Vectorizing Divergent Control-Flow for SIMD Applications

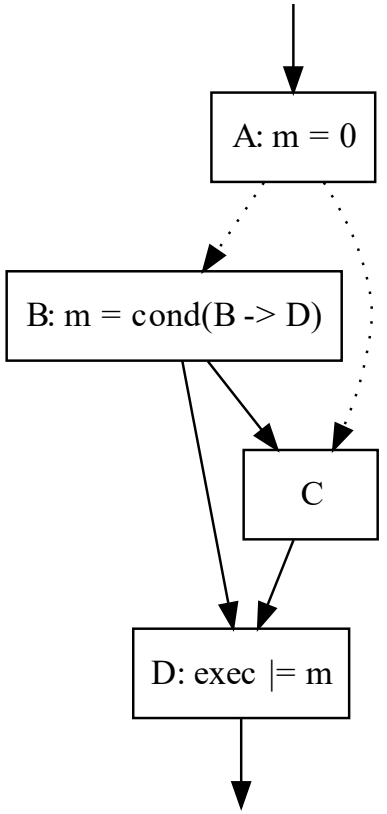
Fabian Wahlster

Technische Universität München

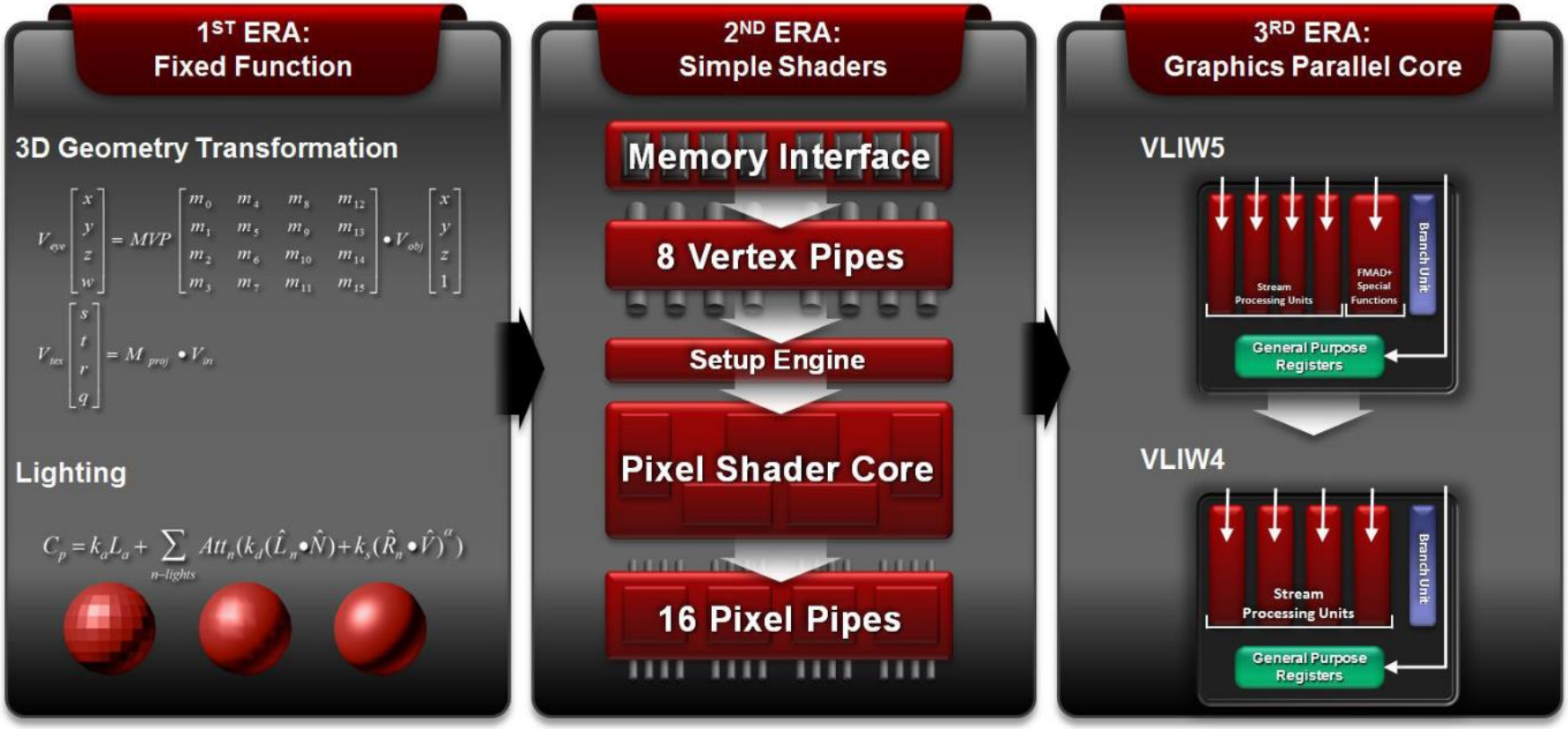
Fakultät für Informatik

Lehrstuhl für Rechnerarchitektur & Parallele Systeme

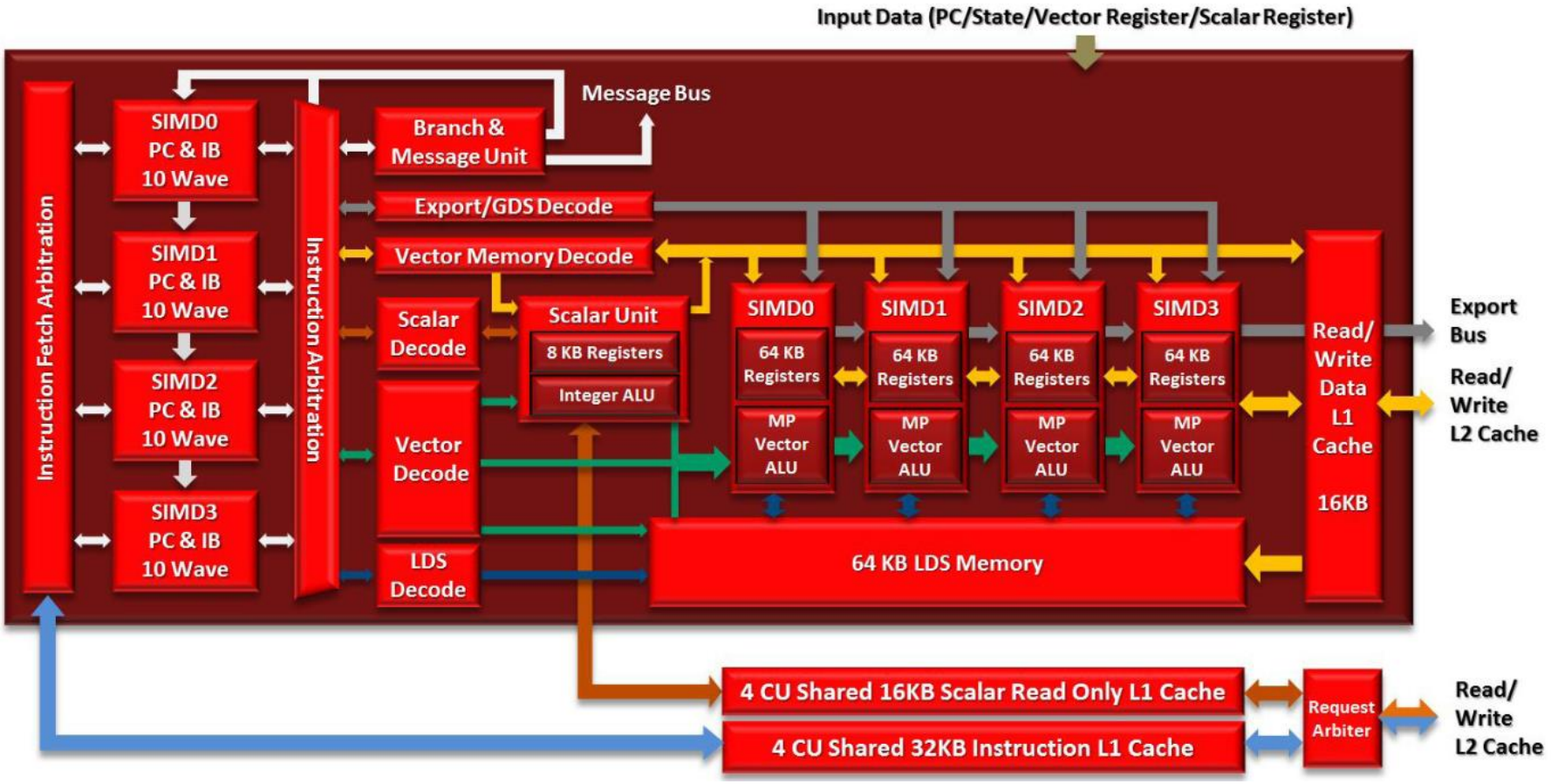
Garching, 14. Februar 2019



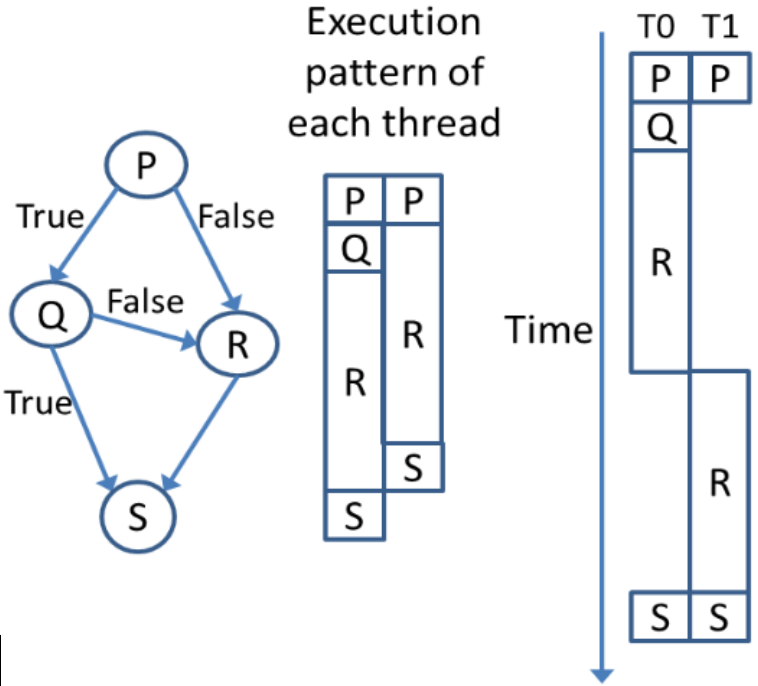
Hardware: Fixed function graphics to GPGPU



Hardware: Fixed function graphics to GPGPU

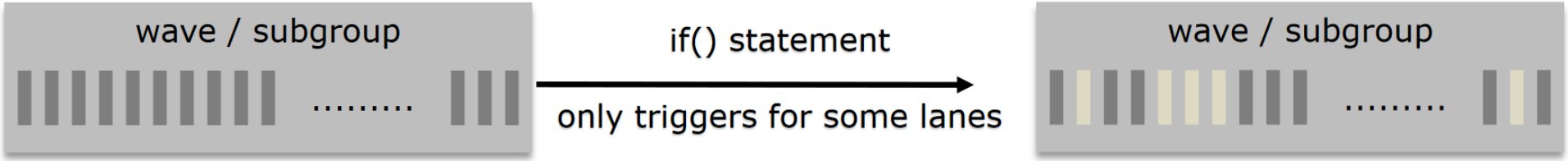


Problem Description: Divergence on wide SIMD

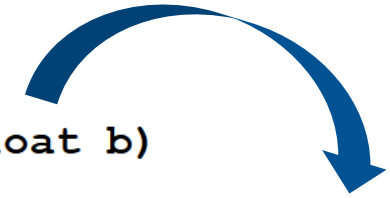


All lanes / invocations active

Subset of lanes / invocations active



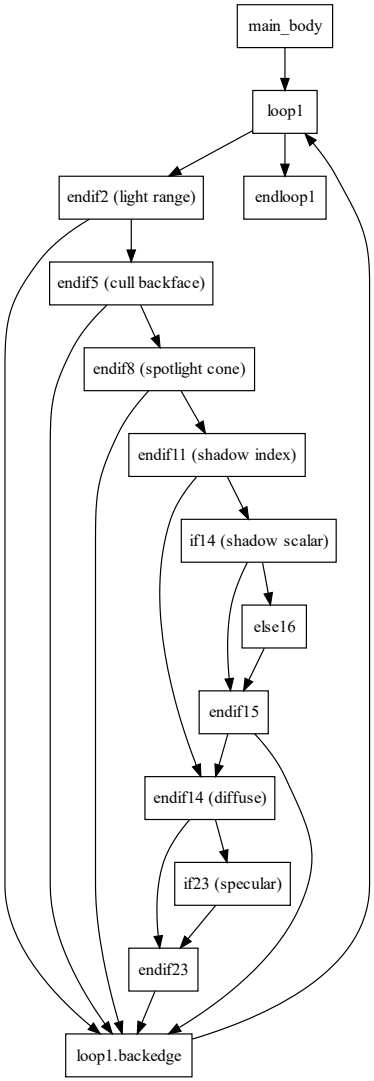
Problem Description: Converting thread-level code to wave-level ISA



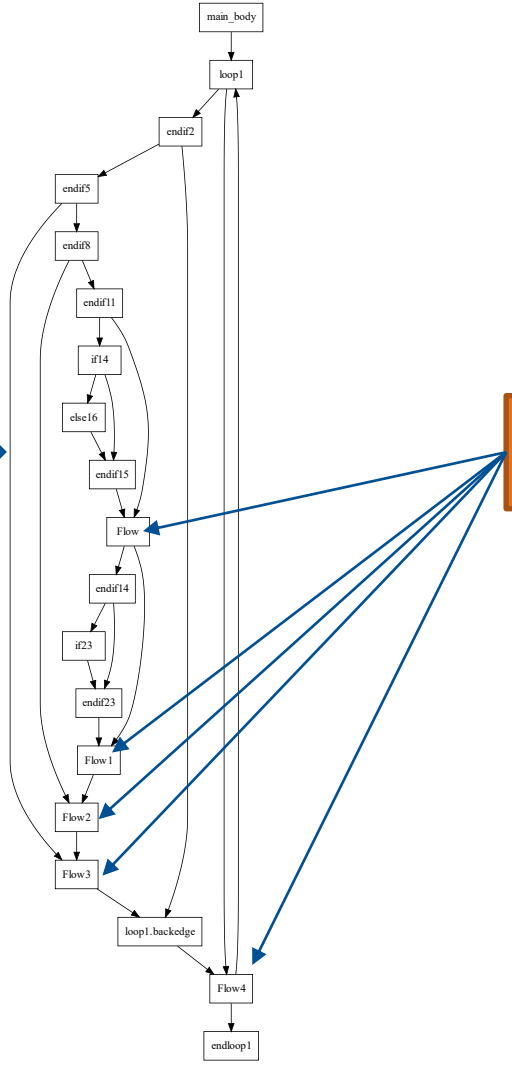
```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

   v_cmp_gt_f32     r0,r1          //a > b, establish VCC
   s_mov_b64        s0,exec        //Save current exec mask
   s_and_b64        exec,vcc,exec  //Do "if"
   s_cbranch_vccz   label0         //Branch if all lanes fail
   v_sub_f32        r2,r0,r1       //result = a - b
   v_mul_f32        r2,r2,r0       //result=result * a

label0:
   s_andn2_b64      exec,s0,exec   //Do "else"(s0 & !exec)
   s_cbranch_execz  label1         //Branch if all lanes fail
   v_sub_f32        r2,r1,r0       //result = b - a
   v_mul_f32        r2,r2,r1       //result = result * b
label1:
   s_mov_b64        exec,s0        //Restore exec mask
```

Problem Description: Structurization in LLVM



StructurizeCFG pass



Unnecessary flow blocks

Related Works

Hardware Reconvergence (Stacks):

