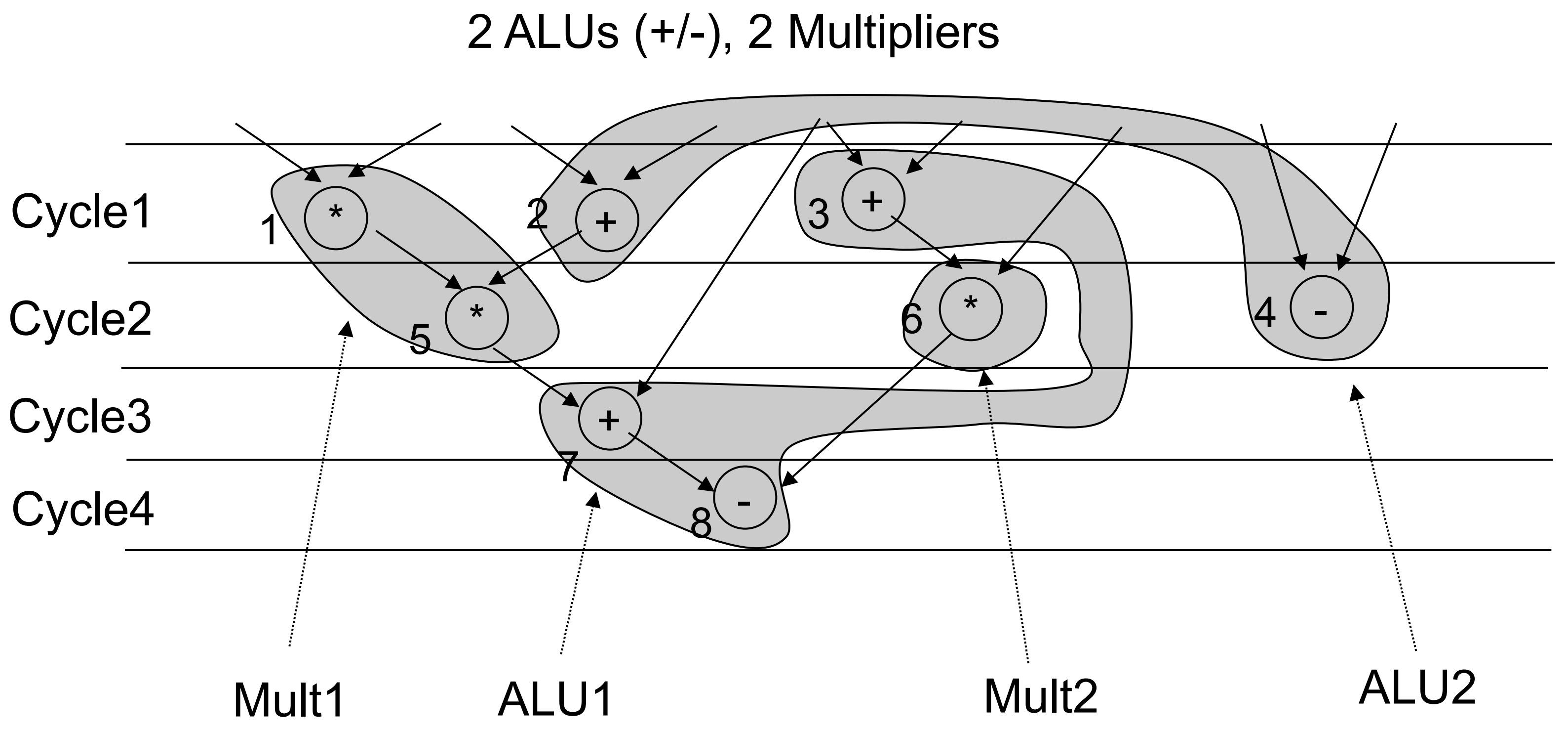
Binding

- Map operations onto resources such that operations in same cycle do not use same resource



