EN2911X: Reconfigurable Computing
Lecture 10: Design Flow: Compilation & Synthesis (2)

Prof. Sherief Reda
Division of Engineering, Brown University
http://scale.engin.brown.edu
Fall ’09

BROWN

Compilation

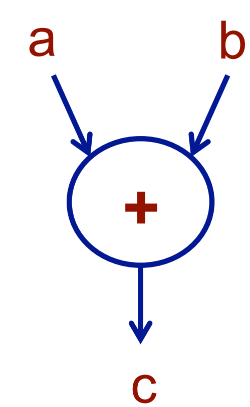
- Reconfigurable configurable has the ability to execute multiple operations in parallel through spatial distribution of the computing resources
- When compiling a SW-based sequential language like (C) into a concurrent language like Verilog, it is necessary to either
  - Manually instruct the compiler to incorporate parallelism either through special instructions or compiler directives
  - Automatically through the compiler or manually by hand expose the concurrency in the application

BROWN

Data-flow graphs (DFG)

- A data-flow graph (DFG) is a graph which represents a data dependencies between a number of operations.
- Dependencies arise from a various reasons
  - An input to an operation can be the output of another operation
  - Serialization constraints, e.g., loading data on a bus and then raising a flag
  - Sharing of resources

- A *dataflow graph* represents operations and data dependencies
  - *Vertex* set is one-to-one mapping with tasks
  - A *directed edge* is in correspondence with the transfer of data from an operation to another one



BROWN

Consider the following example

[Giovanni'94] Design a circuit to numerically solve the following differential equation in the interval [0, a] with step-size *dx*

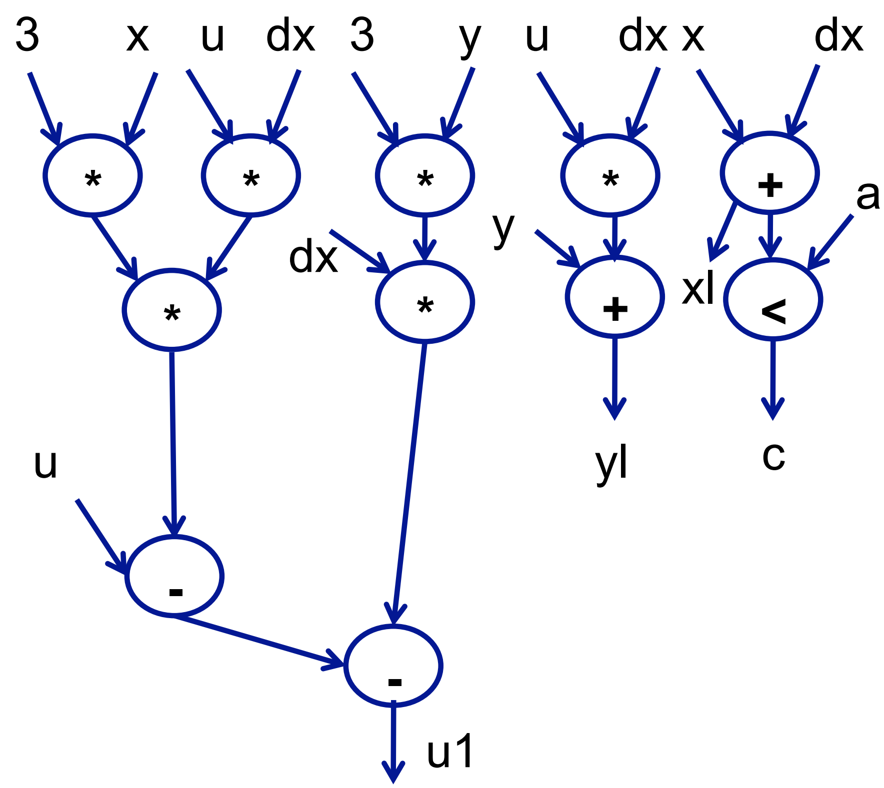
$$y''+3xy'+3y=0$$

$$x(0)=x;\ y(0)=y;\ y'(0)=u$$

```
read (x, y, u, dx, a);
do {
  xl = x + dx;
  ul = u – (3*x*u*dx) – (3*y*dx);
  yl = y + u*dx;
  c = xl < a;
  x = x1; u = u; y = yl;
} while (c);
write(y);
```

