DIVERGENCE ANALYSIS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

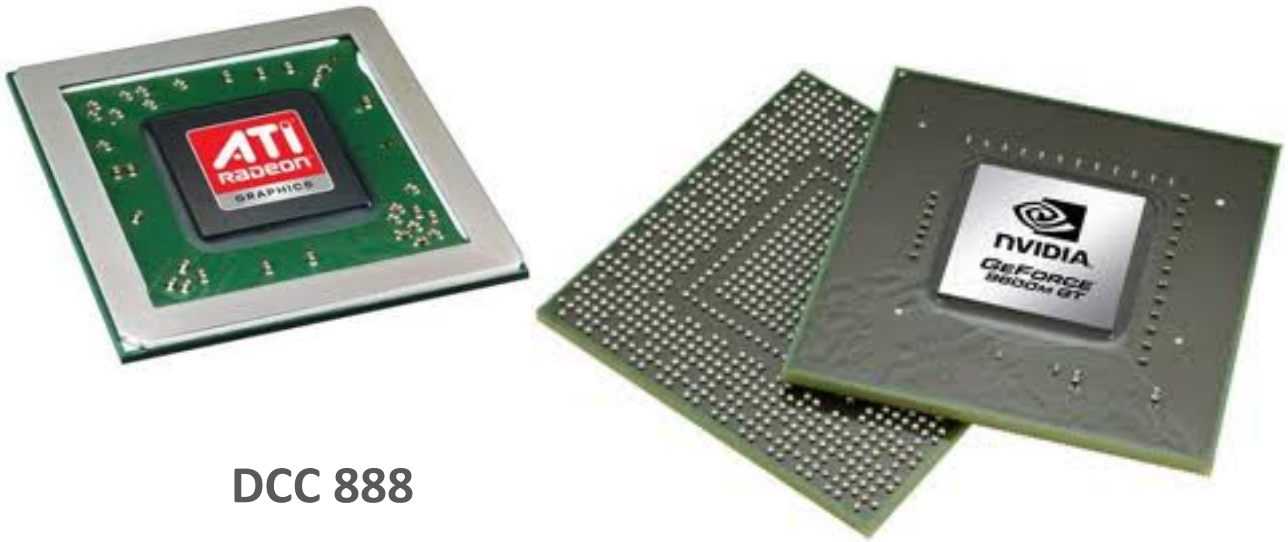
fernando@dcc.ufmg.br

The material in these slides has been taken from two different papers, [Coutinho11] and [Sampaio13]. See "A bit of History" in the last slide

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

What are Graphics Processing Units



DCC 888

The Wheel of Reincarnation

A *scandalously* brief history of GPUs

- In the good old days, the graphics hardware was just the VGA. **All the processing was in software**.

1) Do you know how the frame buffer works?

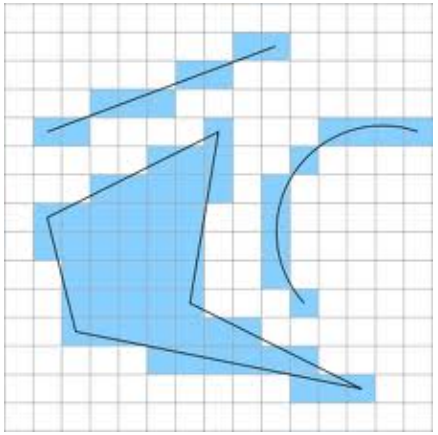
2) Can you program the VGA standard in any way?

- People started complaining: software is slow…
  - But, what do you want to run at the hardware level?

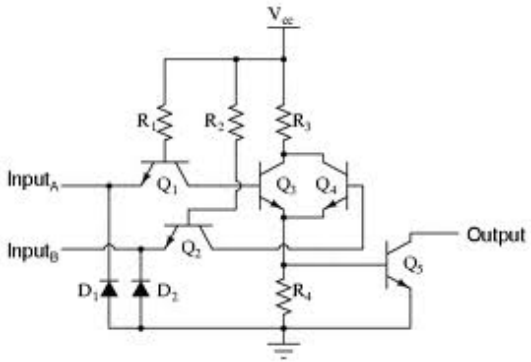
The Wheel of Reincarnation

A *scandalously* brief history of GPUs

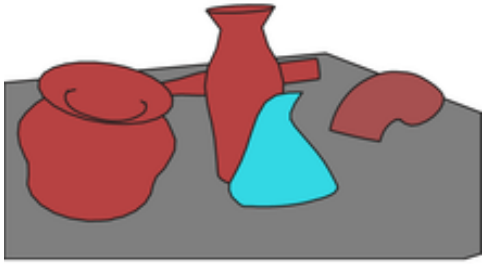
- Some functions, like the rasterizer, are heavily used. What is rasterization?
- Better to **implement these functions in hardware**.



1) How can we implement a function at the hardware level?
2) What is the advantage of implementing a function at the hardware level?
3) Is there any drawback?
4) Can we program (in any way) this hardware used to implement a specific function?



Graphics Pipeline



- Graphics can be processed in a **pipeline**.
  - Transform, project, clip, display, etc…
- Some functions, although different, can be implemented by very similar hardware.

> Shading is an example. Do you know what is a shader?

- Add a graphics API to program the shaders.
  - But this API is so specific… and the hardware is so powerful… what a waste!

A *scandalously* brief history of GPUs

General Purpose Hardware

A *scandalously* brief history of GPUs

- Let's add an **instruction set** to the shader.
  - Let's augment this hardware with **general purpose** integer operations.
  - What about adding some **branching** machinery too?
- Hum… add a **high level language** on top of this stuff.
  - Plus a lot of documentation. Advertise it! It should look cool!
- Oh boy: we have now two general purpose processors.
  - We should unify them. The rant starts all over again…

1.5 turns around the wheel

A *scandalously* brief history of GPUs

- Let's add a display processor to the display processor
  - After all, there are some operations that are really *specific*, and *performance* critical…

Dedicated rasterizer

Brief Timeline

A *scandalously* brief history of GPUs

Year	Transistors	Model	Tech
1999	25M	GeForce 256	DX7, OpenGL
2001	60M	GeForce 3	Programmable Shader
2002	125M	GeForce FX	Cg programs
2006	681M	GeForce 8800	C for CUDA
2008	1.4G	GeForce GTX 280	IEEE FP
2010	3.0G	Fermi	Cache, C++

Computer Organization

- GPUs show different types of parallelism
  - Single Instruction Multiple Data (SIMD)
  - Single Program Multiple Data (SPMD)
- In the end, we have a **MSIMD** hardware.

1) Why are GPUs so parallel?
2) Why traditional CPUs do not show off all this parallelism?

We can think of a SIMD hardware as a firing squad: we have a captain, and a row of soldiers. The captain issues orders, such as set, aim, fire! And all the soldiers, upon hearing one of these orders, performs an action. They all do the same action, yet, they use different guns and bullets.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

The Programming Environment

An *outrageously* concise overview of the programming mode.

- There are two main programming languages used to program graphics processing units today: OpenCL and C for CUDA
- These are not the first languages developed for GPUs. They came after Cg or HLSL, for instance.
  - But they are much more general and expressive.
- We will focus on C for CUDA.
- This language lets the programmer explicitly write code that will run in the CPU, and code that will run in the GPU.
  - It is a heterogeneous programming language.

From C to CUDA in one Step

An *outrageously* concise overview of the programming mode.

```
void saxpy_serial(int n, float alpha, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = alpha*x[i] + y[i];
}
// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);
```

- This program, written in C, performs a typical vector operation, reading two arrays, and writing on a third array.
- We will translate this program to C for CUDA.

1) What is the asymptotic complexity of this program?
2) How much can we parallelize this program? In a world with many – really many – processors, e.g., the PRAM world, what would be the complexity of this program?

The first Cuda program

An *outrageously* concise overview of the programming mode.

```
void saxpy_serial(int n, float alpha, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = alpha*x[i] + y[i];
}
// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);
```

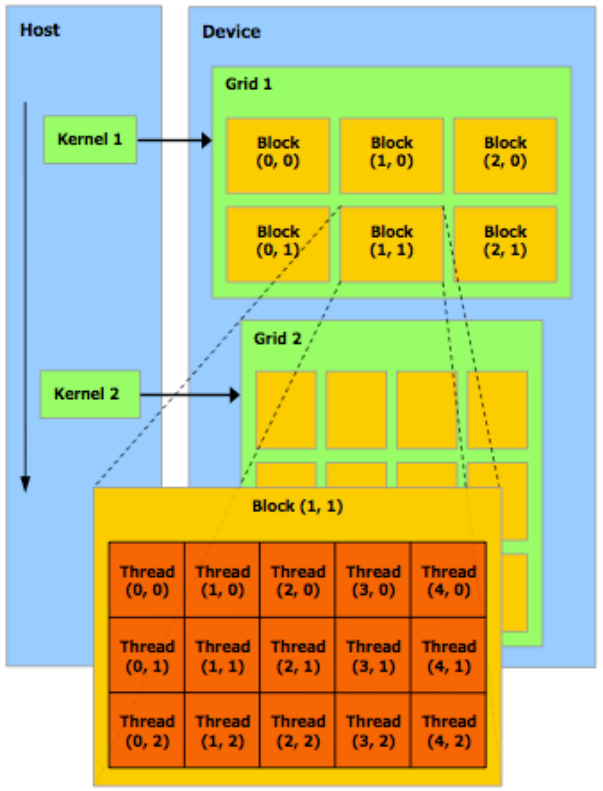
What happened to the **loop** in the **CUDA** program?

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Understanding the Code

An *outrageously* concise overview of the programming mode.

- Threads are grouped in warps, blocks and grids
- Threads in different grids do not talk to each other
  - Grids are divided in blocks
- Threads in the same block share memory and barriers
  - Blocks are divided in warps
- Threads in the same warp follow the SIMD model.




```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Raising the level

An *outrageously* concise overview of the programming mode.

- Cuda programs contain **CPU programs** plus **kernels**
- Kernels are called via a special syntax:

```
kernel<<<dGrd, dBck>>>(A,B,w,C);
```

- The C part of the program is compiled as traditional C.
- The kernel part is first translated into PTX, and then this high level assembly is translated into SASS.

```
__global__ void matMul1(float* B, float* C, float* A, int w) {
 float Pvalue = 0.0;

 for (int k = 0; k < w; ++k) {
  Pvalue += B[threadIdx.y * w + k] * C[k * w + threadIdx.x];
 }

 A[threadIdx.x + threadIdx.y * w] = Pvalue;
}
void Mul(const float* A, const float* B, int width, float* C) {
  int size = width * width * sizeof(float);

  // Load A and B to the device
  float* Ad;
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  float* Bd;
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

  // Allocate C on the device
  float* Cd;
  cudaMalloc((void**)&Cd, size);

  // Compute the execution configuration assuming
  // the matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, width, Cd);

  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}
```

Lowering the level

An *outrageously* concise overview of the programming mode.

- CUDA assembly is called Parallel Thread Execution (PTX)

What do you think an assembly language for parallel programming should have?

```
__global__ void
saxpy_parallel(int n, float a, float *x,
                              float *y) {

  int i = bid.x * bid.x + tid.x;
  if (i < n)
      y[i] = a * x[i] + y[i];

}
```

```
.entry saxpy_GPU (n, a, x,  y) {
        .reg .u16 %rh<4>;
        .reg .u32 %r<6>;
        .reg .u64 %rd<8>;
        .reg .f32 %f<6>;
        .reg .pred %p<3>;
$LBB1__Z9saxpy_GPUifPfS_:
        mov.u16         %rh1, %ctaid.x;
        mov.u16         %rh2, %ntid.x;
        mul.wide.u16    %r1, %rh1, %rh2;
        cvt.u32.u16     %r2, %tid.x;
        add.u32         %r3, %r2, %r1;
        ld.param.s32    %r4, [n];
        setp.le.s32     %p1, %r4, %r3;
        @%p1 bra        $Lt_0_770;
        .loc    28      13      0
        cvt.u64.s32     %rd1, %r3;
        mul.lo.u64      %rd2, %rd1, 4;
        ld.param.u64    %rd3, [y];
        add.u64         %rd4, %rd3, %rd2;
        ld.global.f32   %f1, [%rd4+0];
        ld.param.u64    %rd5, [x];
        add.u64         %rd6, %rd5, %rd2;
        ld.global.f32   %f2, [%rd6+0];
        ld.param.f32    %f3, [alpha];
        mad.f32         %f4, %f2, %f3, %f1;
        st.global.f32   [%rd4+0], %f4;
        exit;
}
```

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

DIVERGENCES

DCC 888

SIMD: The Good and the Bad

Control flow divergences for dummies and brighties

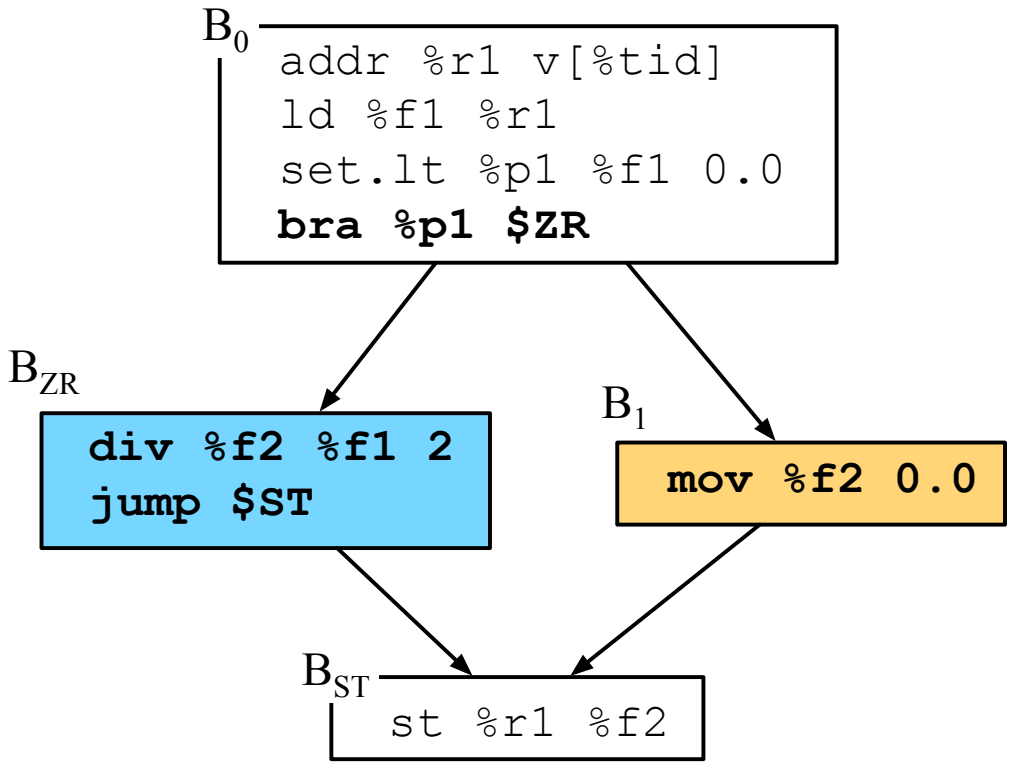
- The Single Instruction, Multiple Data execution model is more parsimonious:
  - Less power dissipation
  - Less space spent on the decoder
- These advantages make it very good for regular applications.
- Yet, not all the applications are so regular, and divergences may happen.
  - Memory divergences.
  - **Control flow divergences**.

Example of Divergent CFG

- Below we have a simple *kernel*, and its Control Flow Graph:

```
__global__ void
ex (float* v) {
  if (v[tid] < 0.0) {
    v[tid] /= 2;
  } else {
    v[tid] = 0.0;
  }
}
```

How/When can we have divergences in this kernel?



What do divergences cost?

Control flow divergences for dummies and brighties

program counter	label	op	def	use_1	use_2	ALU_1	ALU_2
1	B_0	addr	%r1	v	[%tid]	✓	✓
2		ld	%f1	%r1		✓	✓
3		set.lt	%p1	%f1	0.0	✓	✓
4		bra	%p1	$ZR		✓	✓
5	B_{ZR}	**div**	**%f2**	**%f1**	**2**	●	✓
6		**jump**		**$ST**		●	✓
7	B_1	**mov**	**%f2**	**0.0**		✓	●
8	B_{ST}	st		%r1	%f2	✓	✓

The Kernels of Samuel

Control flow divergences for dummies and brighties

- What is the best input for the kernel below?