Binding

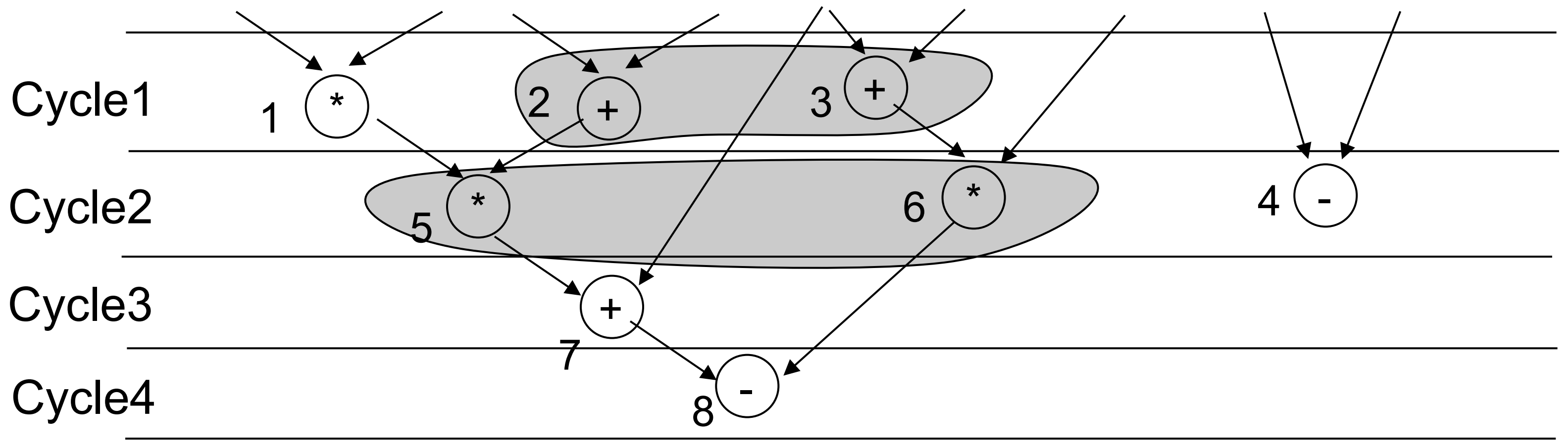
- Many possibilities
  - Bad binding may increase resources, require huge steering logic, reduce clock, etc.