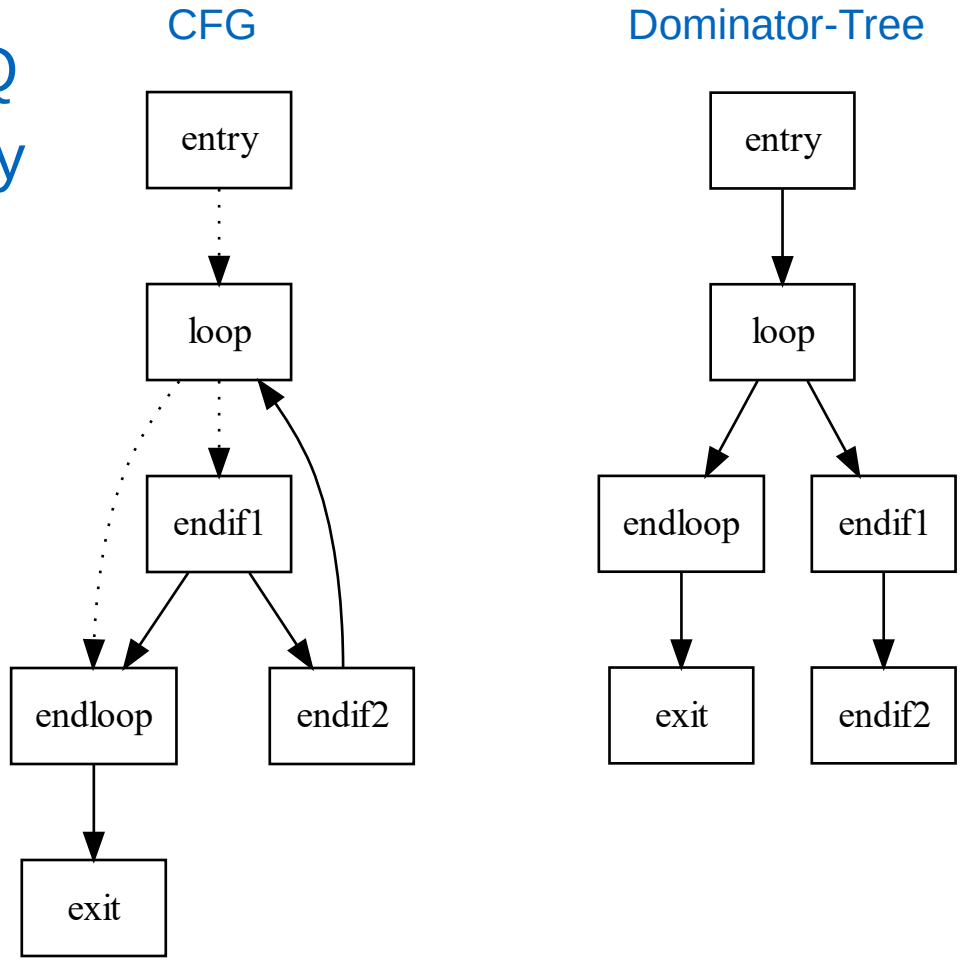
- Thread frontiers (Intel)
- Simultaneous Branch/Warp Interweaving (Nvidia)

Software:

- Partial Control flow linearization
- Taming Control Divergence in GPUs through Control Flow Linearization

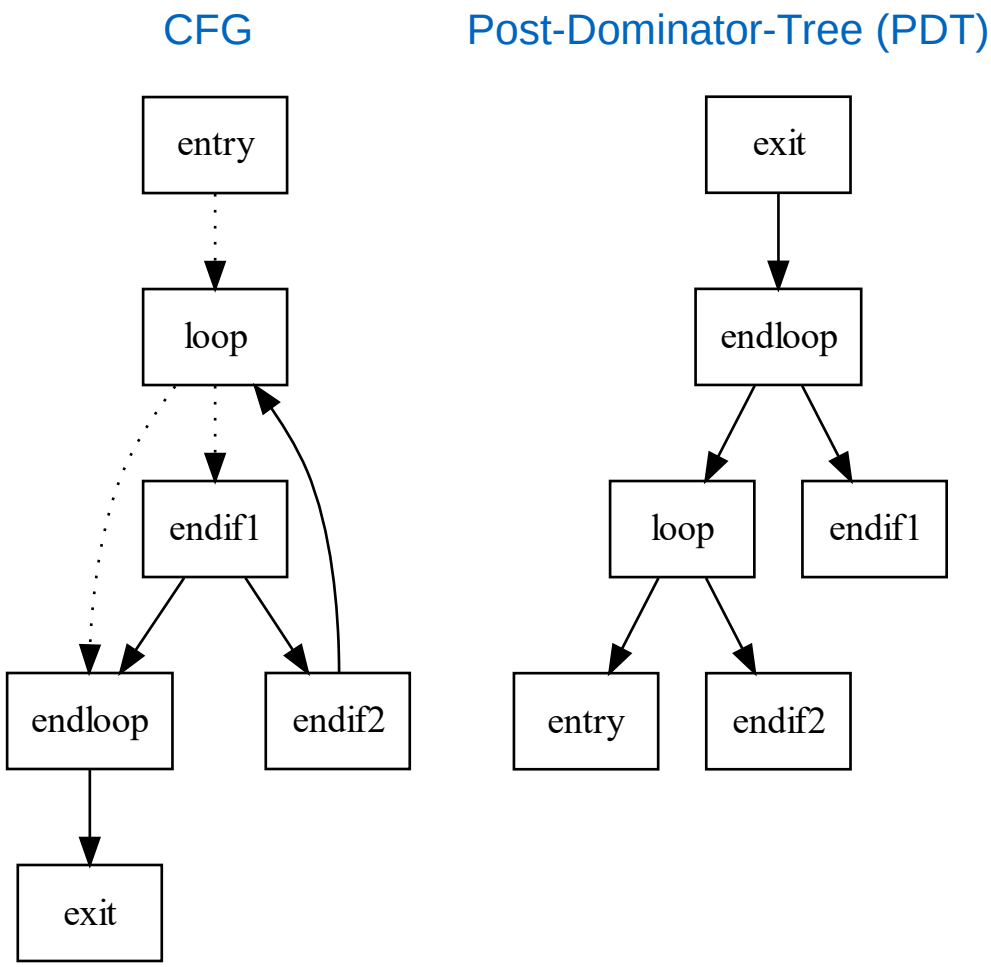
Dominator Trees

- Node P dominates node Q if every path from the entry node has to pass through P to connect to Q.



Post Dominator Trees

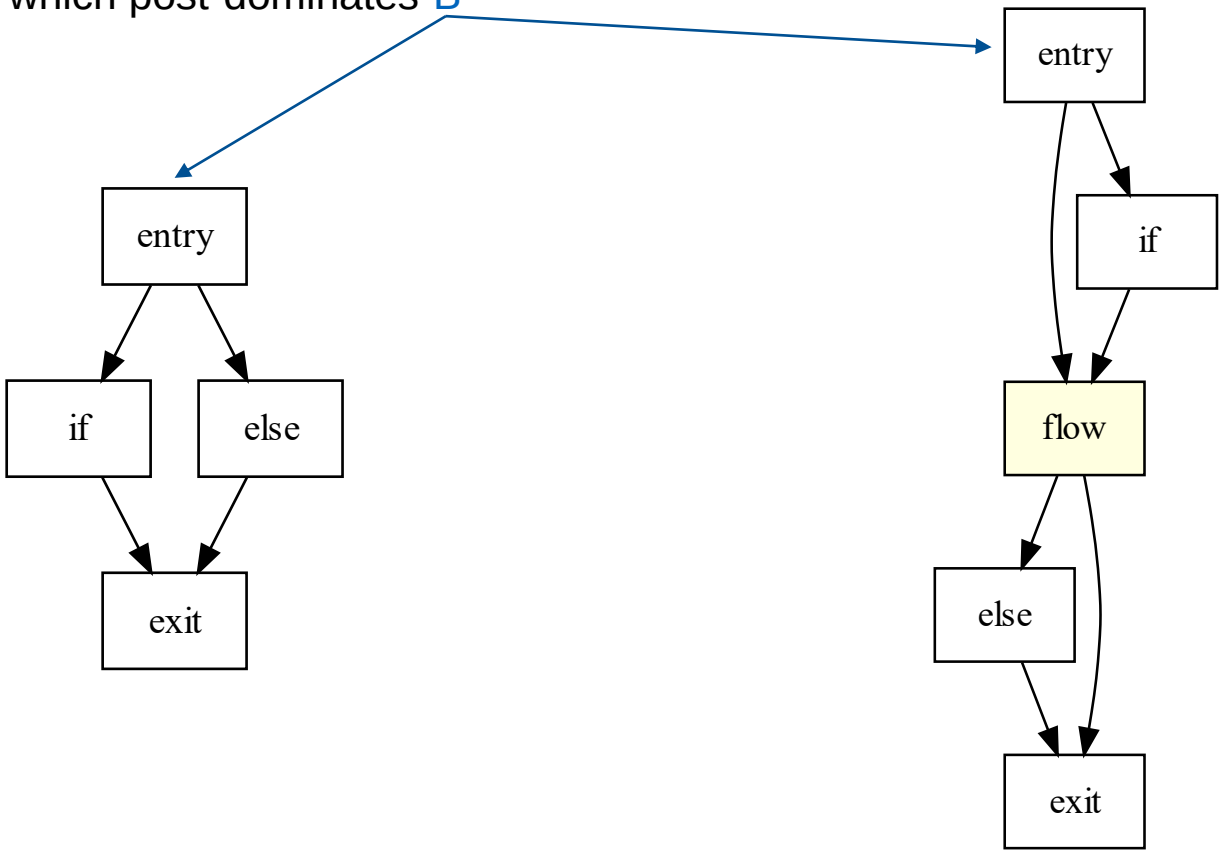
- Node P post-dominates node Q if every path from Q to the exit node has to through P.