Data-flow graph example

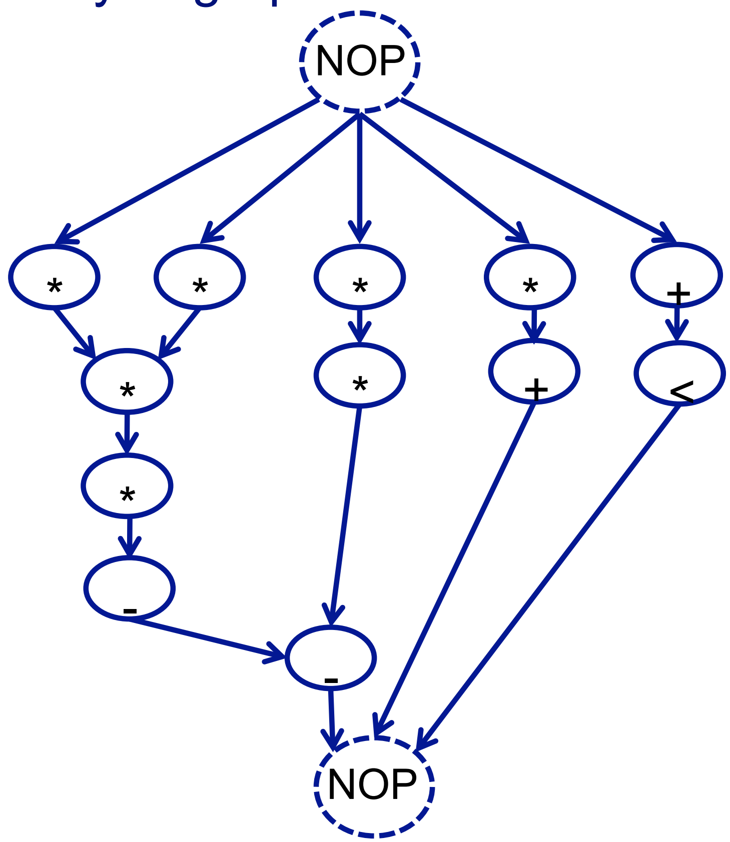
```
xl = x + dx;
ul = u – (3*x*u*dx) – (3*y*dx);
yl = y + u*dx;
c = xl < a;
```



BROWN

Detecting concurrency from DFGs

Extended DFG where vertices can represent links to link graph DFGs in a hierarchy of graphs