```
__global__ void dec2zero(int* v, int N) {
  int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
  if (xIndex < N) {
    while (v[xIndex] > 0) {
      v[xIndex]--;
    }
  }
}
```

When/how does a divergence happen in this kernel?

From `decPar`, available at `http://www.dcc.ufmg.br/~fernando/classes/gpuOpt/examples.zip`.

Trying different inputs

Control flow divergences for dummies and brighties

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

5

What would be the best way to initialize the kernel, thus pleasing Samuel?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

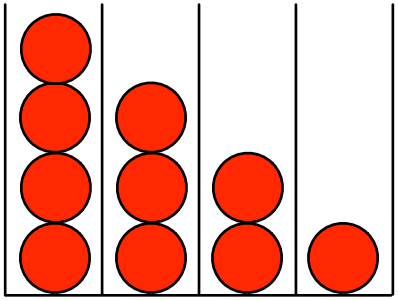
```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

Samuelic array 1



SUM: 20480000
TIME: 16250

Control flow divergences for dummies and brighties

Samuelic array 2

Control flow divergences for dummies and brighties

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

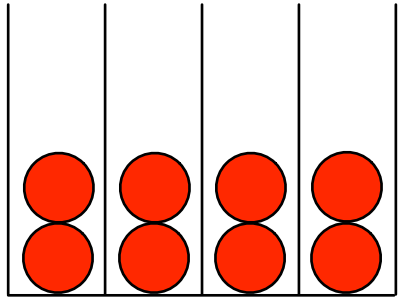
```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

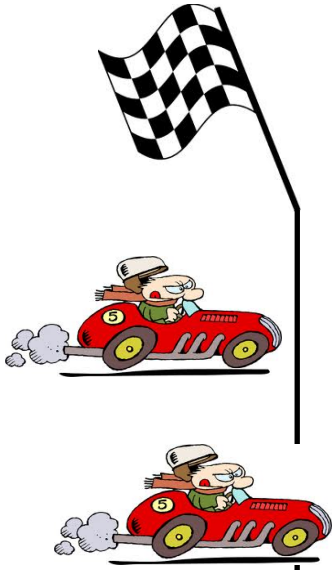
```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```



SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153



Samuelic array 3

Control flow divergences for dummies and brighties

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

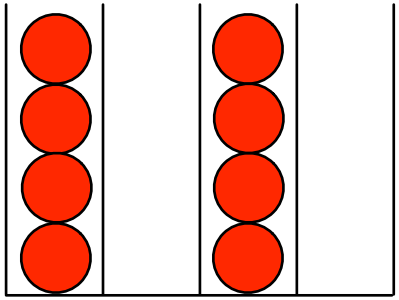
2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

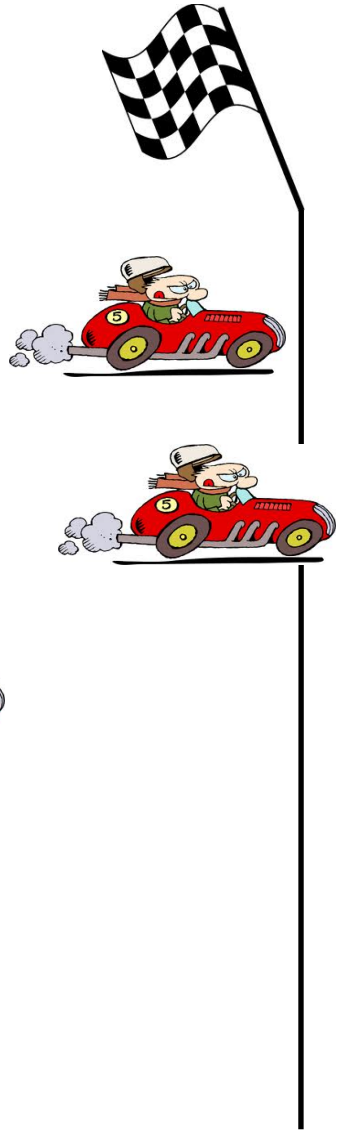
```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```



SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO

Control flow divergences
for dummies and brighties

Samuelic array 4

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

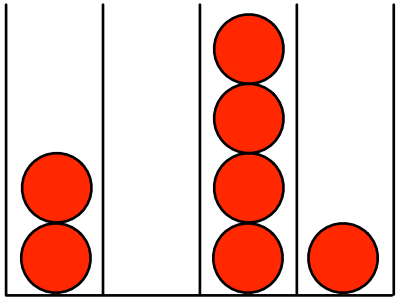
```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

3

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

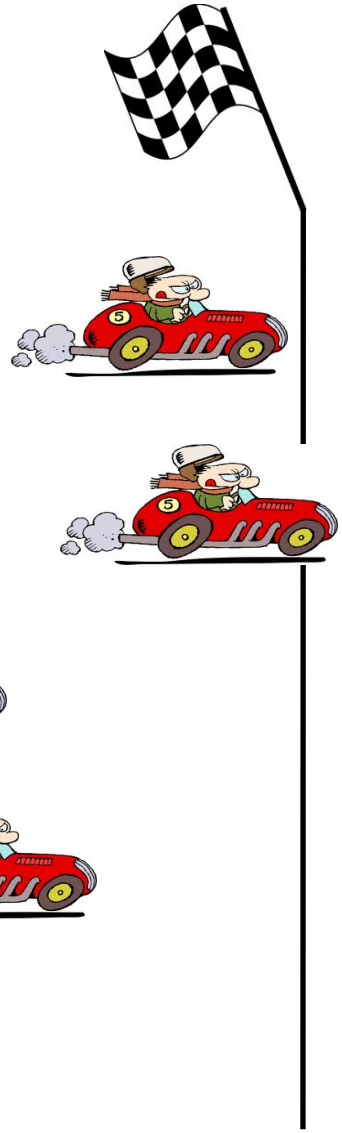


SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193

SUM: 20294984
TIME: 30210



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO

Samuelic array 5

Control flow divergences for dummies and brighties

```
void vecIncInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = size - i - 1;
  }
}
```

1

```
void vecConsInit(int* data, int size) {
  int cons = size / 2;
  for (int i = 0; i < size; ++i) {
    data[i] = cons;
  }
}
```

2

```
void vecAltInit(int* data, int size) {
  for (int i = 0; i < size; ++i) {
    if (i % 2) {
      data[i] = size;
    }
  }
}
```

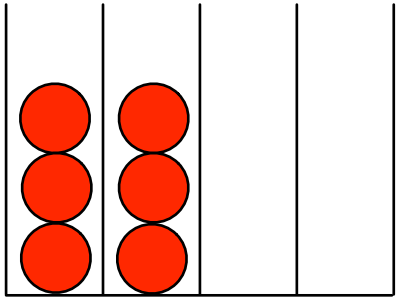
3

```
void vecRandomInit(int* data, int
size) {
  for (int i = 0; i < size; ++i) {
    data[i] = random() % size;
  }
}
```

4

```
void vecHalfInit(int* data, int size) {
  for (int i = 0; i < size/2; ++i) {
    data[i] = 0;
  }
  for (int i = size/2; i < size; ++i) {
    data[i] = size;
  }
}
```

5



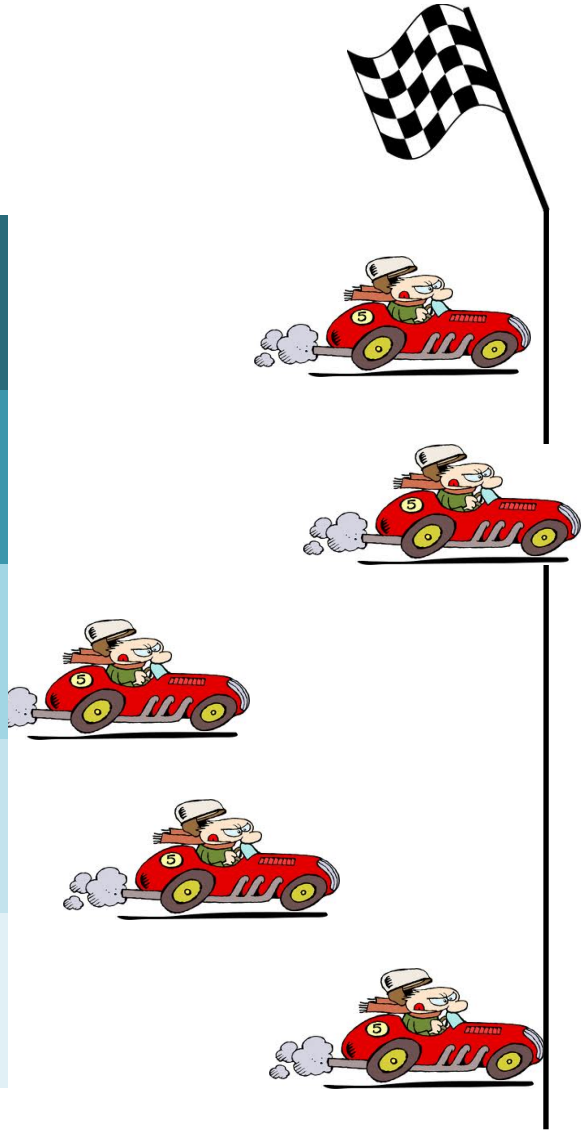
SUM: 20480000
TIME: 16250

SUM: 20480000
TIME: 16153

SUM: 20476800
TIME: 32193

SUM: 20294984
TIME: 30210

SUM: 20480000
TIME: 16157



Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Dynamic Detection of Divergences

DCC 888

Profiling for Divergences

GIVEN A PROGRAM, AND ITS INPUT, HOW MANY TIMES EACH BRANCH HAS BEEN VISITED, AND HOW MANY DIVERGENCES HAVE HAPPENED PER BRANCH?

1) Why is this question important?

2) How can we answer it?

A Simple Solution

- We can measure divergences with a *profiler* that works via instrumentation
- At each divergent path, we do a referendum among all the warp threads.
  - If they all vote together, then there is no divergence.
  - If they don't, then there is divergence.
- But we must find a writer…

Finding a thread to write the result is not so trivial, because a specific thread may not be active. How can we find a writer?

Finding a Writer

What is the asymptotic complexity of this algorithm?

```
int writer = 0;
bool gotWriter = false;
while (!gotWriter) {
  bool iAmWriter = false;
  if (lane_id == writer) {
    iAmWriter = true;
  }
  if (∃ t ∈ w | iAmWriter == true) {
    gotWriter = true;
  } else {
    writer++;
  }
}
```

High level description of block $L_{2(\text{find writer})}$

$L_{2(up)}$

```
1  %t1 = and %tid %k
2  %p2 = eq %t1 0
3  %writer = mov 0
```

$L_{2(\text{find writer})}$

```
4  %iAmWriter = eq %laneid, %writer
5  %gotWriter = vote.any %iAmWriter
6  %continue = neg %gotWriter
7  %writer = add %writer, 1
8  bra %continue $L2_find_writer
```

$L_{2(bottom)}$

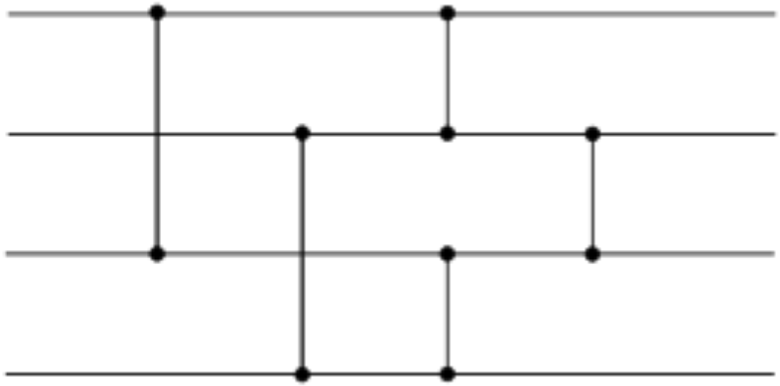
```
9  @iAmWriter %τ[L2] = atom.add %τ[L2] 1
10            %consensus = vote.uni %p2
11            %dvrg = neg %consensus
12            %wrDv = and %iAmWriter %dvrg
13 @wrDv      %δ[L2] = atom.add %δ[L2] 1
14            bra %p2 L3
```

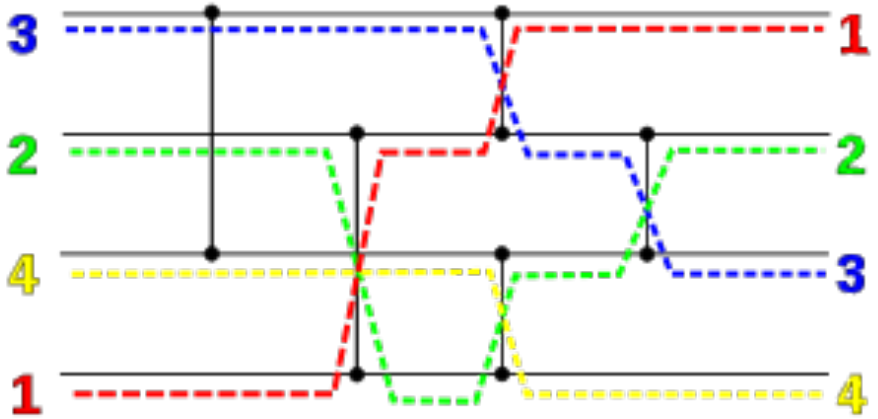
Is there any chance of not finding a writer at all?

Example: Bitonic Sort

- Bitonic Sort is the canonical example of a sorting network.
- A sorting network performs the same comparisons to sort any possible input.

Why is bitonic sort so good for parallel sorting?





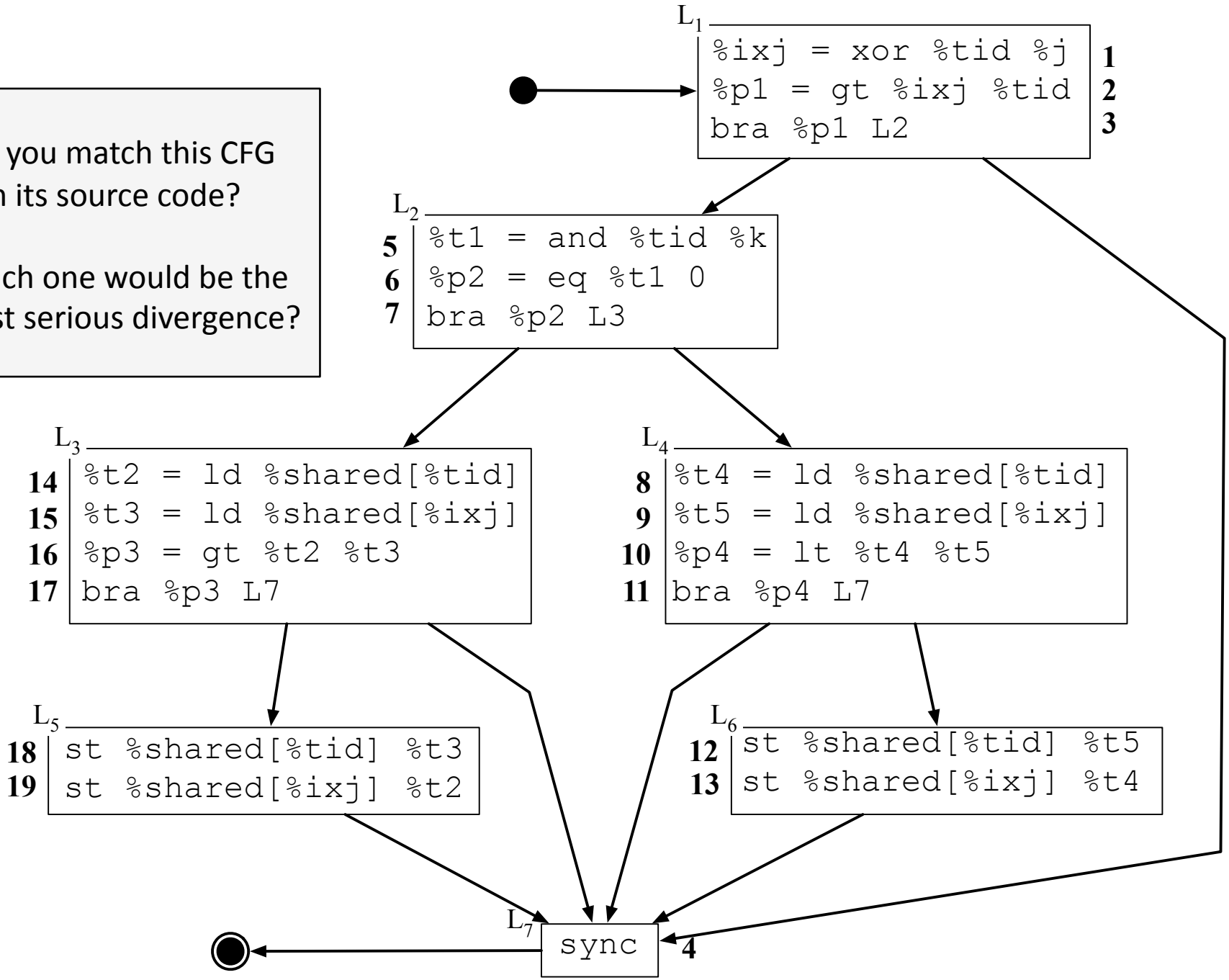
```
__global__ static void bitonicSort(int * values) {♧
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {
          if (shared[tid] > shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }
  }
  values[tid] = shared[tid];
}
```

In the rest of this example, we shall focus on this part of the Bitonic Sort kernel.