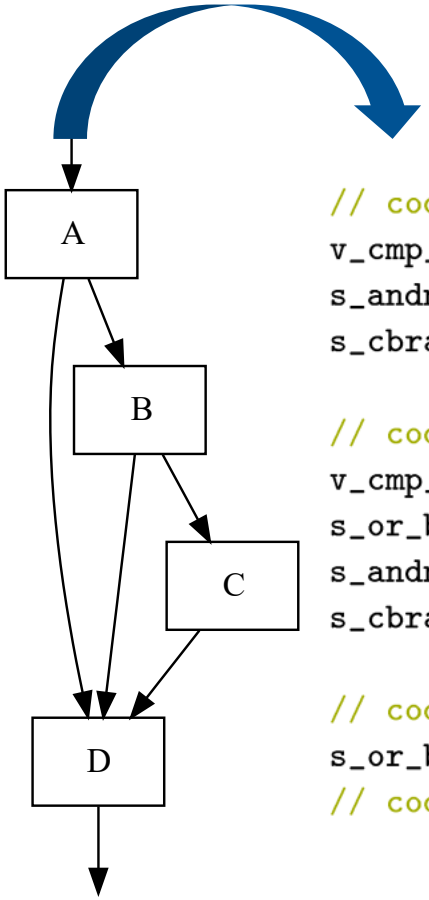
Reconverging CFGs

Definition:

- Every non-uniform terminator B (conditional branch) has exactly two successors
- One of which post-dominates B



Lowering to wave-level

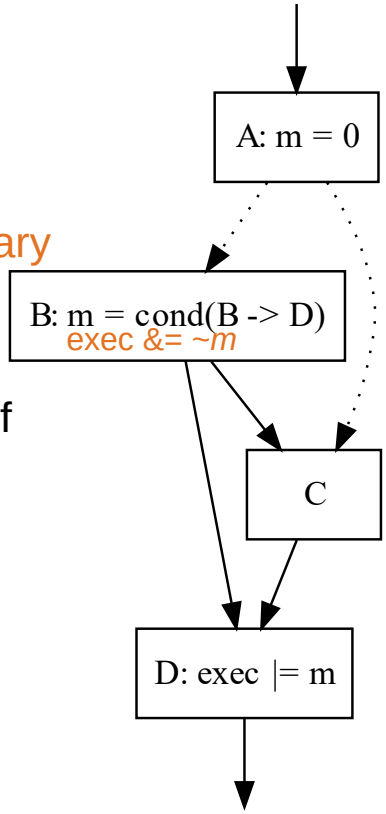



```
// code for A
v_cmp_???       s[0:1], ...          // initialize re-join mask in m
s_andn2_b64     exec, exec, s[0:1]   // subtract re-join mask stored in s[0:1]
s_cbranch_execz D

// code for B
v_cmp_???       vcc, ...             // condition for jumping to D
s_or_b64        s[0:1], vcc          // accumulate re-join mask in m
s_andn2_b64     exec, exec, vcc      // subtract re-join mask from execution mask
s_cbranch_execz D

// code for C
s_or_b64        exec, exec, s[0:1]   // add previously subtracted re-join masks
// code for D
```

Lowering Reconverging CFGs

For each conditional non-uniform node N:

- Virtual register *m* holds re-join mask for basic block N (containing diverging threads branching to the primary successor)
- Subtract *m* from the exec register and direct control flow to secondary successor by placing its instructions directly after basic block N
- Re-join mask is used to update the exec register at the beginning of the primary successor
- *m* must be correctly initialized (at entry points to divergent flow) to avoid unrelated data being merged into the execution mask

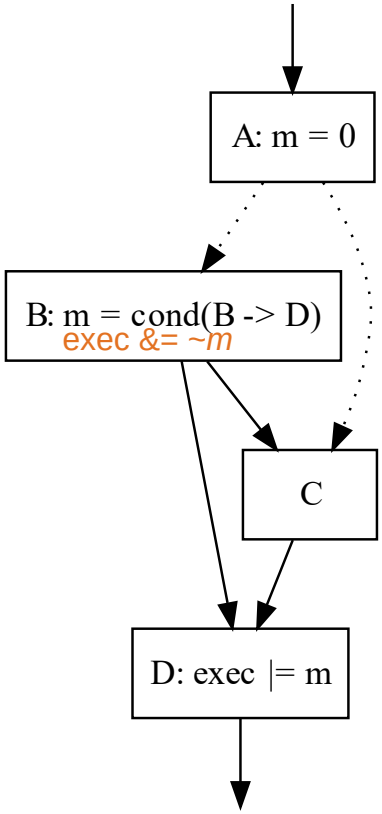


Lowering Reconverging CFGs

For each unprocessed primary successor N in the CFG:

- Decide how to initialize or modify mask *m* for incoming edges
- Decide how to update the exec mask register (binary op)
- Insert scalar branch instruction to skip N if exec is empty
  - (might skip mask initialization instructions of previous basic blocks)

✓ Mark N as processed



Lowering Reconverging CFGs

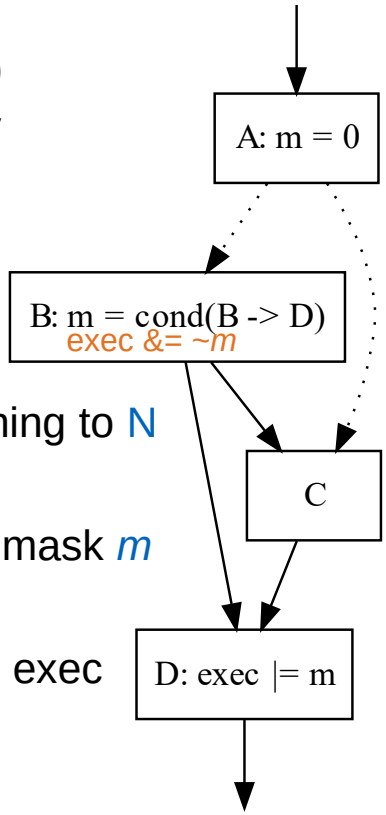
Decide how to initialize or modify mask *m* for incoming edges:

- P: Inspect predecessors for which N is the primary successor (B)
- P*: Inspect (direct) successors of nodes in P, capturing secondary successors and N itself (B, C, D)

For every node X in P

- If X has a predecessor Y not in P* (e.g. X = B, Y = A)
  - ➢ Insert MOVE instruction at X to set *m* to the condition of X branching to N
- Else (nested primary)
  - ➢ Insert OR instruction at X to add condition of X branching to N to mask *m*

- ➢ Insert AND-NOT in X to subtract condition of X branching to N from exec
- ➢ Insert scalar branch to N if exec register is empty



Lowering Reconverging CFGs

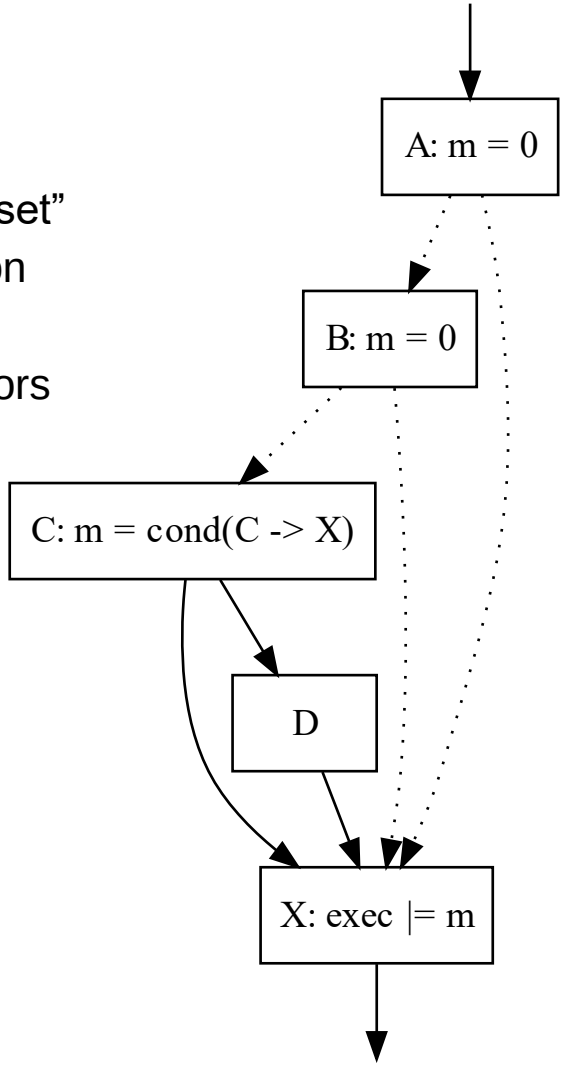
Initialize mask *m* for incoming uniform- or backward edges:

- Need to identify entry nodes not included in the “reachable set” to avoid uninitialized mask *m* corrupting thread join operation

- P*: successors of nodes in P, capturing secondary successors and N itself: “reachable set” (C, D, X)

- Q: predecessors of “reachable set” P* which have predecessor that are not part of the reconverging subgraph

  - Predecessors of (D, X) not part of (C, D ,X) that are not covered by initialization at the entry node (C: A and B)
  - Insert MOVE instruction at nodes in Q, setting *m* to 0



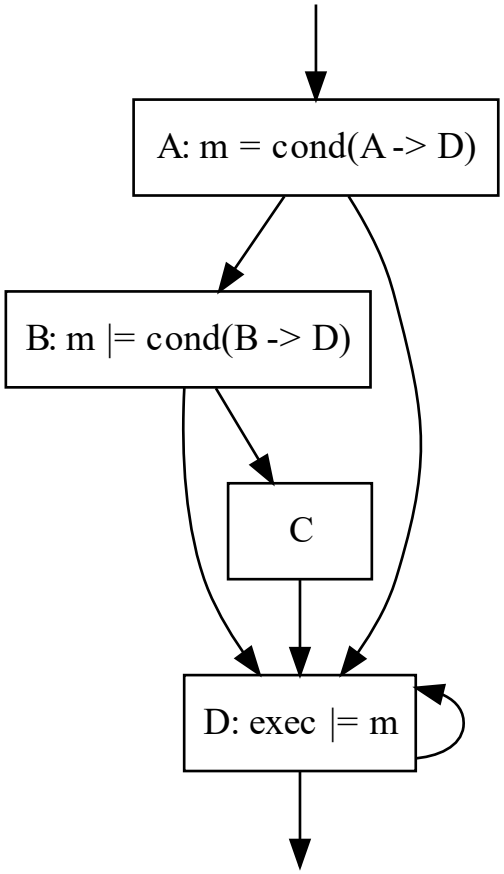
Lowering Reconverging CFGs

Initialize mask *m* for incoming uniform- or backward edges:

- Need to identify entry nodes not included in the “reachable set” to avoid uninitialized mask *m* corrupting thread join operation

- Q: predecessors of “reachable set” P* which have predecessor that are not part of the reconverging subgraph

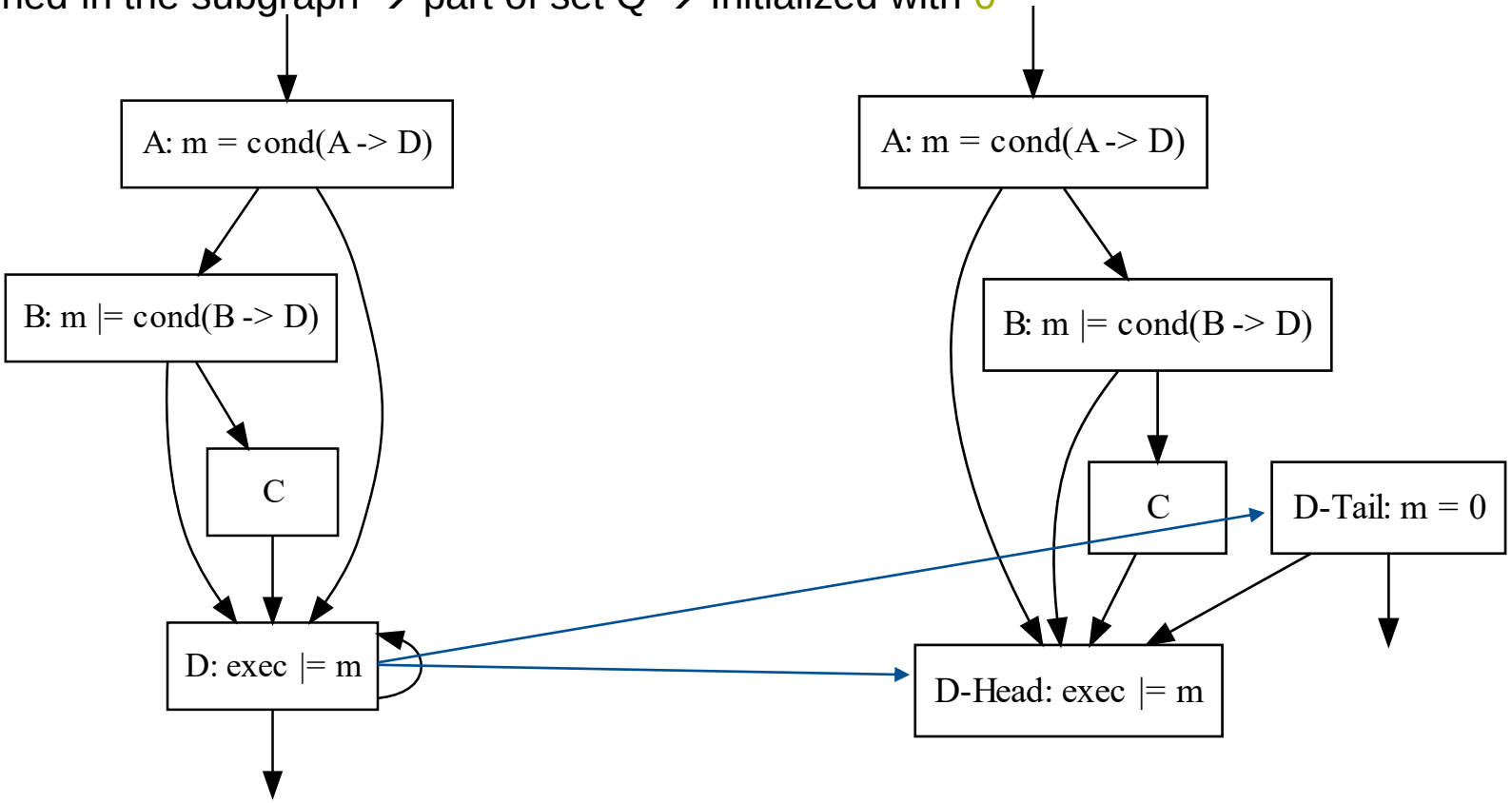
- Q and the node X for which N is the primary successor cover all nodes entering the flow of a particular reconverging subgraph



Lowering Reconverging CFGs

Initialize mask *m* for incoming uniform- or backward edges:

- Split primary successor with backward edge so that the backward edge is a predecessor not contained in the subgraph → part of set Q → Initialized with 0



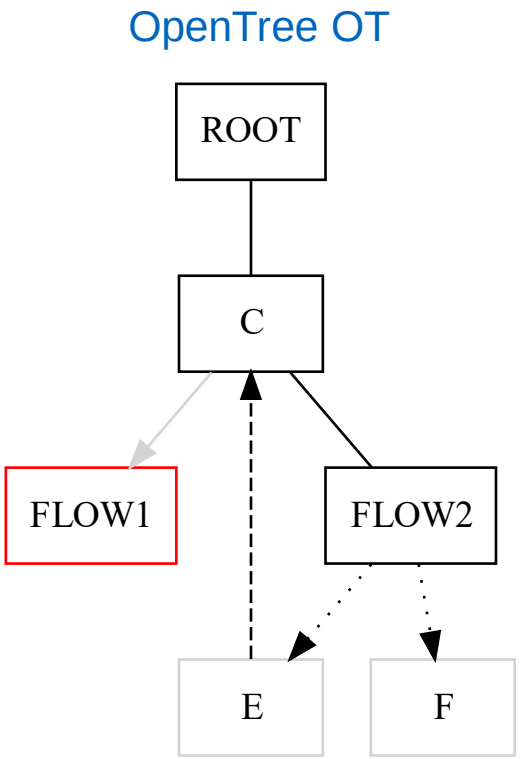
Transforming to reconverging control flow

Approach:

- Compute basic block ordering in which to process input CFG
- Maintain *open tree* OT structure containing unprocessed “open” edges to reroute control flow towards the exit node by inserting new *flow* blocks

Ordering:

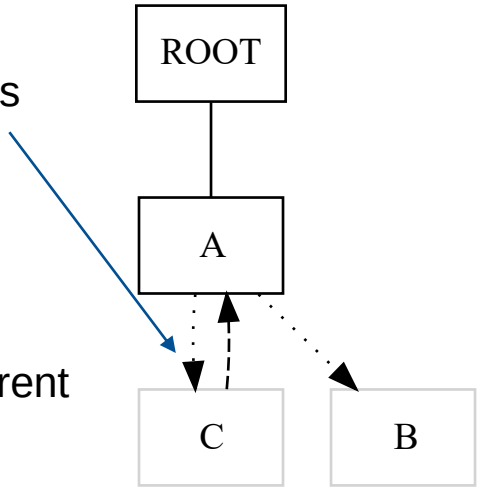
- Any ordering is viable as long as the exit node comes last
- Ordering is based on traversal of the input CFG
- Quality of reconverging CFG depends on input basic block ordering



Transforming to reconverging control flow

Processing nodes:

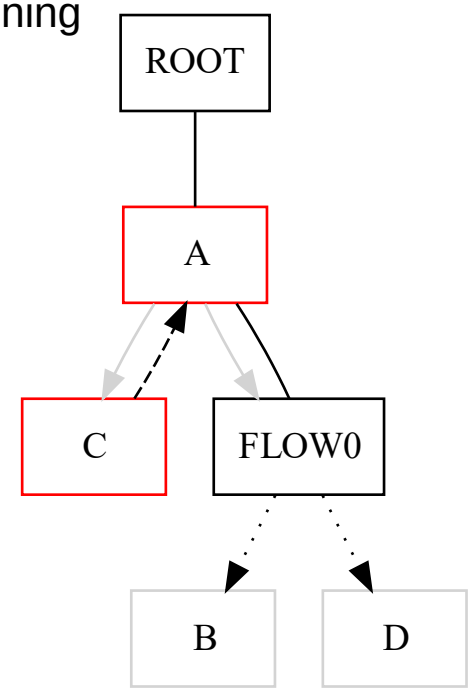
- Nodes of the OT have sets of open *Incoming* and *Outgoing* edges (to nodes of the OT, not CFG) that need to be processed
- A node can be closed if both sets are emptied by processing
- Closed nodes are removed and their child nodes moved to its parent



Transforming to reconverging control flow

Adding node B to OT:

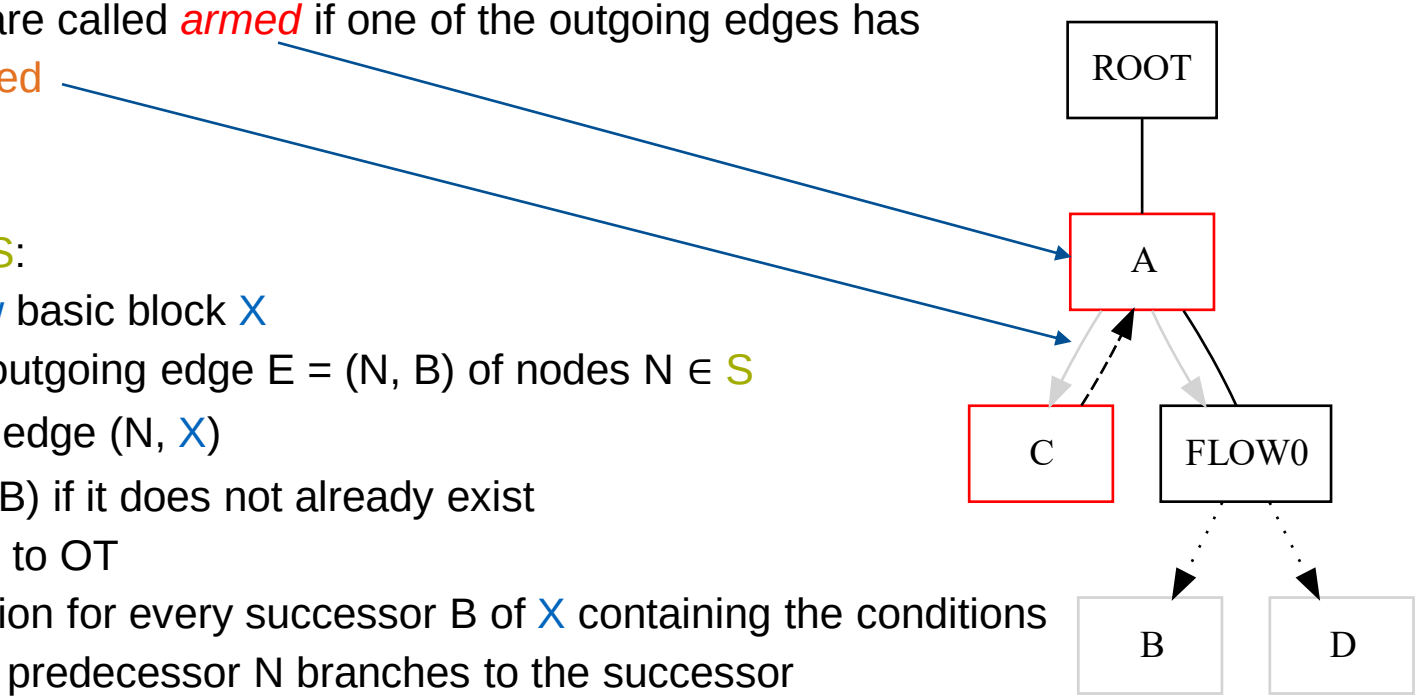
- Collapse all paths leading from the root to B into a single path containing all its visited predecessors
- Attach B as child to the lowest predecessor
- Close all open edges from visited predecessors leading to B
- Mark B as visited



Transforming to reconverging control flow

Rerouting:

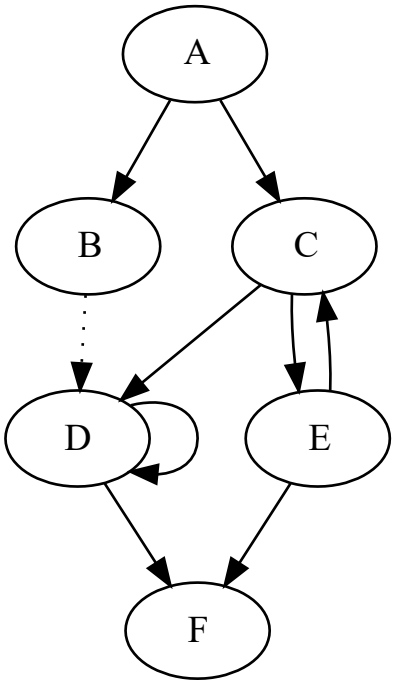
- Divergent nodes are called *armed* if one of the outgoing edges has already been closed



- Reroute Subtree S:
  - Create new flow basic block X
  - For each open outgoing edge E = (N, B) of nodes N ∈ S
    - Replace E by edge (N, X)
    - Add edge (X, B) if it does not already exist
- Add basic block X to OT
- Add a Phi-instruction for every successor B of X containing the conditions under which each predecessor N branches to the successor