Paths in the graph represent concurrent streams of operations

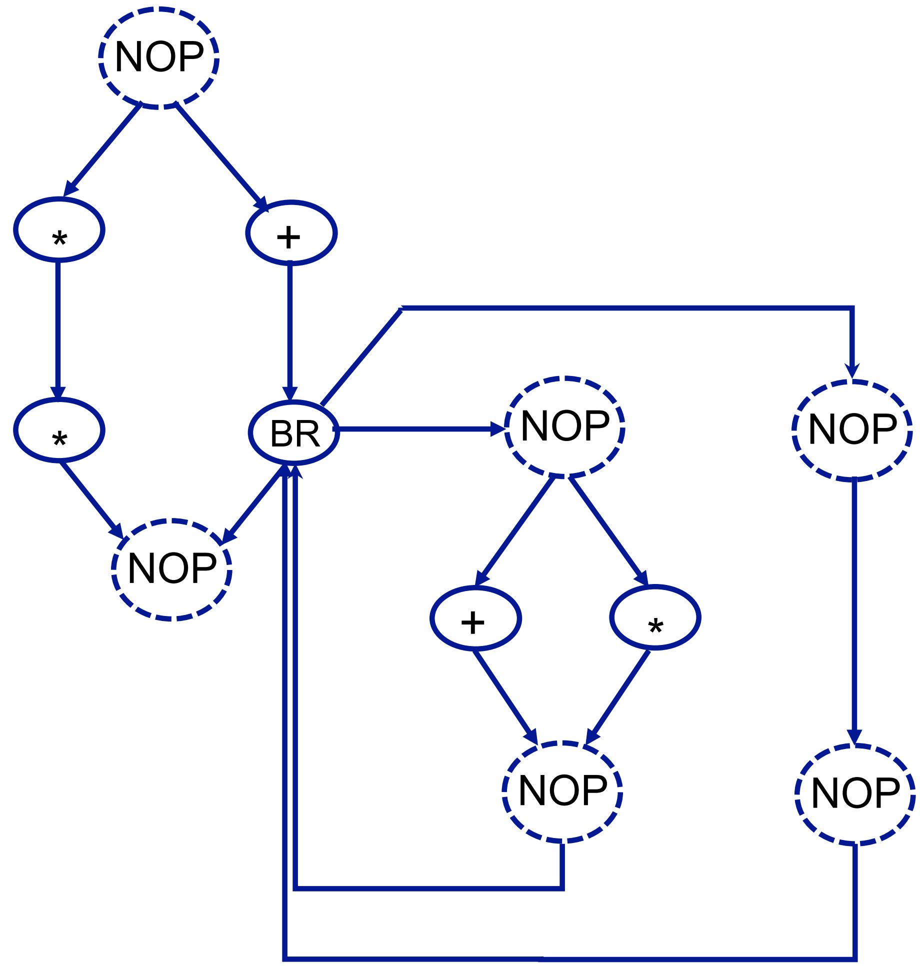
BROWN

Control / data-flow graphs (CDFG)

- Control-flow information (branching and iteration) can be also represented graphically
- Data-flow graphs can be extended by introducing branching vertices that represent operations that evaluate conditional clauses
- Iteration can be modeled as a branch based on the iteration exit condition
- Vertices can also represent model calls

CDFG example

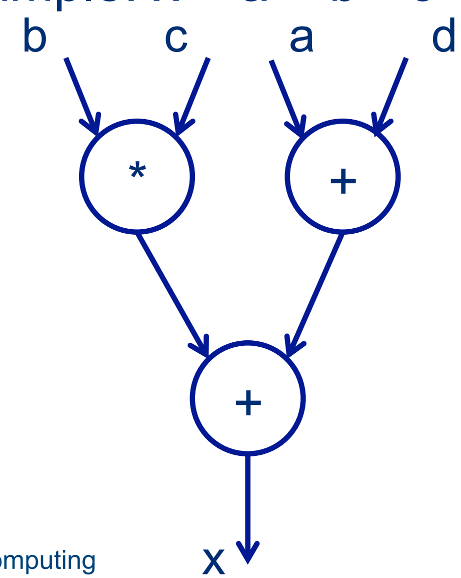
```
x = a * b;
y = x * c;
z = a + b;
if (z ≥ 0) {
  p = m + n;
  q = m * n;
}
```

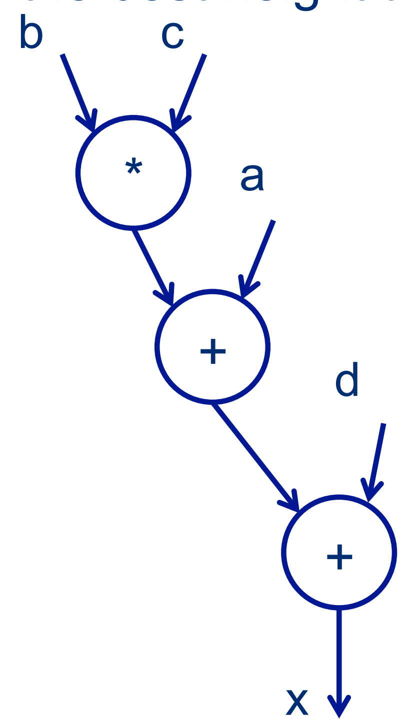


BROWN

Behavioral code optimizations

- *Tree-height reduction* applies to arithmetic expression trees and strives to achieve the expression split into two-operand expressions to exploit parallelism
- The idea is to attempt to balance the expression tree as much as possible
- If we have *n* operations, what is the best height that can be achieved?
- Example: x = a + b * c + d