♧: GPU-Quicksort: A practical Quicksort algorithm for graphics processors

1) Can you match this CFG with its source code?

2) Which one would be the most serious divergence?



Warp Trace

How can we measure the cost of divergences?

cycle	label	opcode	def	use_1	use_2	t_0	t_1	t_2	t_3
1	L_1	xor	%ixj	%tid	%j	ι_1	ι_1	ι_1	ι_1
2		gt	%p1	%ixj	%tid	ι_2	ι_2	ι_2	ι_2
3		bra		%p1	L_2	ι_3	ι_3	ι_3	ι_3
4	L_2	and	%t1	%tid	%k	ι_5	•	ι_5	•
5		eq	%p2	%t1	0	ι_6	•	ι_6	•
6		bra		%p2	L_3	ι_7	•	ι_7	•
7	L_3	load	%t2	%shared	%tid	ι_{14}	•	•	•
8		load	%t3	%shared	%ixj	ι_{15}	•	•	•
9		gt	%p3	%t2	%t3	ι_{16}	•	•	•
10		bra		%p3	L_7	ι_{17}	•	•	•
11	L_4	load	%t4	%shared	%tid	•	•	ι_8	•
12		load	%t5	%shared	%ixj	•	•	ι_9	•
13		lt	%p4	%t4	%t5	•	•	ι_{10}	•
14		bra		%p3	L_7	•	•	ι_{11}	•
15	L_5	store		%tid	%t3	ι_{18}	•	•	•
16		store		%tid	%t2	ι_{19}	•	•	•
17	L_6	store		%tid	%t5	•	•	ι_{12}	•
18		store		%tid	%t4	•	•	ι_{13}	•
19	L_7	sync				ι_4	ι_4	ι_4	ι_4

These values give the number of divergences and the number of visits in each branch (per warp).

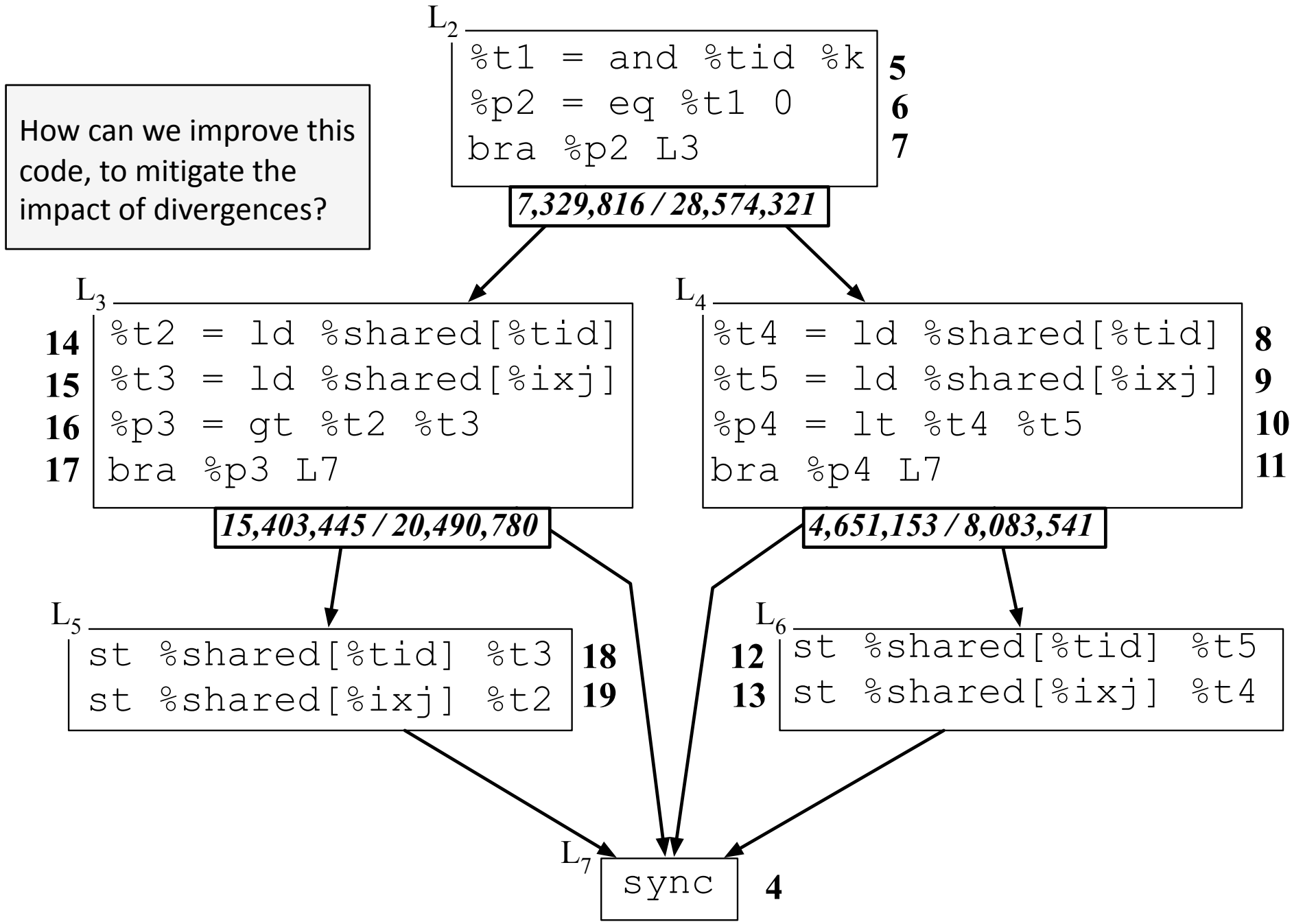
```
__global__ static void bitonicSort(int * values) {
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {
          if (shared[tid] > shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }

  values[tid] = shared[tid];
}
```

7,329,816 / 28,574,321

15,403,445 / 20,490,780

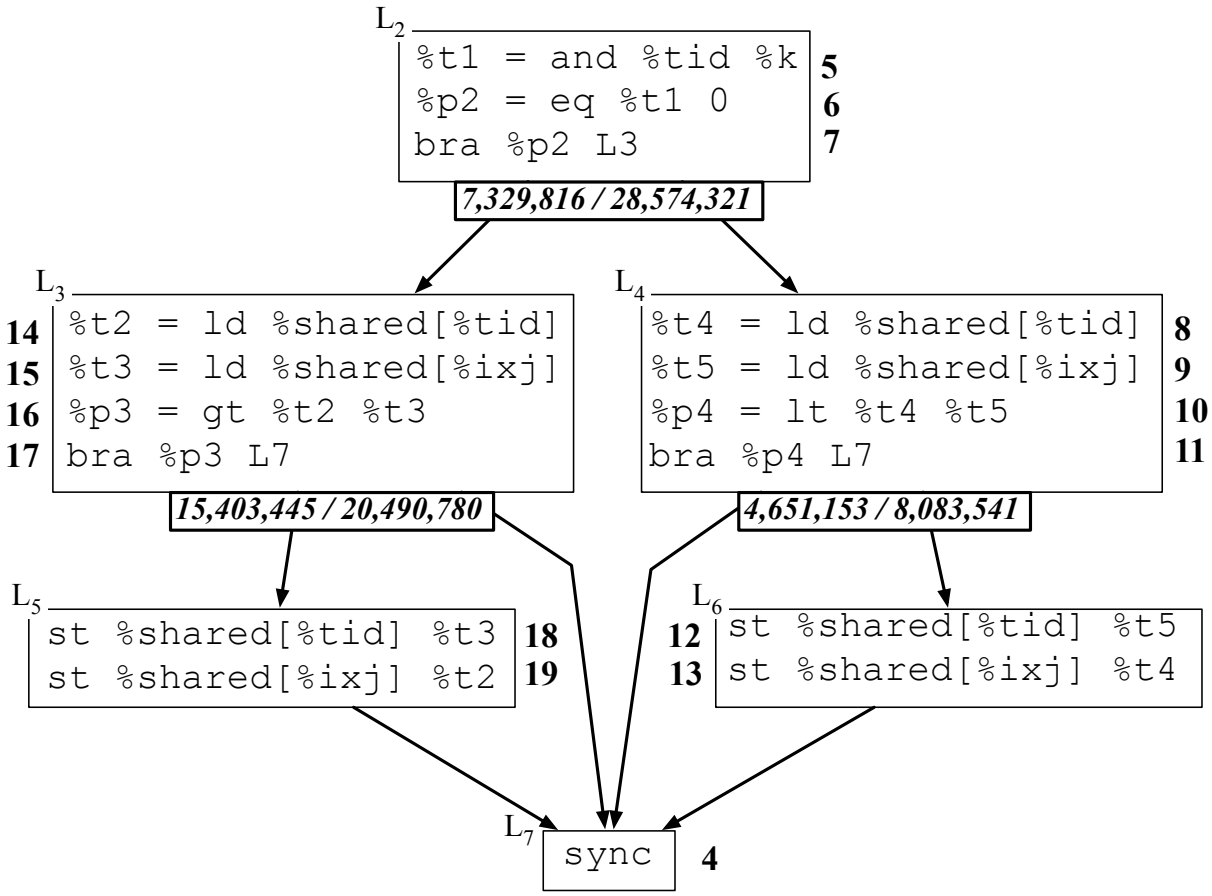
4,651,153 / 8,083,541


How can we improve this code, to mitigate the impact of divergences?
L2
%t1 = and %tid %k
%p2 = eq %t1 0
bra %p2 L3
5
6
7
7,329,816 / 28,574,321
L3
14 %t2 = ld %shared[%tid]
15 %t3 = ld %shared[%ixj]
16 %p3 = gt %t2 %t3
17 bra %p3 L7
15,403,445 / 20,490,780
L4
%t4 = ld %shared[%tid] 8
%t5 = ld %shared[%ixj] 9
%p4 = lt %t4 %t5 10
bra %p4 L7 11
4,651,153 / 8,083,541
L5
st %shared[%tid] %t3 18
st %shared[%ixj] %t2 19
L6
12 st %shared[%tid] %t5
13 st %shared[%ixj] %t4
L7
sync
4

```
if ((tid & k) == 0) {
  if (shared[tid] > shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
} else {
  if (shared[tid] < shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
}
```

How can the programmer change the code on the left to mitigate the impact of divergences?

Is there any classic compiler optimization that we could use to mitigate the impact of divergences?



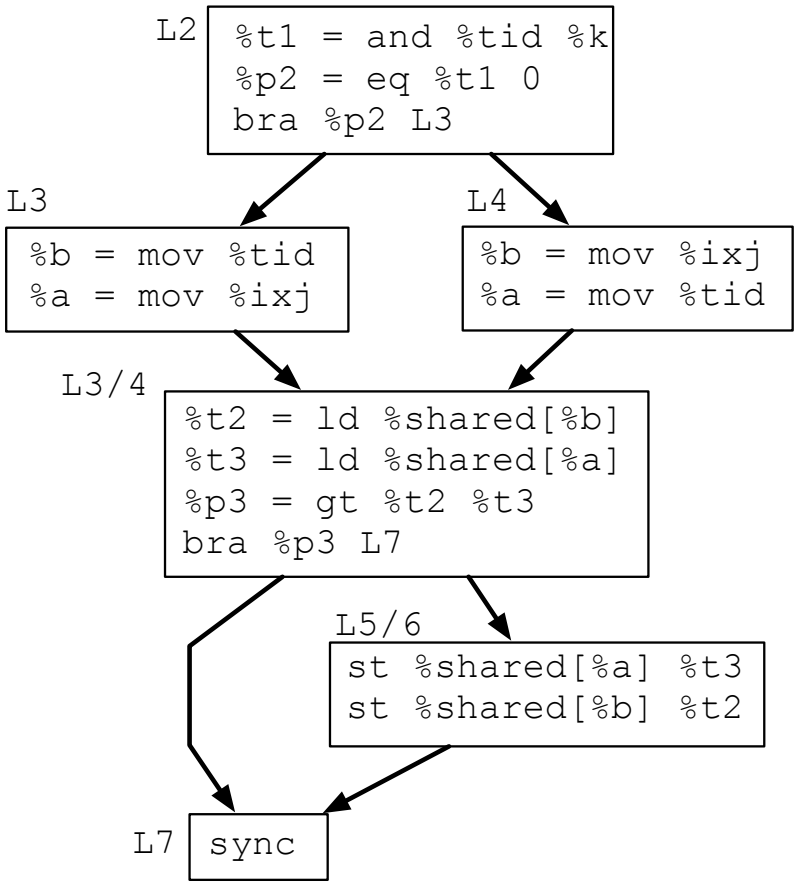
First Optimization: 6.7% speed up♤

(a)

```
unsigned int a, b;
if ((tid & k) == 0) {
  b = tid;
  a = ixj;
} else {
  b = ixj;
  a = tid;
}
if (sh[b] > sh[a]) {
  swap(sh[b], sh[a]);
}
```

Yet, it is still possible to improve this code, if we use C's ternary selectors. What could we do?

(b)



♤: Numbers refer to GTX 260

Second Optimization: 9.2% speed up

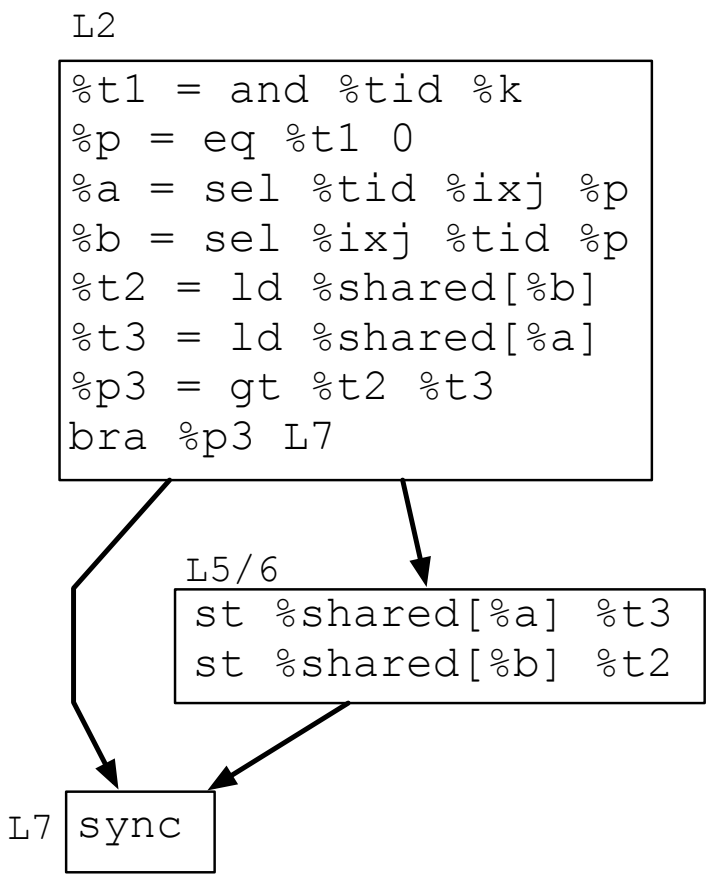
(a)

```
int p = (tid & k) == 0;
unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;

if (sh[b] > sh[a]) {
  swap(sh[b], sh[a]);
}
```

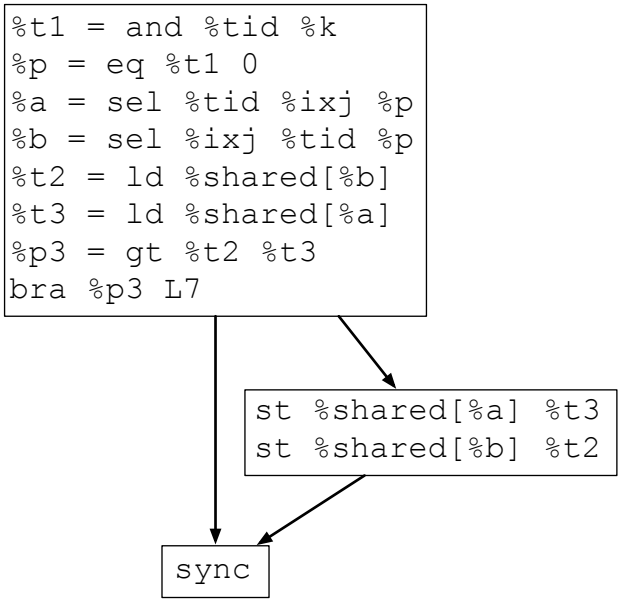
This code still contains conditional assignments, but this is not the same as divergent execution. Why?