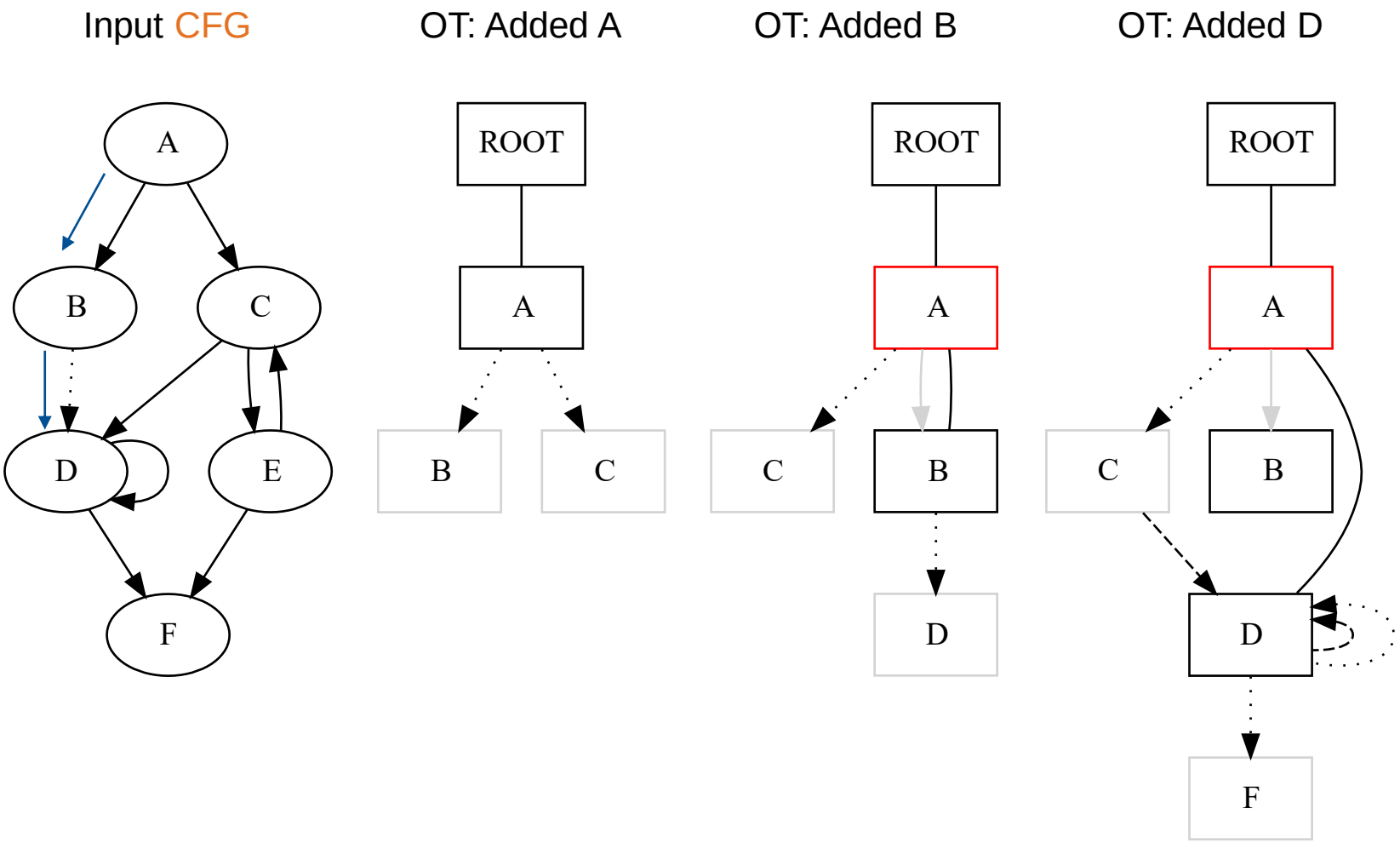
Transforming to reconverging control flow

Input CFG



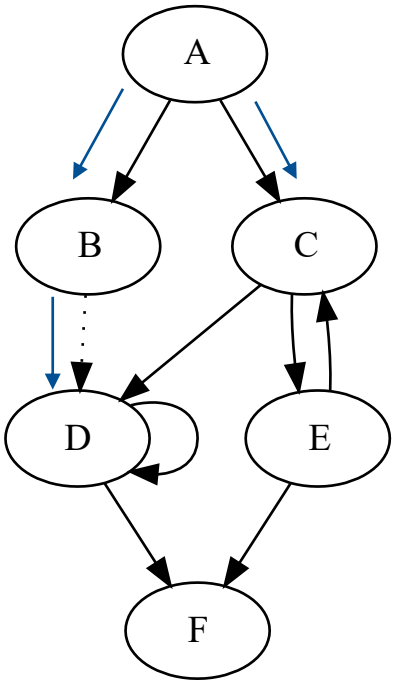

```
define amdgpu_ps void @inputcfg(i32 %in_A, i32 %in_C, i32 %in_D, i32 %in_E)
{
A:
        %cc_A = icmp eq i32 %in_A, 0
        br i1 %cc_A, label %B, label %C
B:
        br label %D
C:
        %cc_C = icmp eq i32 %in_C, 0
        br i1 %cc_C, label %D, label %E
D:
        %cc_D = icmp eq i32 %in_D, 0
        br i1 %cc_D, label %D, label %F
E:
        %cc_E = icmp eq i32 %in_E, 0
        br i1 %cc_E, label %C, label %F
F:
        ret void
}
```