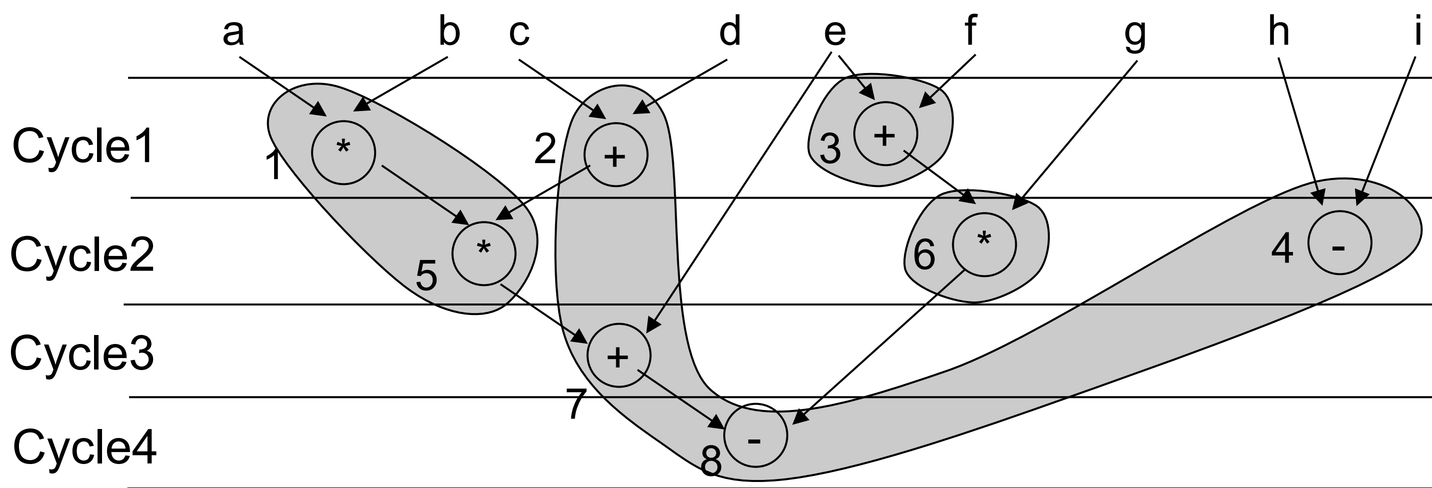
Binding

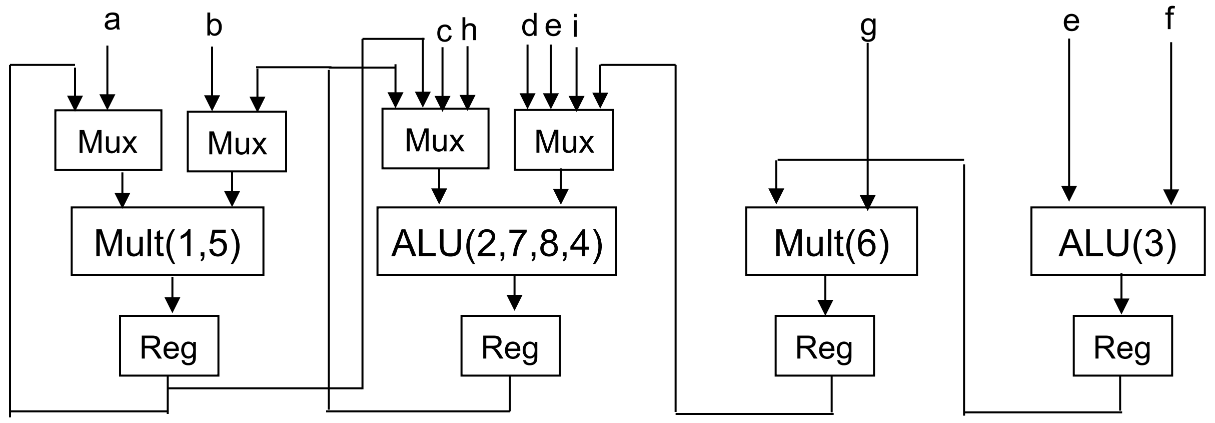
- Can't do this
  - 1 resource can't perform multiple ops simultaneously!

2 ALUs (+/-), 2 Multipliers



Translation to Datapath





1) Add resources and registers
2) Add mux for each input
3) Add input to left mux for each left input in DFG
4) Do same for right mux
5) If only 1 input, remove mux

Summary

Main Steps

- Front-end (lexing/parsing) converts code into intermediate representation
  - We looked at CDFG
- Scheduling assigns a start time for each operation in DFG
  - CFG node start times defined by control dependencies
  - Resource allocation determined by schedule
- Binding maps scheduled operations onto physical resources
  - Determines how resources are shared
- Big picture:
  - Scheduled/Bound DFG can be translated into a datapath
  - CFG can be translated to a controller
  - => High-level synthesis can create a custom circuit for any CDFG!

Limitations

- Task-level parallelism
  - Parallelism in CDFG limited to individual control states
    - Can’t have multiple states executing concurrently
  - Potential solution: use model other than CDFG
    - Kahn Process Networks
      - Nodes represents parallel processes/tasks
      - Edges represent communication between processes
    - High-level synthesis can create a controller+datapath for each process
      - Must also consider communication buffers
  - Challenge:
    - Most high-level code does not have explicit parallelism
      - Difficult/impossible to extract task-level parallelism from code

Limitations

- Coding practices limit circuit performance
  - Very often, languages contain constructs not appropriate for circuit implementation
    - + Recursion, pointers, virtual functions, etc.
- Potential solution: use specialized languages
  - Remove problematic constructs, add task-level parallelism
- Challenge:
  - Difficult to learn new languages
  - Many designers resist changes to tool flow

Limitations

- Expert designers can achieve better circuits
  - High-level synthesis has to work with specification in code
    - + Can be difficult to automatically create efficient pipeline
    - + May require dozens of optimizations applied in a particular order
  - Expert designer can transform algorithm
    - + Synthesis can transform code, but can't change algorithm
- Potential Solution: ???
  - New language?
  - New methodology?
  - New tools?