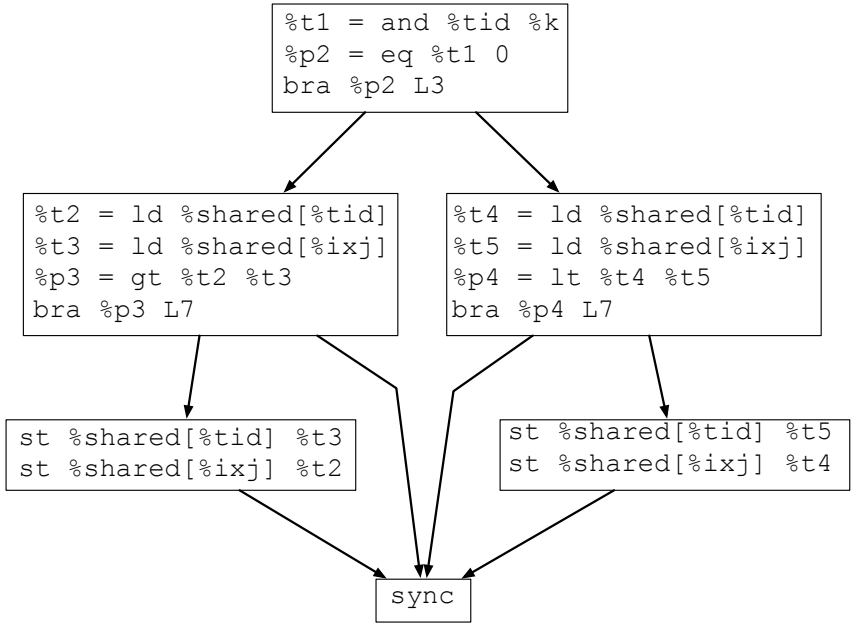
(b)



The final result

```
if ((tid & k) == 0) {
  if (shared[tid] > shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
} else {
  if (shared[tid] < shared[ixj]) {
    swap(shared[tid], shared[ixj]);
  }
}
```






```
int p = (tid & k) == 0;
unsigned b = p ? tid : ixj;
unsigned a = p ? ixj : tid;

if (shared[b] > shared [a]) {
  swap(shared[b], shared[a]);
}
```

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Static Detection of Divergences

DCC 888

Divergence Analysis

GIVEN A PROGRAM, WHICH BRANCHES MAY CAUSE DIVERGENCES, AND WHICH BRANCHES WILL NEVER DO IT?

1) Why is this question important?

2) Can we answer this question with our profiler?

Divergent and Uniform Variables

- A program variable is *divergent* if different threads see it with different values.
- If different threads always see that variable with the same value, then we say that it is *uniform*.

- These are the divergent variables:
  1. *v* = **`tid`**
  2. **`atomic`** { v = f(...) }
  3. *v* is **data dependent** on divergent variable *u*.
  4. *v* is **control dependent** on divergent variable *u*.

Can you explain **this** kind of divergence? What about **this** kind?

The thread ID is always divergent

```
__global__
void saxpy (int n, float alpha, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = alpha * x[i] + y[i];
}
```

Each thread sees a different thread id, so…

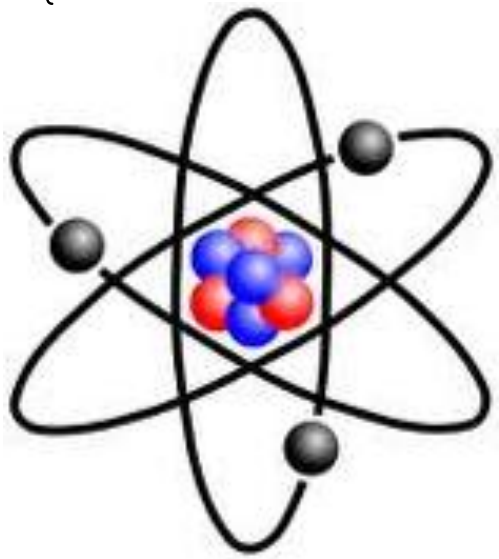
Threads in different blocks may have the same threadIdx.x. Is this a problem?

Variables Defined by Atomic Operations

- The macro ATOMINC increments a global memory position, and returns the value of the result.

```
__global__
void ex_atomic (int index, float* v) {
  int i = 0;
  i = ATOMINC( v[index] );
}
```

Why is variable i in the program above divergent?

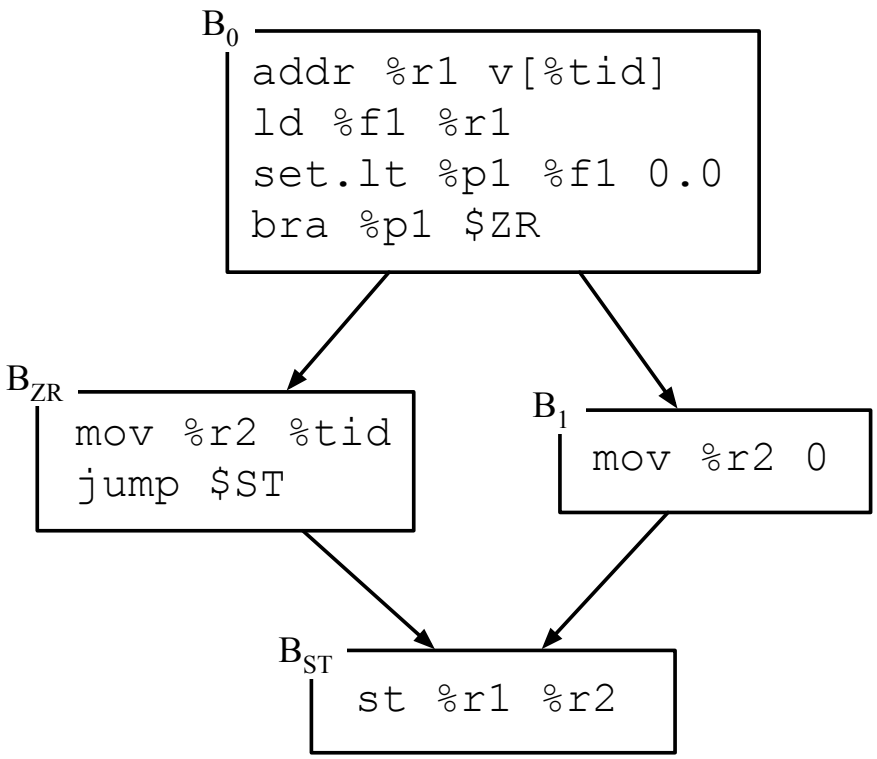


Dependences

- We recognize two types of dependences: *data* and *control*.
- If the program contains an assignment such as $v = f(v_1, v_2, \ldots, v_n)$, then v is **data dependent** on every variable in the right side, i.e, v_1, v_2, …, and v_n.
- If the value assigned to variable v depends on a branch controlled by a predicate p, then we say that v is **control dependent** on p.
- Divergences propagate transitively on the graph determined by the dependence relation.

Finding Dependencies

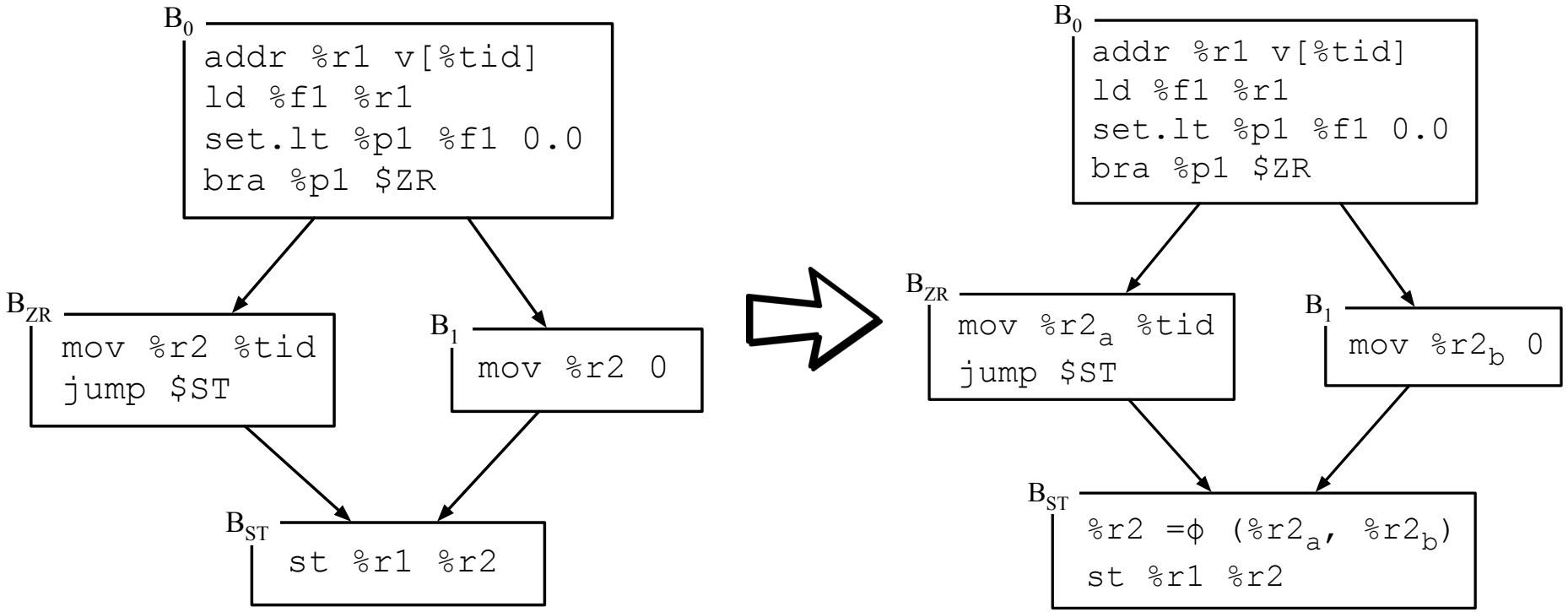
- We would like to bind a single abstract state to each variable: either that variable is *divergent*, or it is *uniform*.
- This is not so straight forward, because a variable may be divergent at some program points, and uniform at others.



Yet, this is a problem that we can easily solve, by choosing an appropriate program representation. Which one?

Static Single Assignment Form

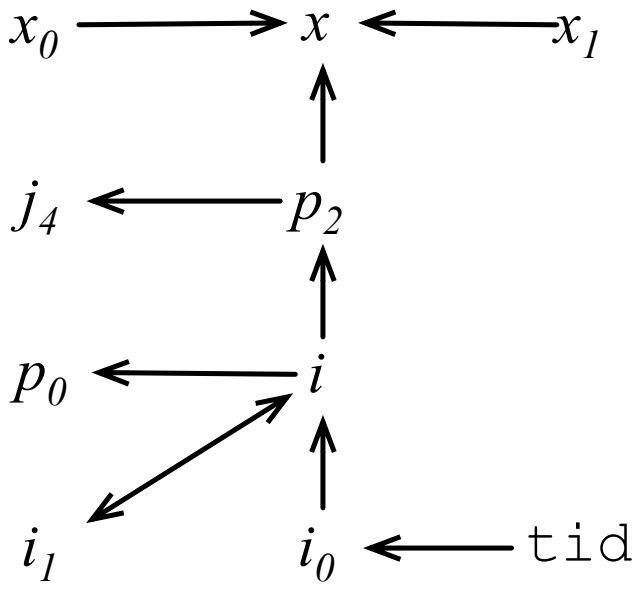
- We can use the Static Single Assignment (SSA) form to ensure that each variable name has only one static definition site.
- SSA is already used in almost every compiler anyway…