Transforming to reconverging control flow

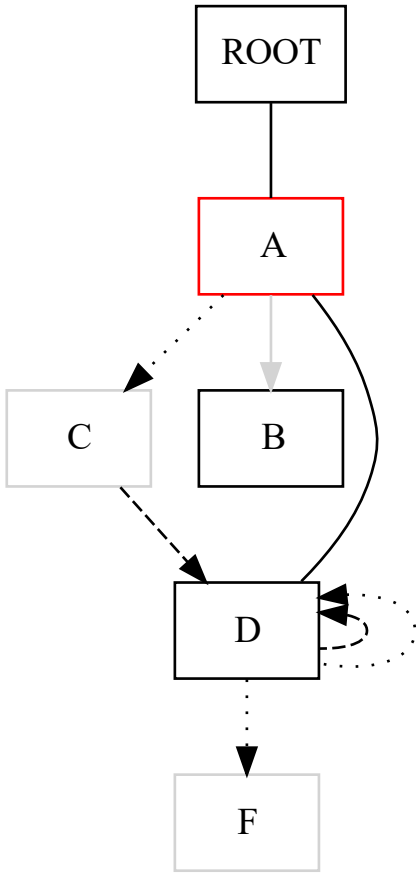


Transforming to reconverging control flow

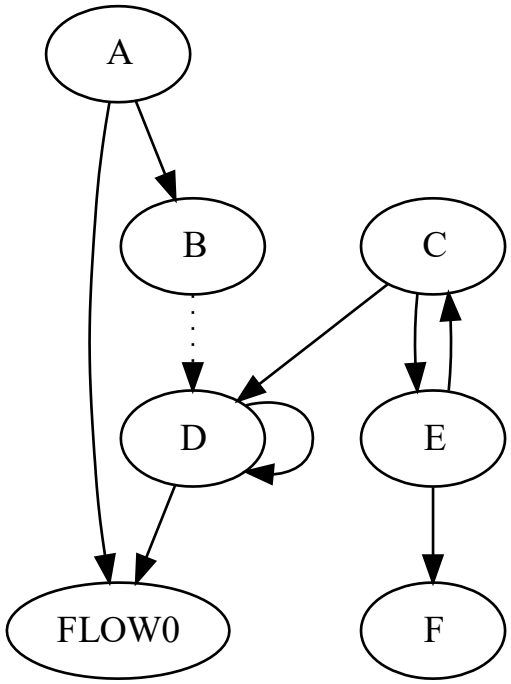
Input CFG



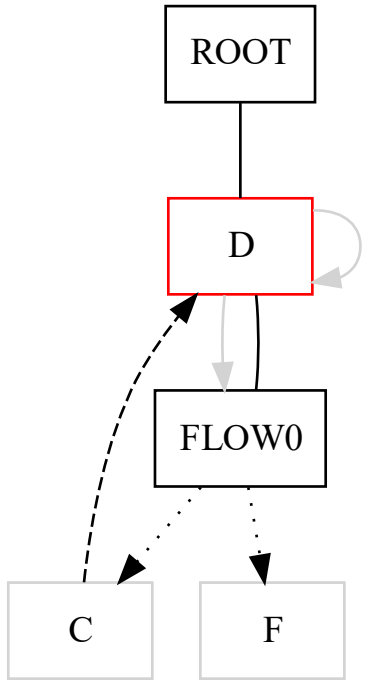
OT: Added D



CFG: Reroute Condition 1

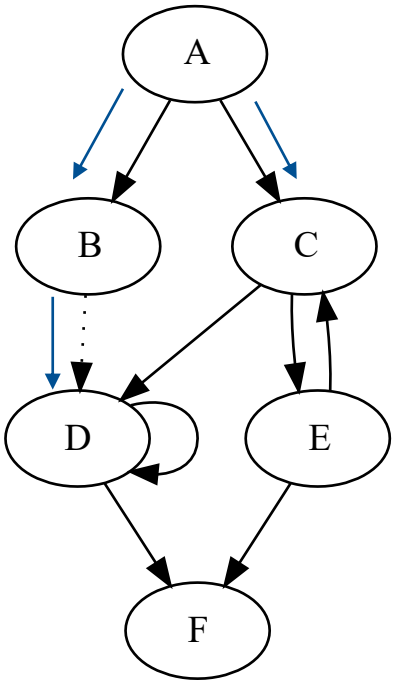


OT: Added FLOW0

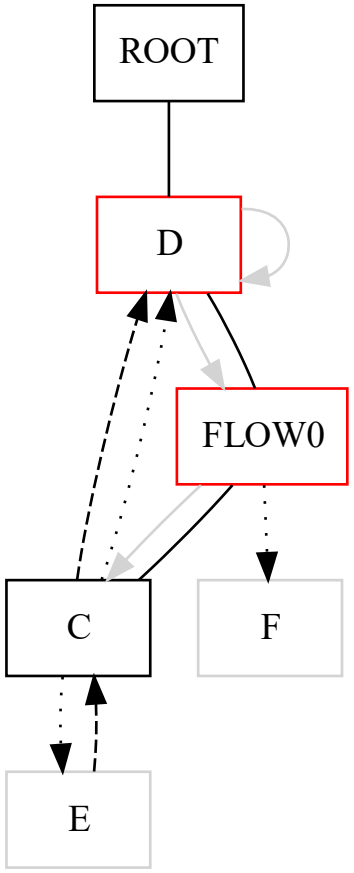


Transforming to reconverging control flow

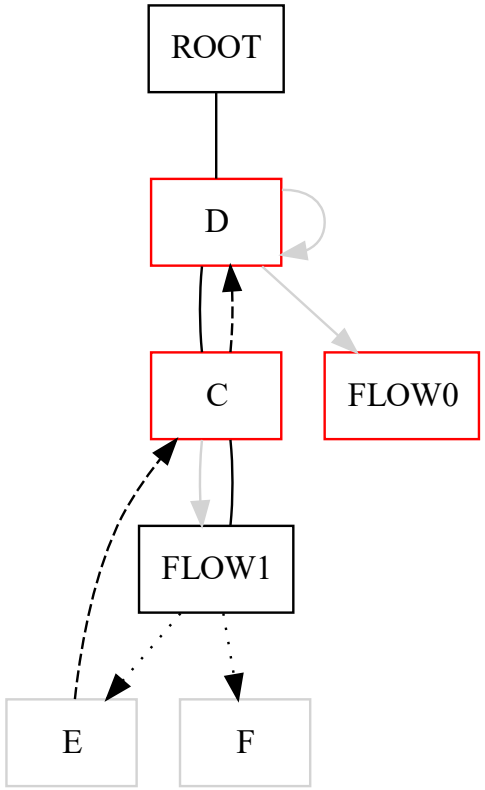
Input CFG



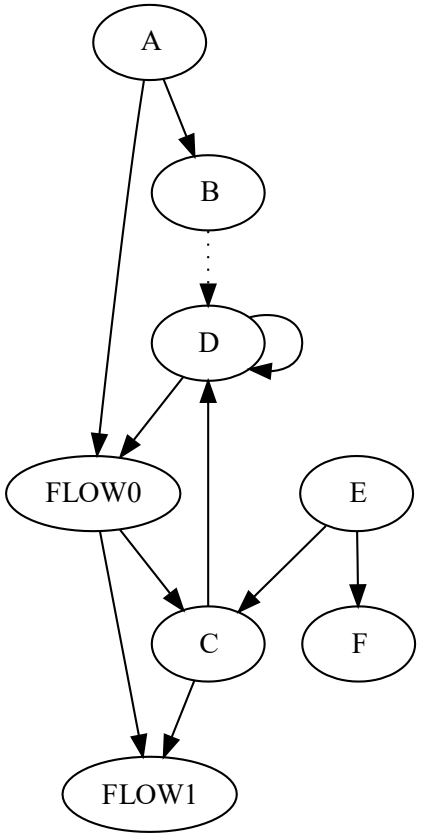
OT: Added C



OT: Added FLOW1

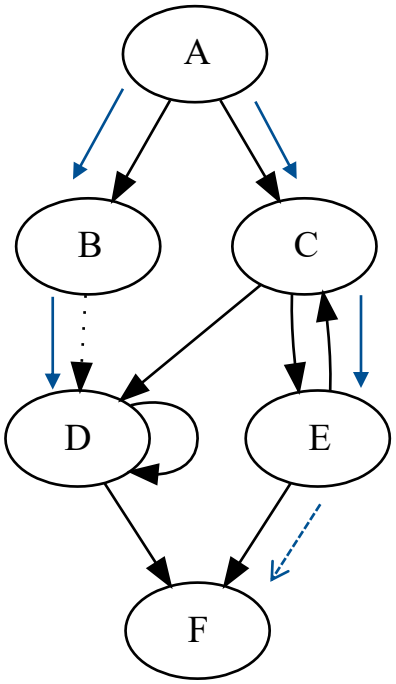


CFG: Rerouting Condition 2

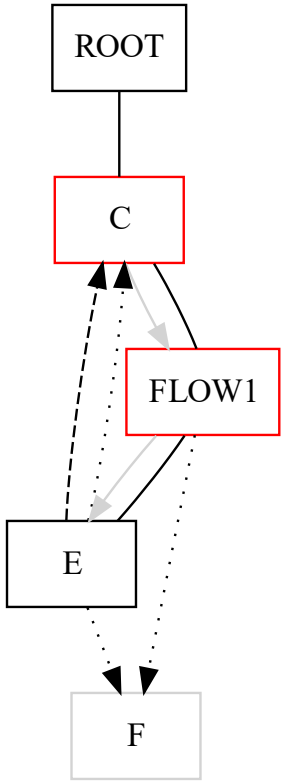


Transforming to reconverging control flow

Input CFG

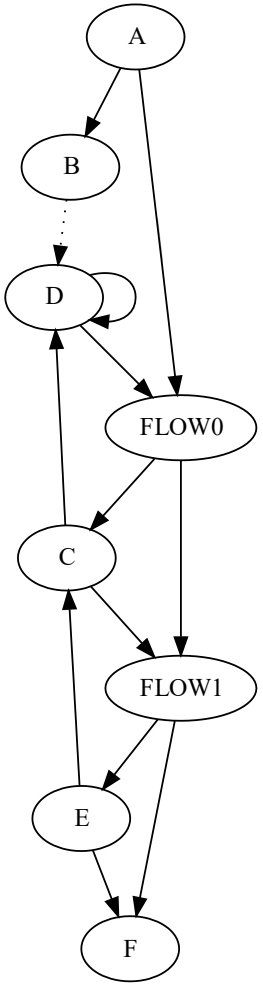


OT: Added E



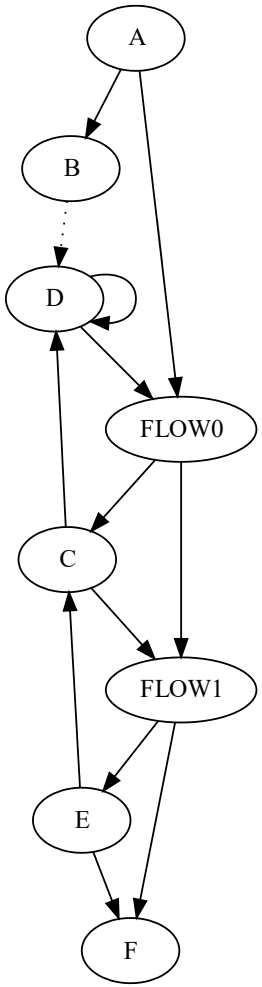
OT: Added F

CFG: After F



Transforming to reconverging control flow

Reconverging CFG:




```
%cc_A = icmp eq i32 %in_A, 0
br i1 %cc_A, label %B, label %flow0
```

```
br label %D
```

```
%cc_D = icmp eq i32 %in_D, 0
br i1 %cc_D, label %D, label %flow0
```

```
%1 = phi i1 [true, %A], [false, %D]
%2 = phi i1 [false, %A], [true, %D]
br i1 %1, label %C, label %flow1
```

```
%cc_D = icmp eq i32 %in_D, 0
br i1 %cc_D, label %D, label %flow0
```

```
%0 = phi i1 [%2, %flow0], [false, %C]
br i1 %0, label %F, label %E
```

```
%cc_E = icmp eq i32 %in_E, 0
br i1 %cc_E, label %C, label %F
```

```
ret void
```

Reconverging Control-Flow Graphs

Contributions:

- New SPMD vectorization approach
- Simple and concise definition of Reconvergence for CFGs (weaker than structuredness)
- Proof-of-Concept lowering algorithm and CFG transformation

Properties:

- Support for unstructured and irreducible input CFGs
- Preserve uniform control flow
- Retain CFGs that are already reconverging
- Insert fewer new basic blocks than structurization requires (StructurizeCFG)