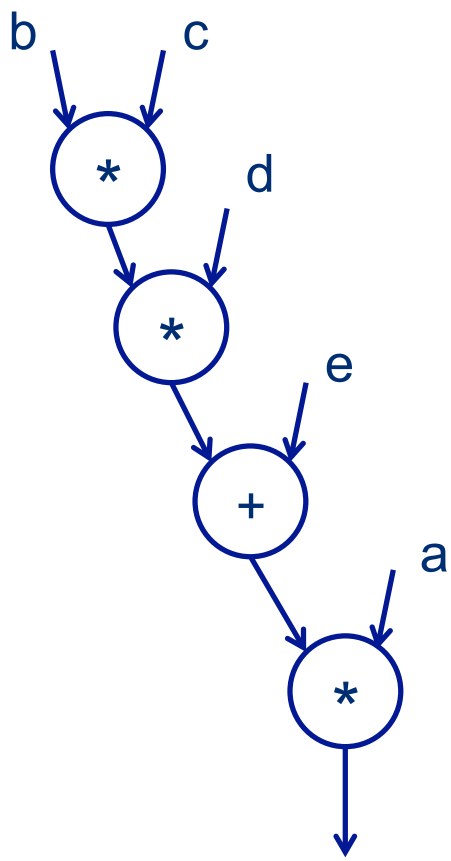




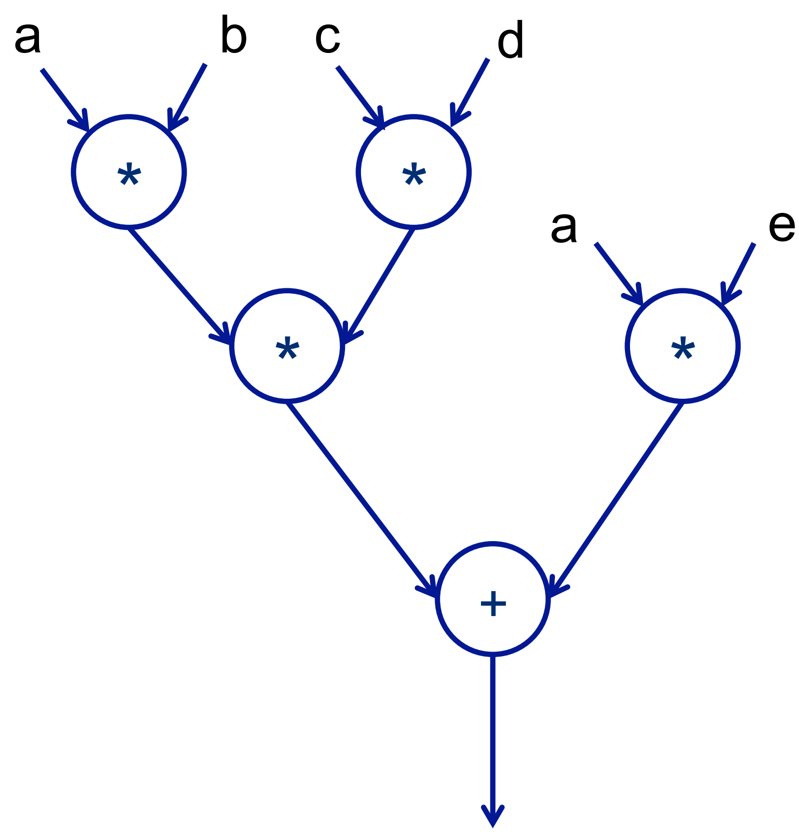
BROWN

Tree-height reduction

x = a*(b*c*d + e)