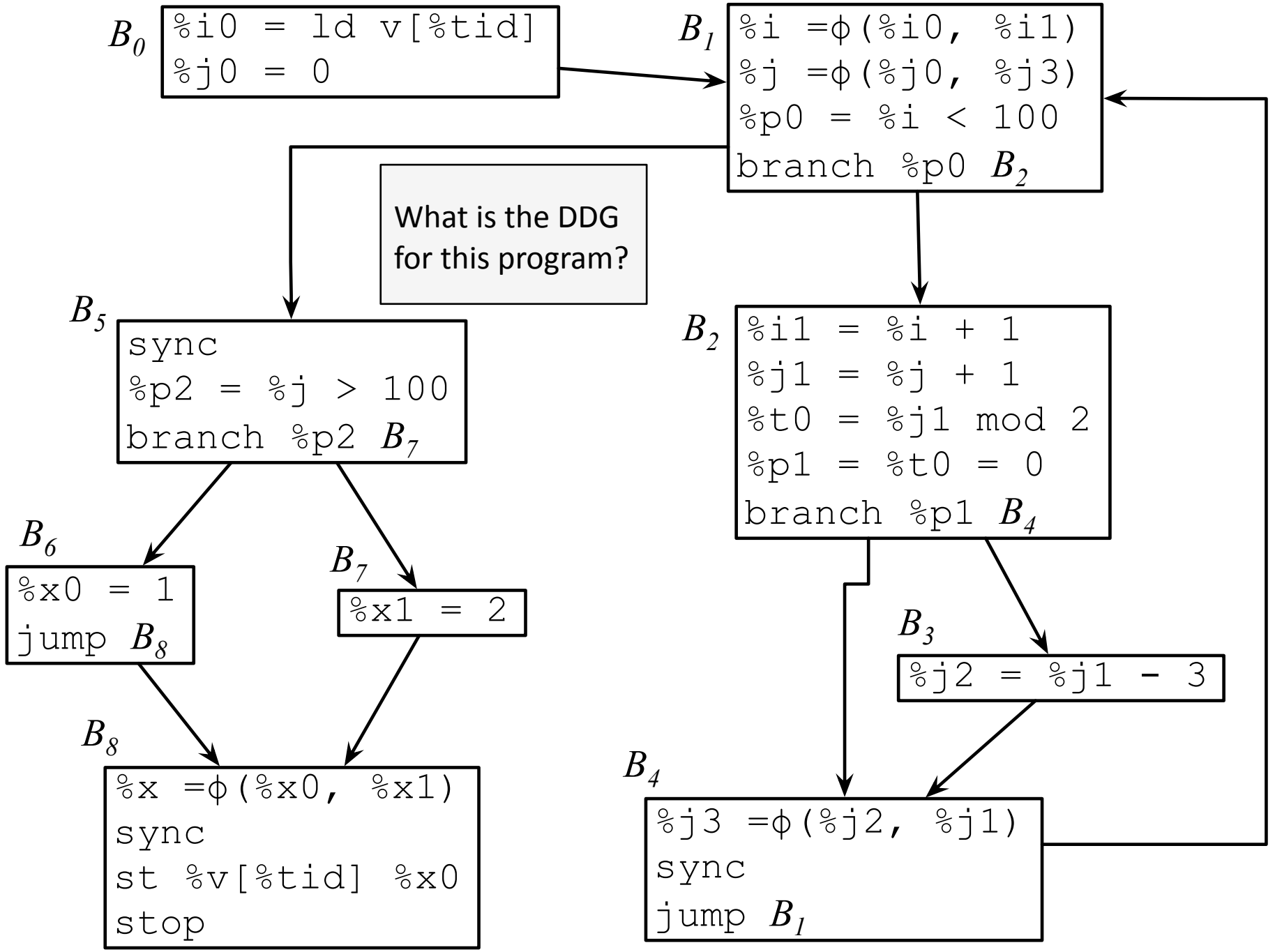


The Data Dependence Graph

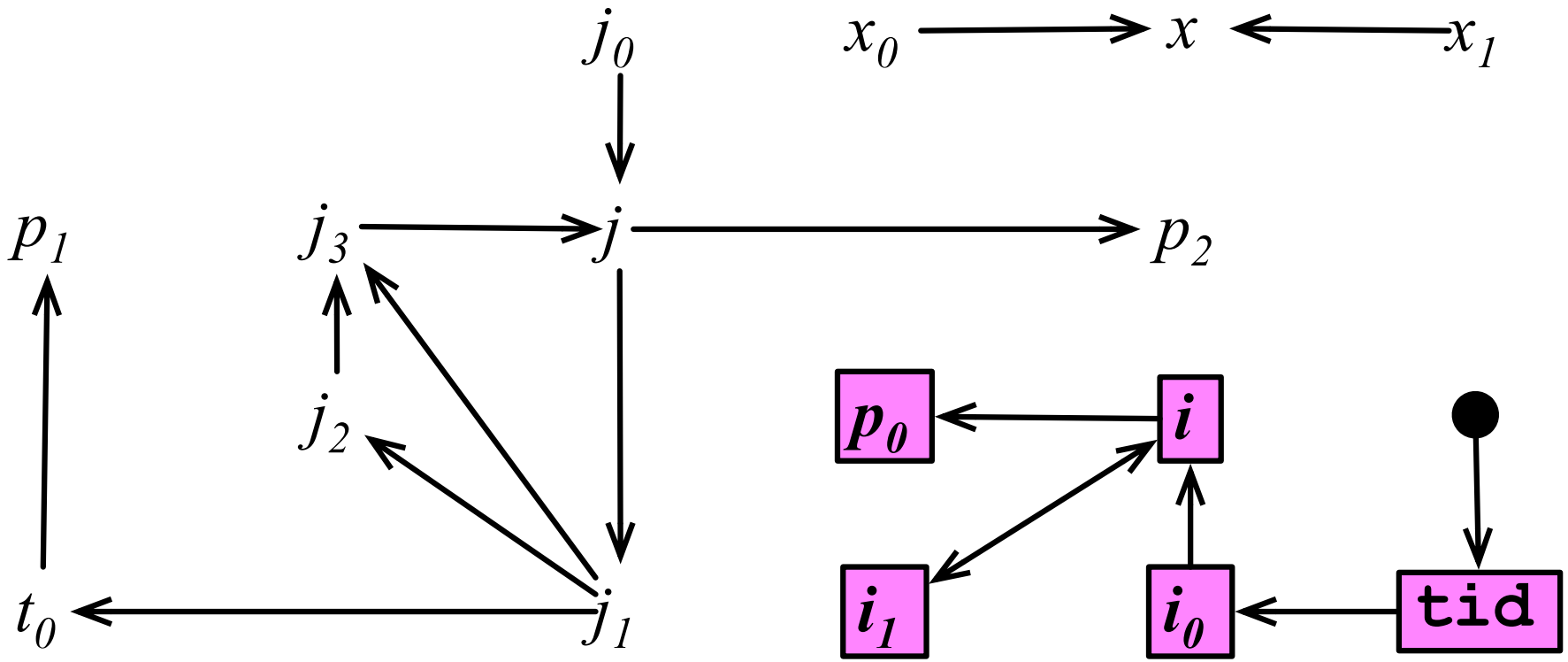
- We can see divergences as a reachability problem on the *Data Dependence Graph* (DDG).
- The DDG has a node for each variable.
- If variable *v* is data dependent on variable *u*, then we have a node from *u* to *v*.

Once we have built this graph, from which nodes do we start the reachability search?



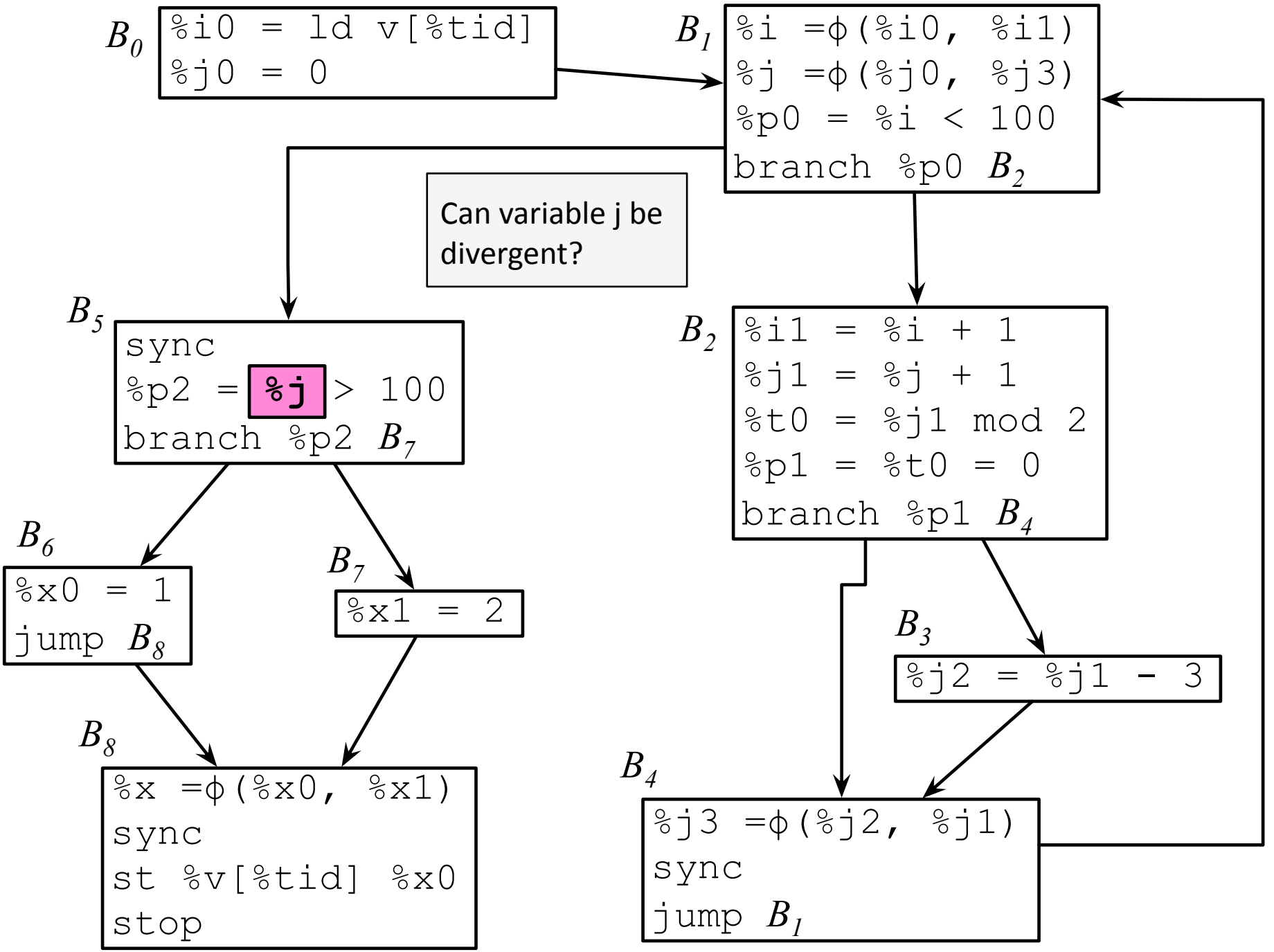

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the DDG for this program?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

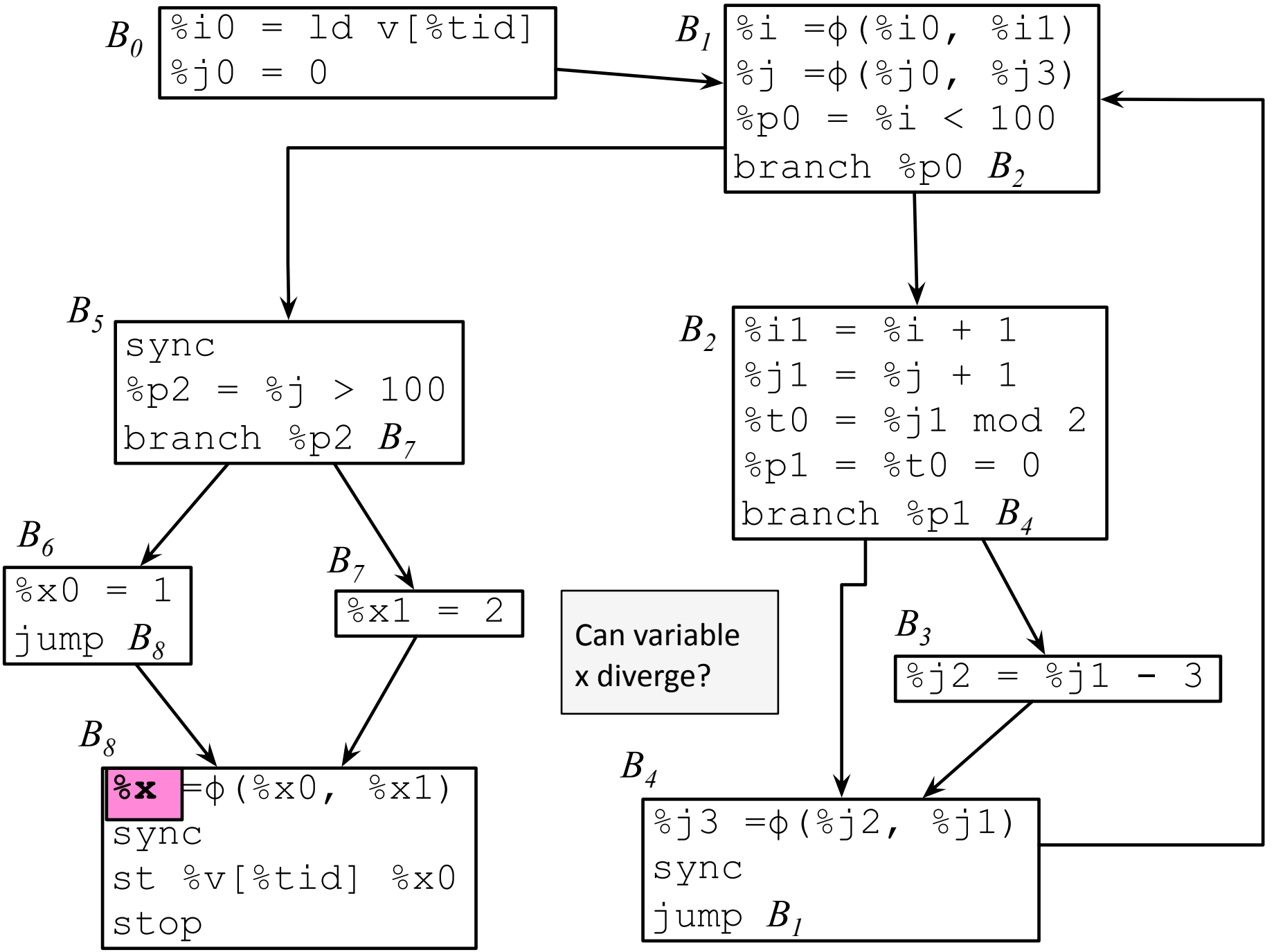
Data Dependences are not Enough

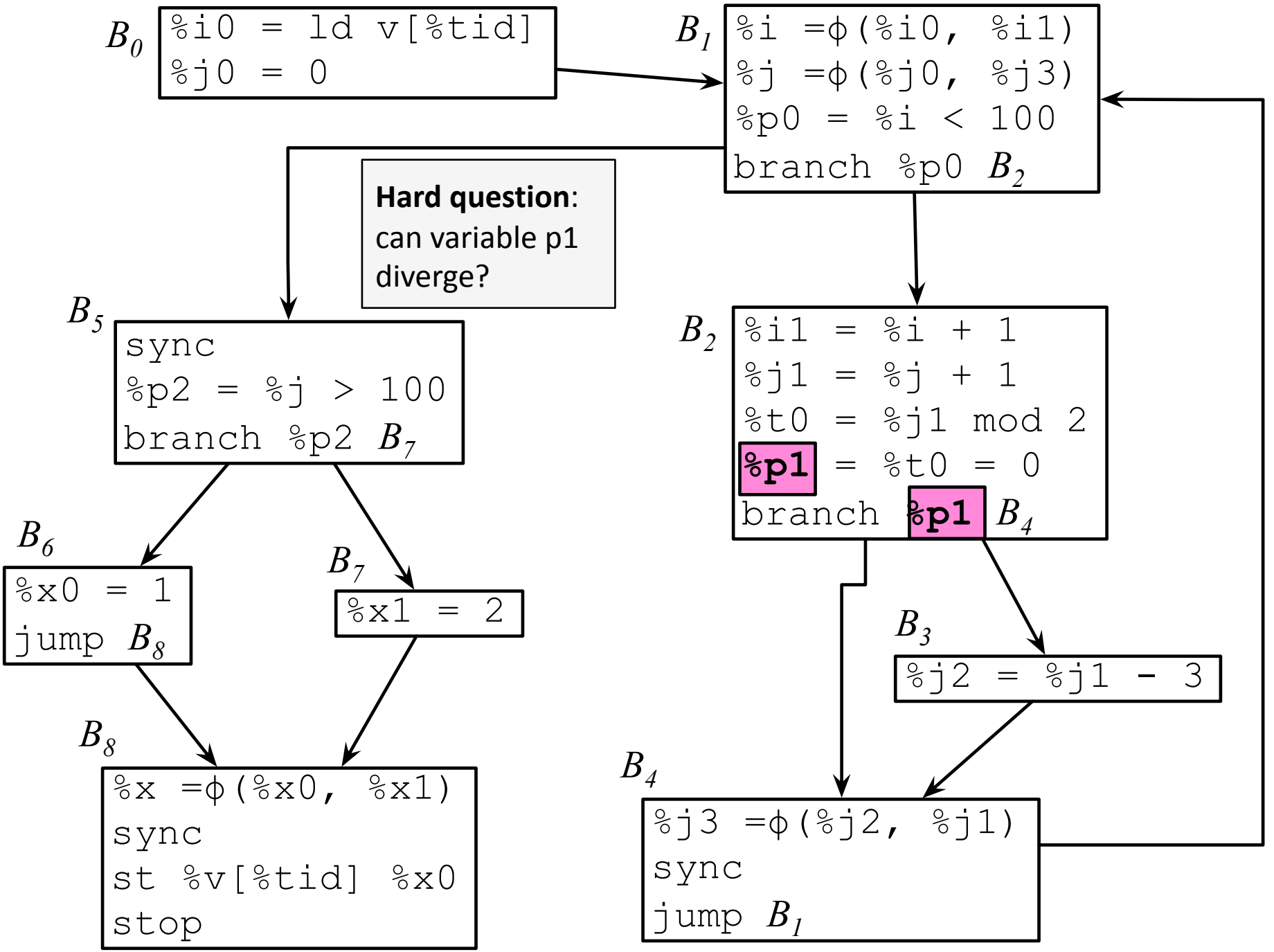


This is the data dependence graph for our example. It correctly marks some variables as divergent, yet, it misses some divergences.

Any intuition on why these data dependences are not enough to catch all the divergences?


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Can variable j be divergent?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
Can variable
x diverge?
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

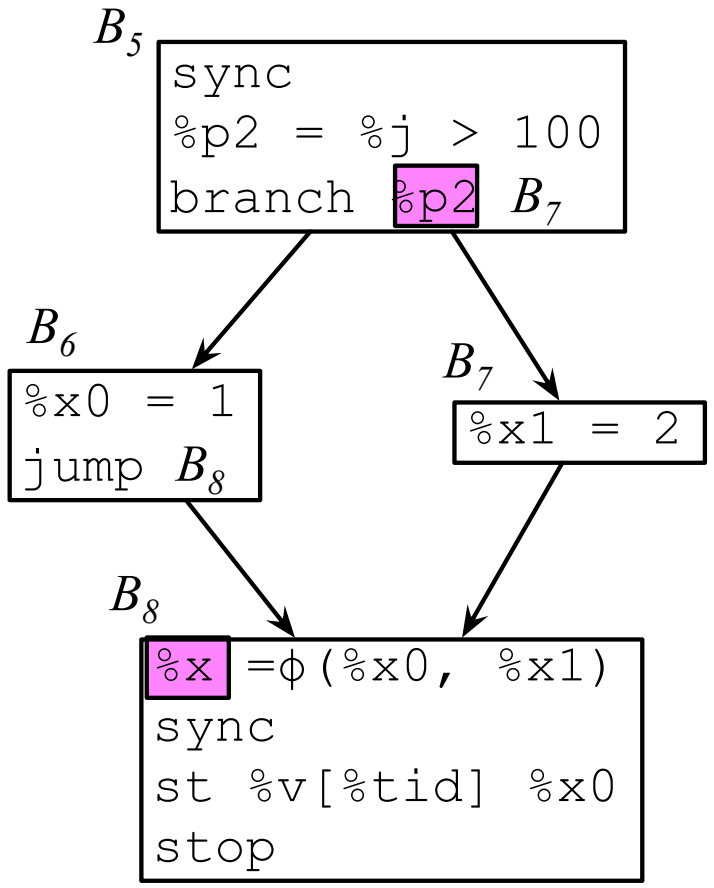

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
Hard question:
can variable p1
diverge?
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_3
%j2 = %j1 - 3
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop

Control Dependences

- A variable *v* is *control dependent* on a predicate *p*, if the value assigned to *v* depends on a branch that *p* controls.

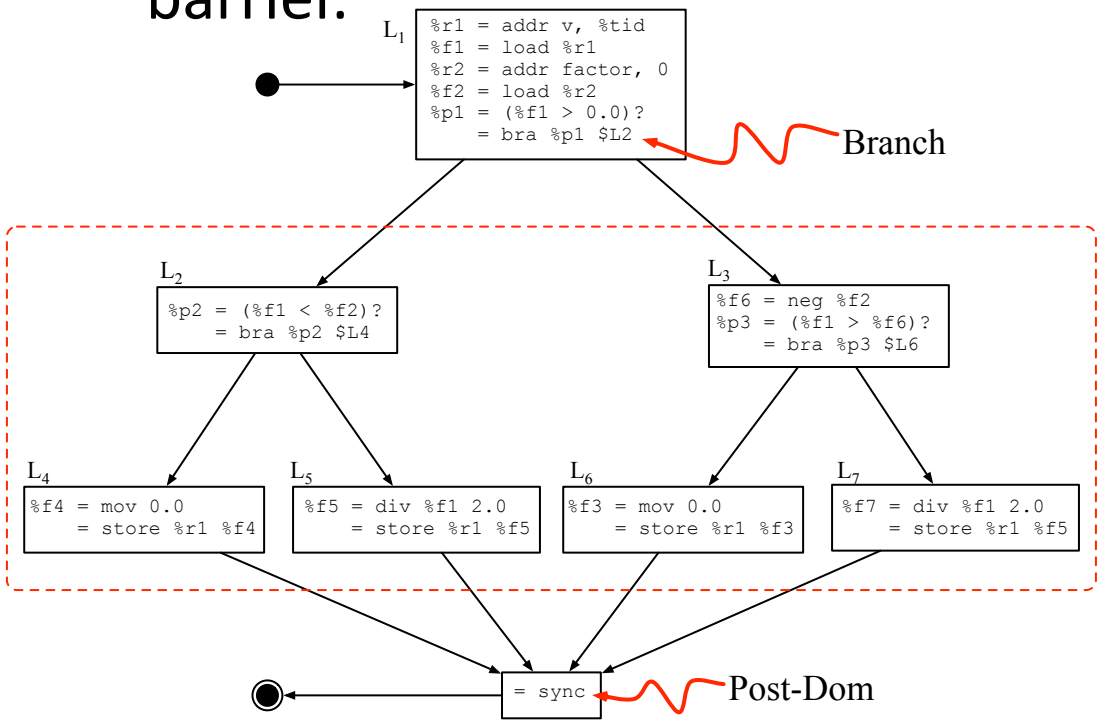
1) In this program on the right, why does the value assigned to %x depends on %p2?

2) How can we find if a variable is control dependent on another?



The influence region

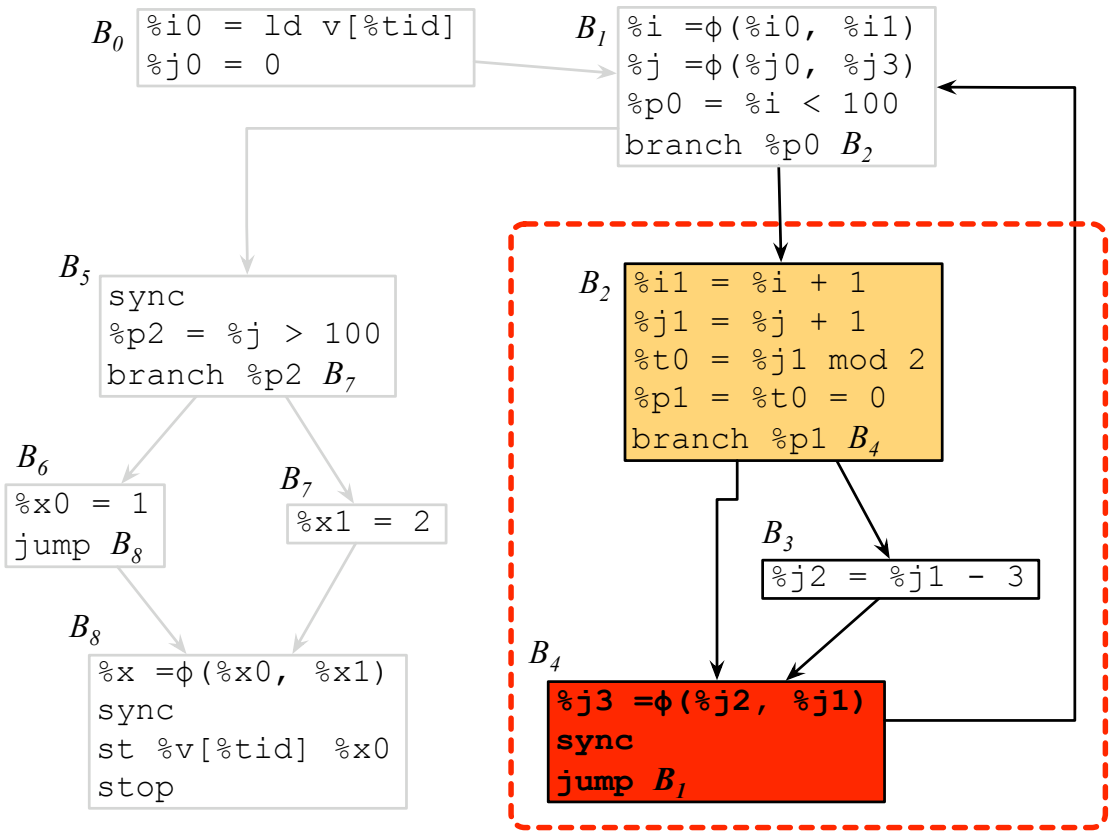
- To find control dependences, we will need the notion of *influence region*.
- The influence region of a predicate is the set of basic blocks that may (or may not) be reached depending on the value of the predicate, up to the synchronization barrier.

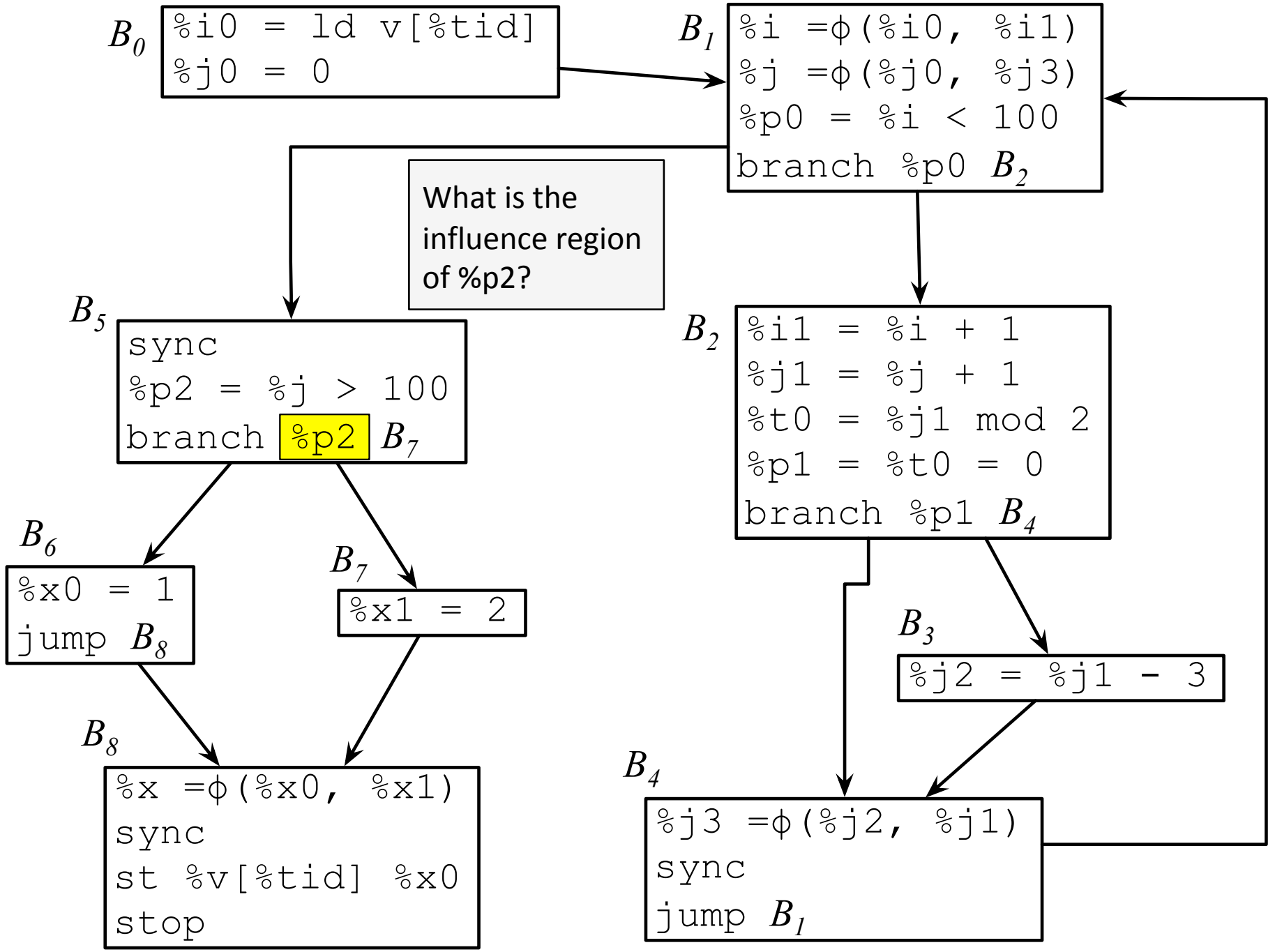


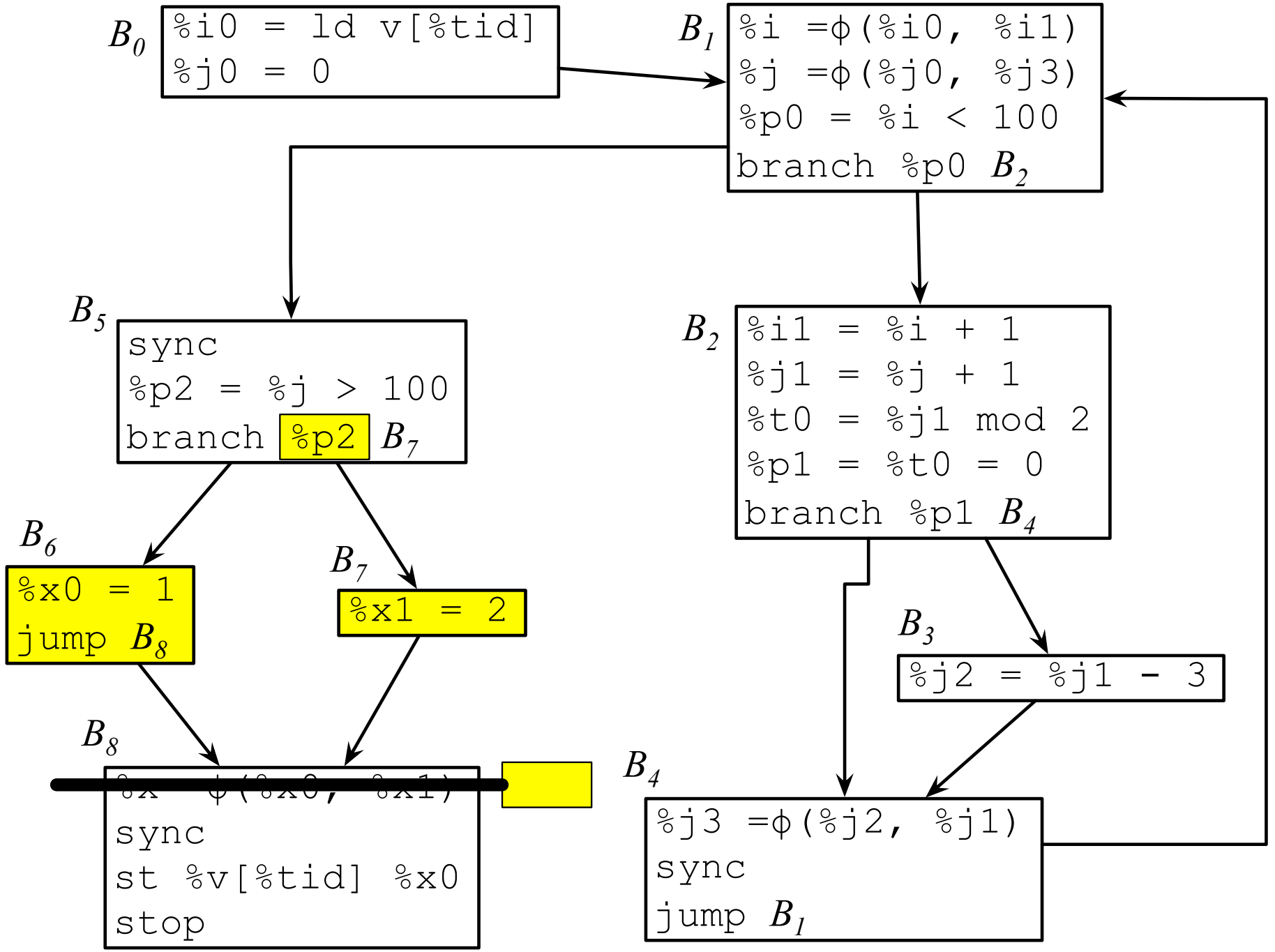
- The influence region goes from the branch to its **post-dominator**.
- The Synchronization barrier is placed at the *immediate* post-dominator of the branch.