Exploiting the distributive property at the expense of adding an operation



Constant and variable propagation

- *Constant propagation* consists of detecting constants operands and pre-computing the value of the operation with that operand. The result might a constant which can be propagated to other operations as input
- Example:
  - a = 0; b = a + 1; c = 2 * b

  Replaced by → a = 0; b = 1; c =2

- *Variable propagation* consists of detecting the copies of the variable and using the right-hand side in the following references in place of the left-hand side
- Example:
  - a = x; b = a + 1; c = 2 * a

  Replaced by → a = x; b = x + 1; c = 2 * x

CSE and DCA

- *Common Sub-expression Elimination (CSE)* avoids unnecessary computations.
- Example:
  - a = x + y; b = a + 1; c = x + y

  Can be replaced by → a = x + y; b = a + 1; c = a

- *Dead code elimination (DCA)*. Dead code consists of all operations that cannot be reached, or whose results is never referenced elsewhere. Example:

a = x; b = x + 1; c = 2 * x;

The first assignment can be removed if it is never subsequently referenced

BROWN

Operator strength reduction & code motion

- *Operator strength reduction* means reducing the cost of implementing an operator by using a “weaker” one (that uses less hardware / simpler and faster)
- <u>Example:</u>
  - a = x^2; b = 3 * x

  Replaced by → a = x * x; t = x << 1; b = x + t

- *Code motion* often applies to loop invariants, i.e., quantities that are computed inside an iterative construct but whose values do not change from iteration to iteration.
- <u>Example:</u>
  - for (i = 1; i < a *b) {…}

  Replaced by → t = a * b; for (i =1; i <= t) {…}

Control-flow-based transformations

- Control-flow transformations are typically utilized to create more opportunities for data-flow transformations to be exercised
- *Model expansion* consists in flattening locally the model call hierarchy. Therefore the called model disappears, being swallowed by the calling one.
- A possible benefit is that the scope of application of some optimization techniques is enlarged yielding potentially a better circuit
- <u>Example:</u>
  - x = a + b; y = a*b; z = func(x, y)
  - where func(p, q) = {t = q-p+p*q; return t;}

  → By expanding func, we get

  x = a + b; y = a* b; z = a – b+a*b;

  → CSE x= a+b; y = a*b; z = a-b+y;

BROWN

Conditional expansion

- A conditional construct can be always transformed in a parallel construct with a test in the end.
- Conditional expansion can increase the performance of the circuit when the conditional clause depends on some late-arriving signal.
- However, it can preclude the possibility of hardware sharing

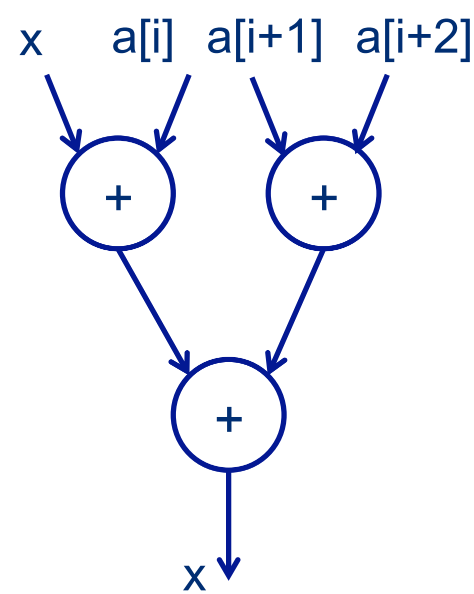
- If (C) then x=A else x=B
  ➔ compute A and B in parallel, x= C ?A:B

Loop unrolling (expansion)

- In *loop expansion*, or unrolling, a loop is replaced by as many instances of the body as the number of operations. The benefit is in expanding the scope of other transformations
- Example:

```
x = 0;
for (i = 1; i <= 12; i++) {
  x = x + a[i];
}
```

```
x = 0;
for (i = 1; i <= 12; i = i+3) {
  x = x + a[i] + a[i+1] + a[i+2];
}
```



BROWN