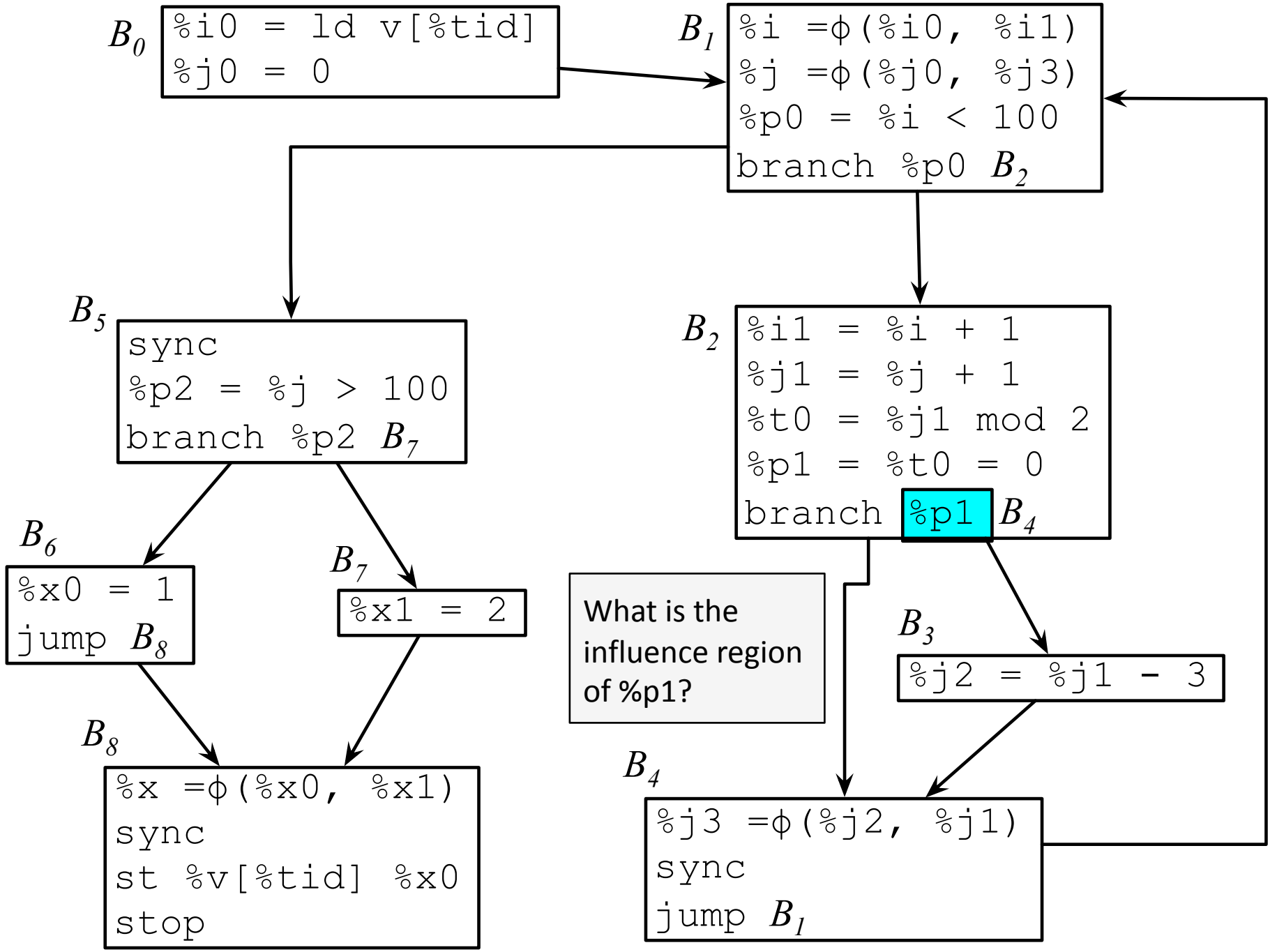
Post-Dominance

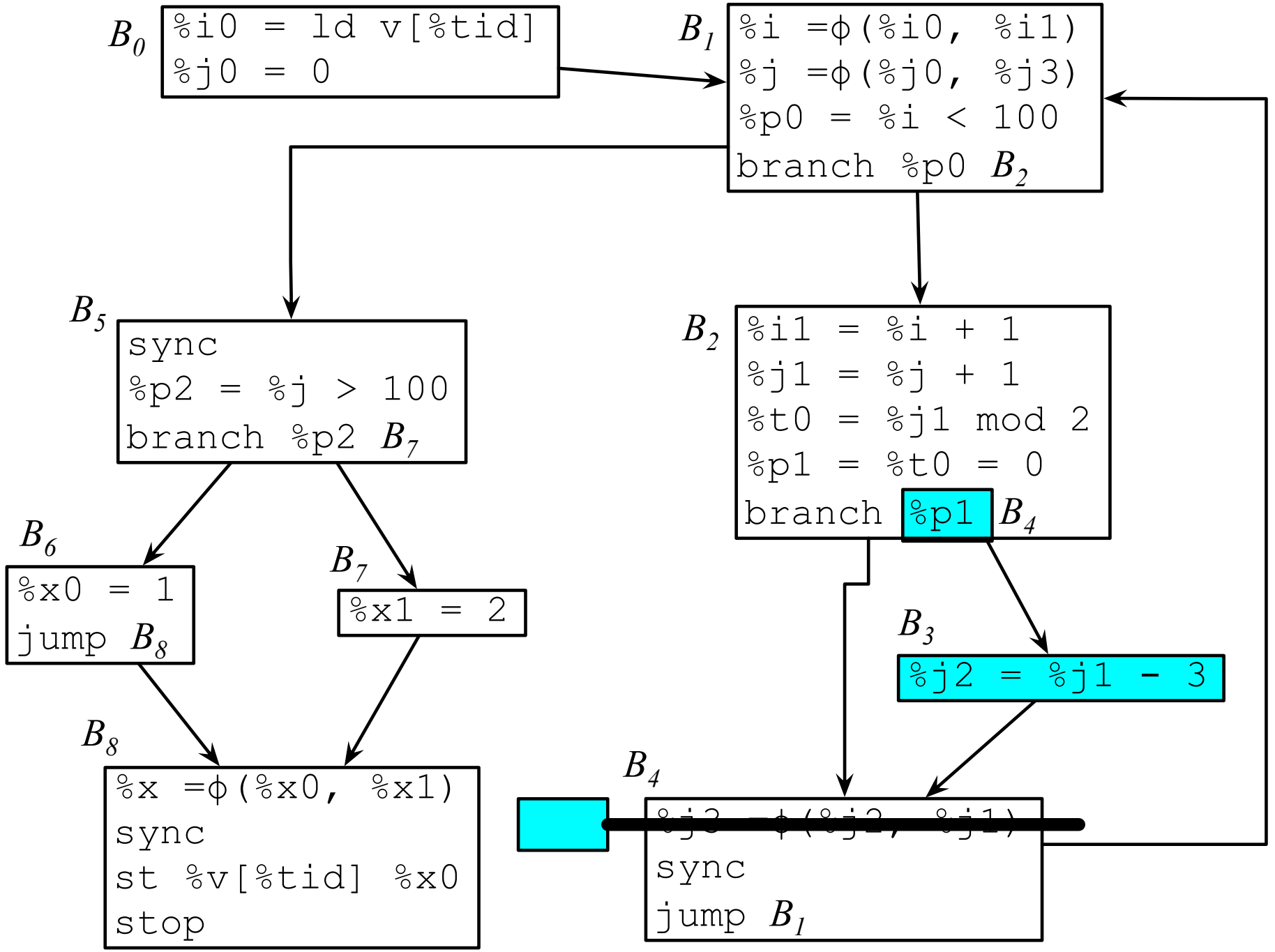
- To find control dependencies, we need **this** notion.
- A Basic block B_1 post-dominates another basic block B_2 if, and only if, every path from B_2 to the end of the program goes across B_1.
- For instance, in our running example, B_4 post-dominates B_2; thus, there is a sync barrier at B_4 to synchronize the threads that may diverge at B_2.

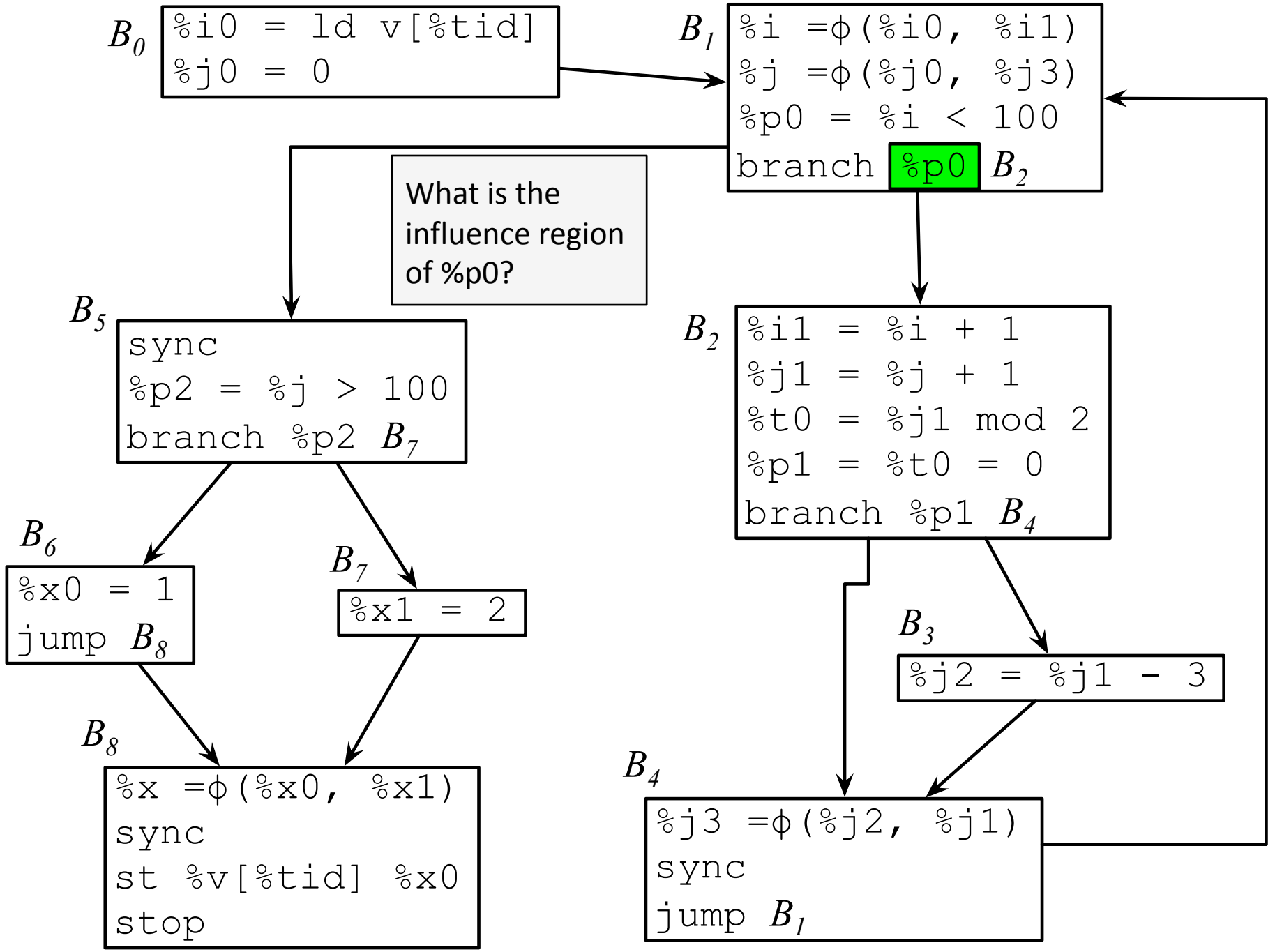


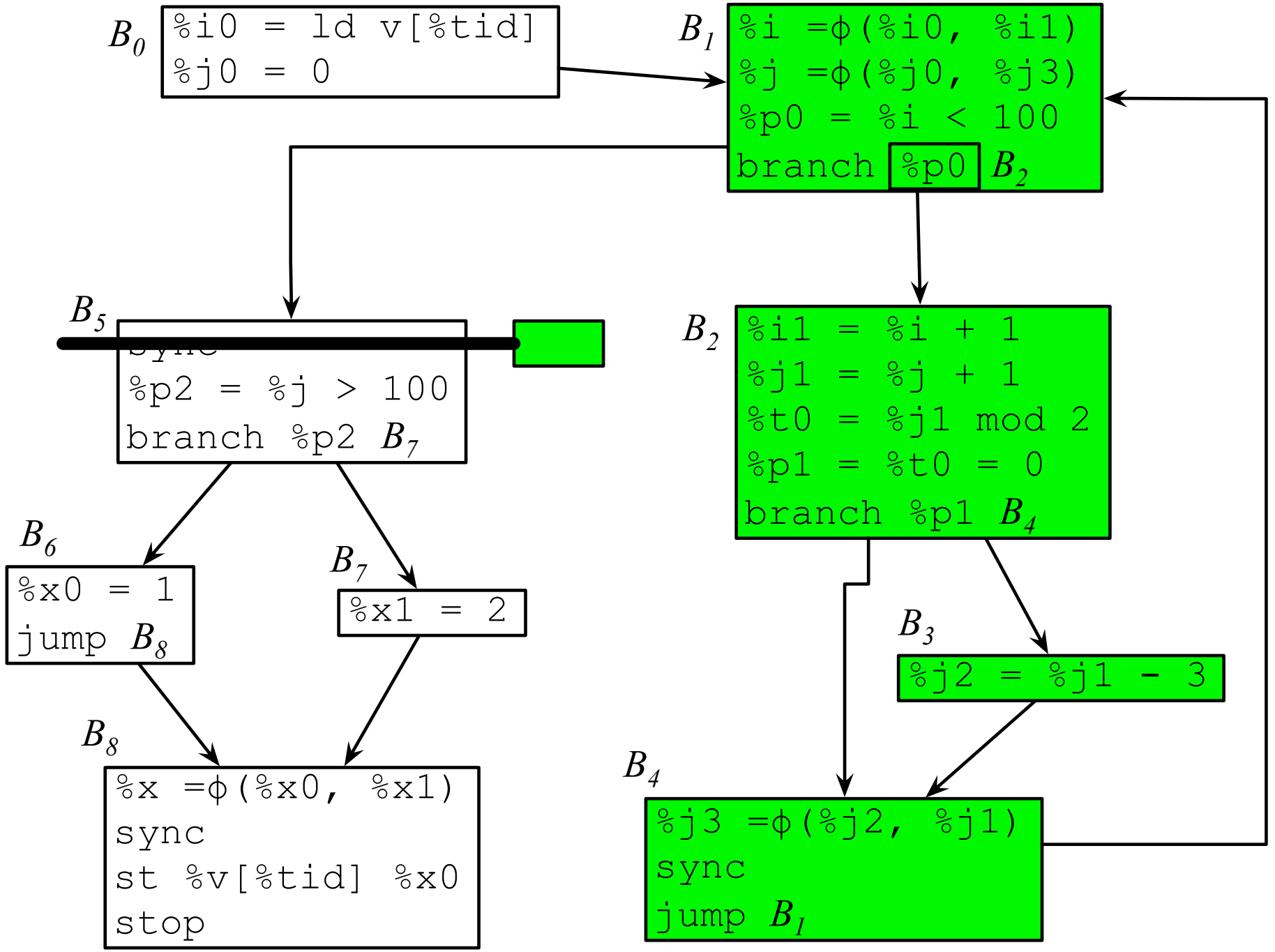

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the influence region of %p2?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x = ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_3
%j2 = %j1 - 3
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
What is the influence region of %p1?


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1

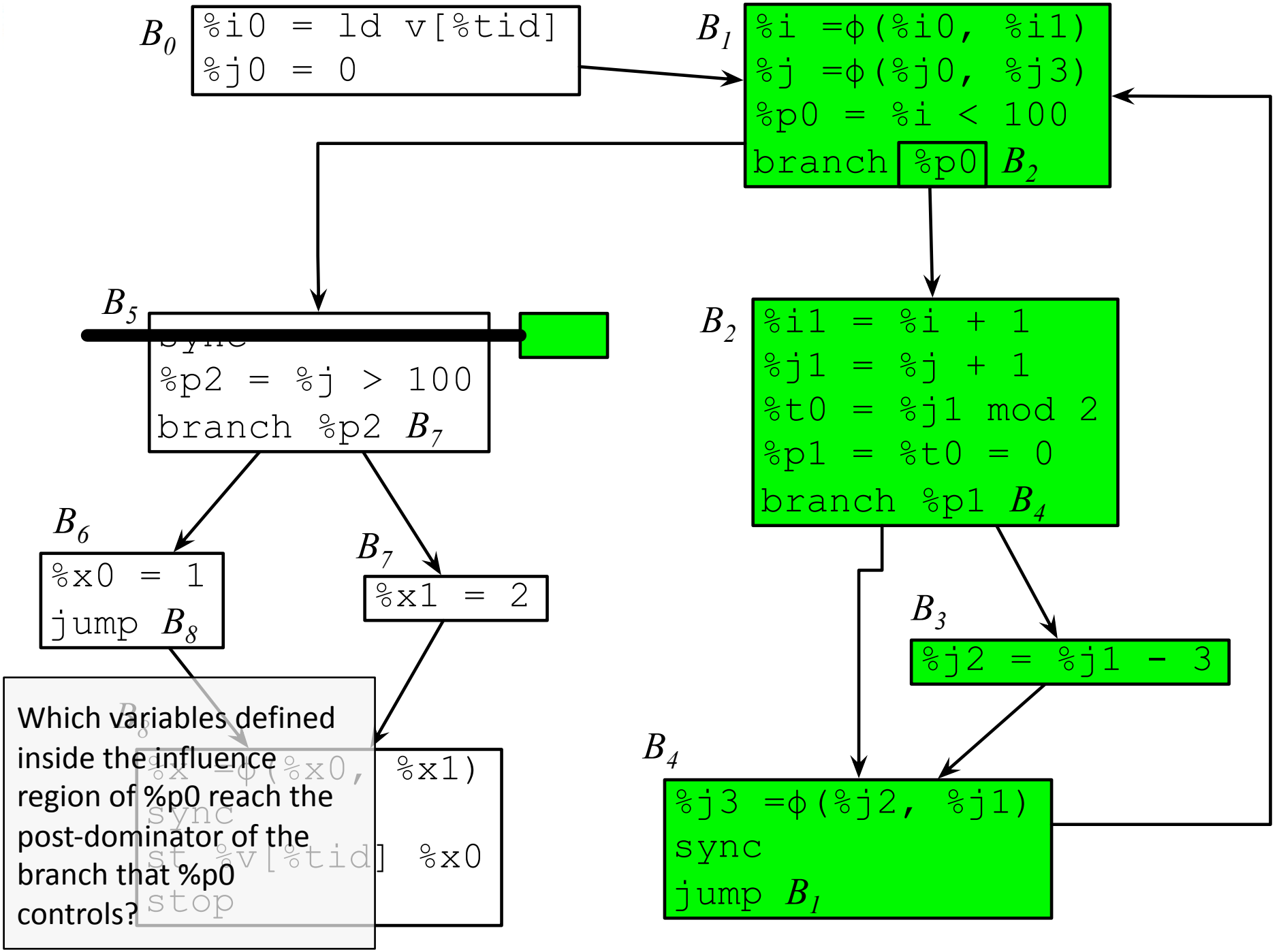

B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
What is the influence region of %p0?
B_5
sync
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_3
%j2 = %j1 - 3
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1
B_5
%p2 = %j > 100
branch %p2 B_7
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_8
%x =ϕ(%x0, %x1)
sync
st %v[%tid] %x0
stop

The law of control dependences

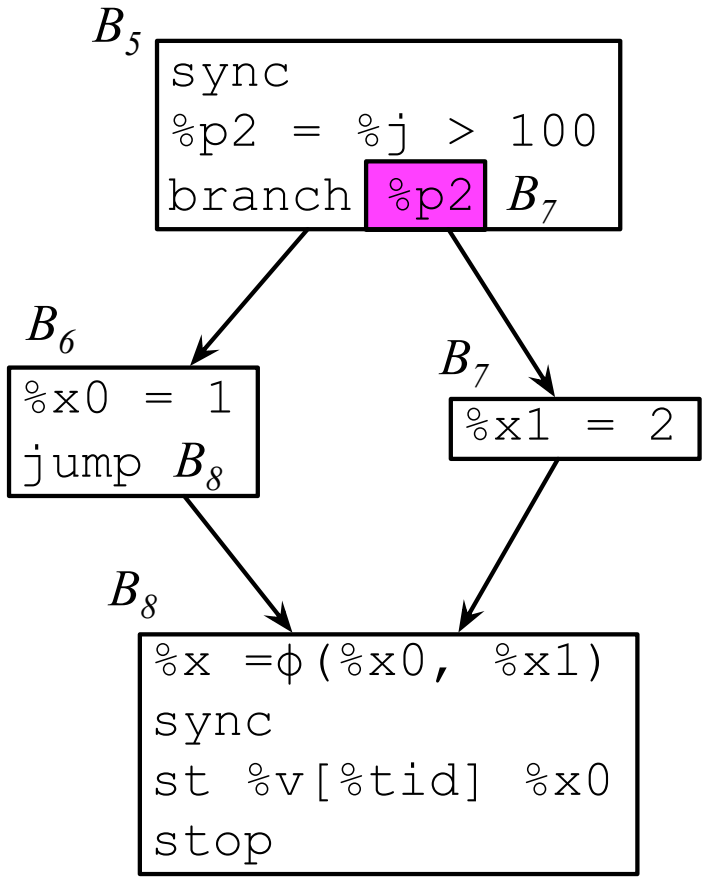
- **Theorem**: Let branch %p B be a branch instruction, and let l_p be its post dominator. A variable v is control dependent on %p if, and only if, v is defined inside the influence region of the branch and v reaches l_p.
  - Just to remember: l_p has an implicit synchronization barrier for the threads that have diverged at the branch.

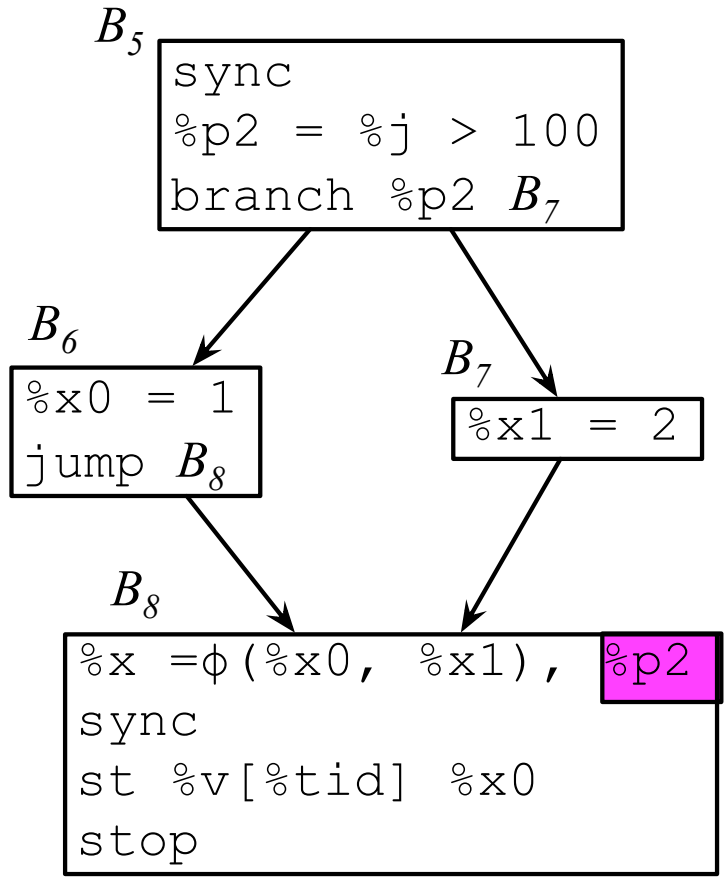
What does it mean for a variable to *reach* a program point?


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
%p2 = %j > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_4
%j3 =ϕ(%j2, %j1)
sync
jump B_1
Which variables defined inside the influence region of %p0 reach the post-dominator of the branch that %p0 controls?

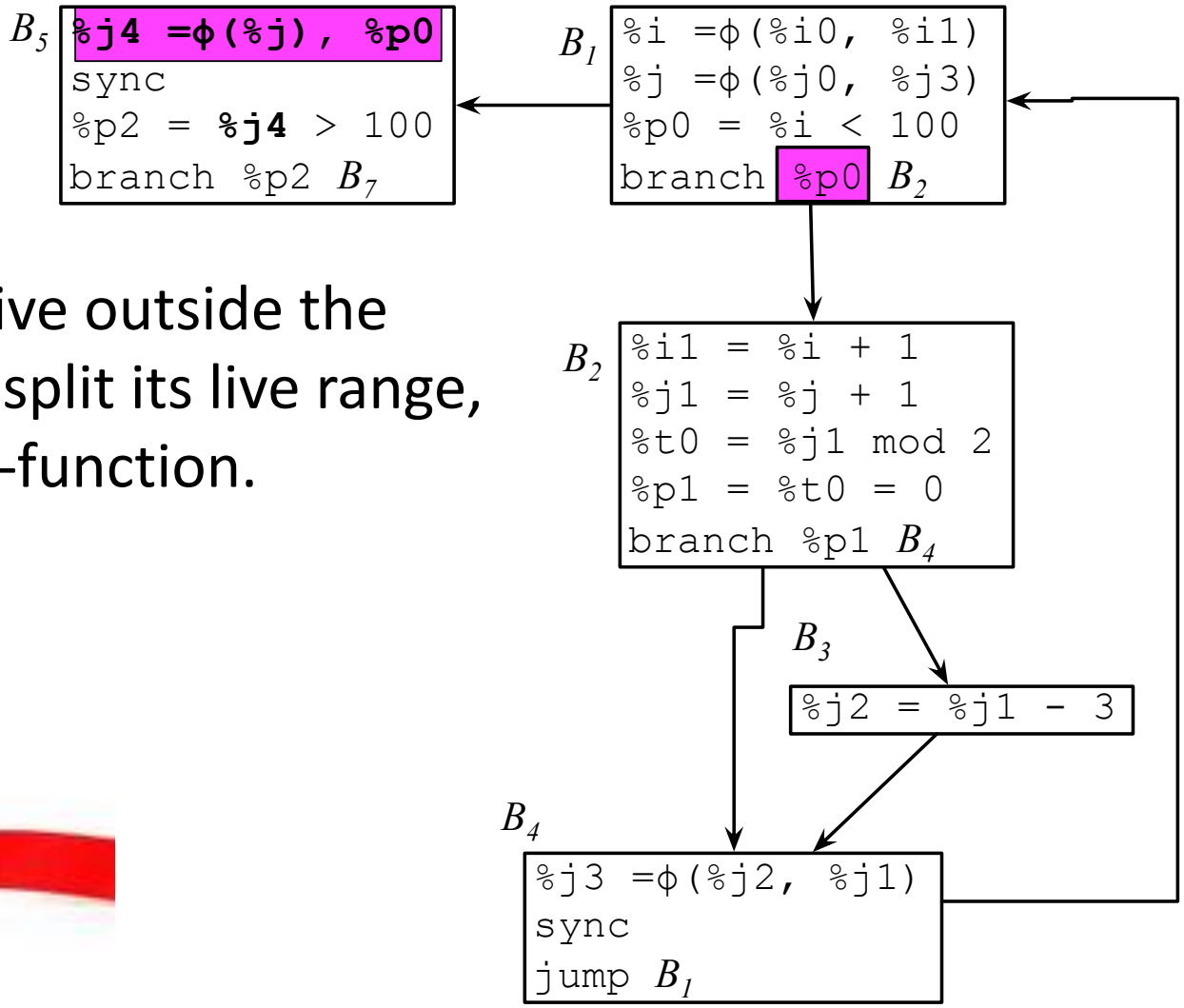
From Control to Data Dependences

- To transform control dependences on data dependences, we augment phi-functions with predicates. For instance, to create a data dependence between %x and %p2:

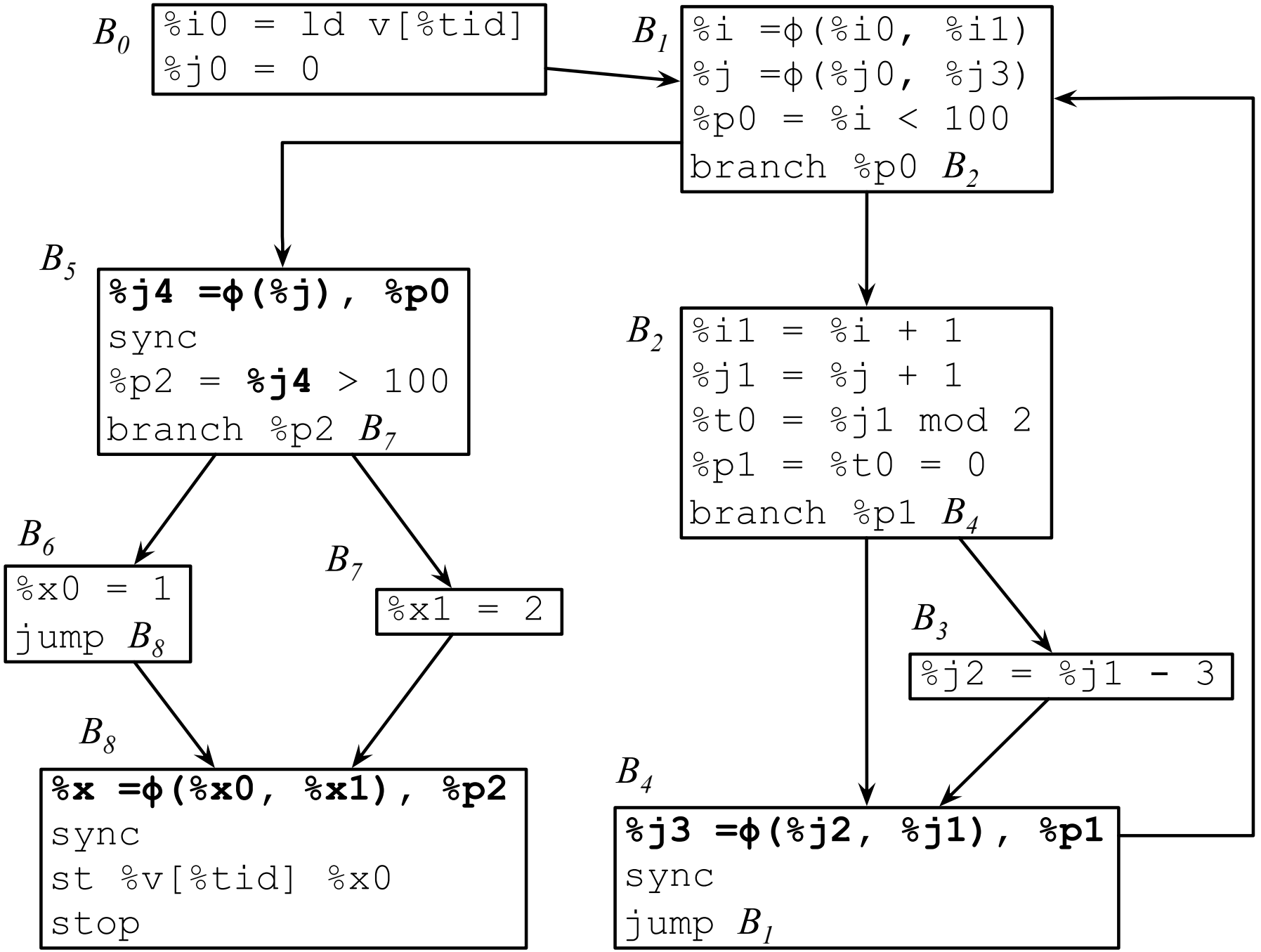




Live Range Splitting

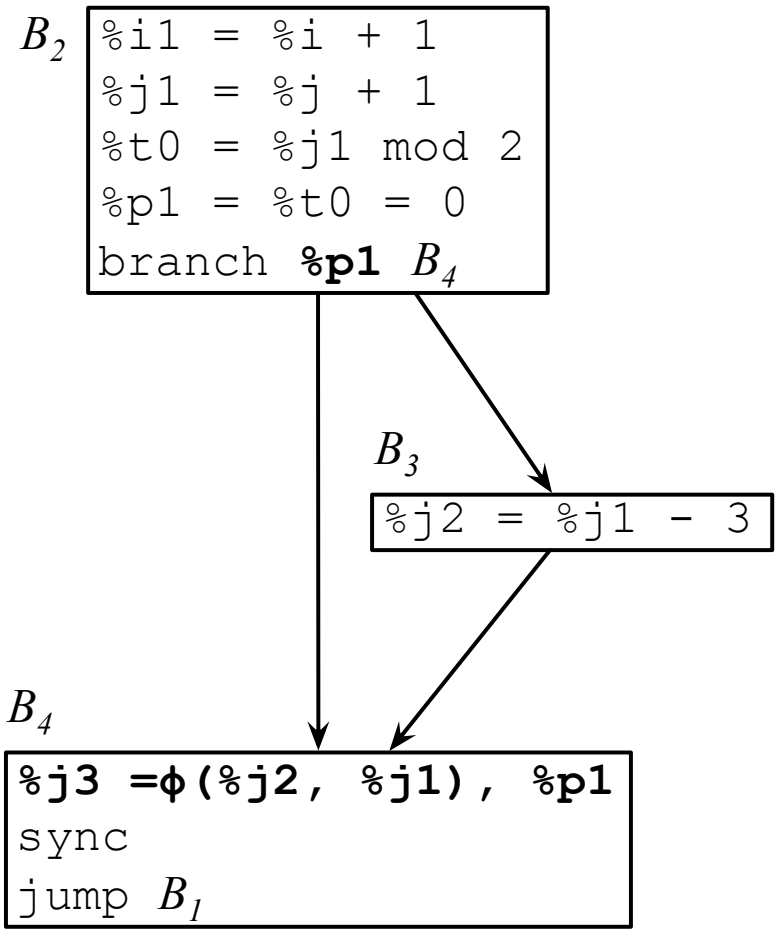


- If a variable is alive outside the IR(%p), then we split its live range, with a unary phi-function.


B_0
%i0 = ld v[%tid]
%j0 = 0
B_1
%i =ϕ(%i0, %i1)
%j =ϕ(%j0, %j3)
%p0 = %i < 100
branch %p0 B_2
B_5
%j4 =ϕ(%j), %p0
sync
%p2 = %j4 > 100
branch %p2 B_7
B_2
%i1 = %i + 1
%j1 = %j + 1
%t0 = %j1 mod 2
%p1 = %t0 = 0
branch %p1 B_4
B_6
%x0 = 1
jump B_8
B_7
%x1 = 2
B_3
%j2 = %j1 - 3
B_8
%x =ϕ(%x0, %x1), %p2
sync
st %v[%tid] %x0
stop
B_4
%j3 =ϕ(%j2, %j1), %p1
sync
jump B_1

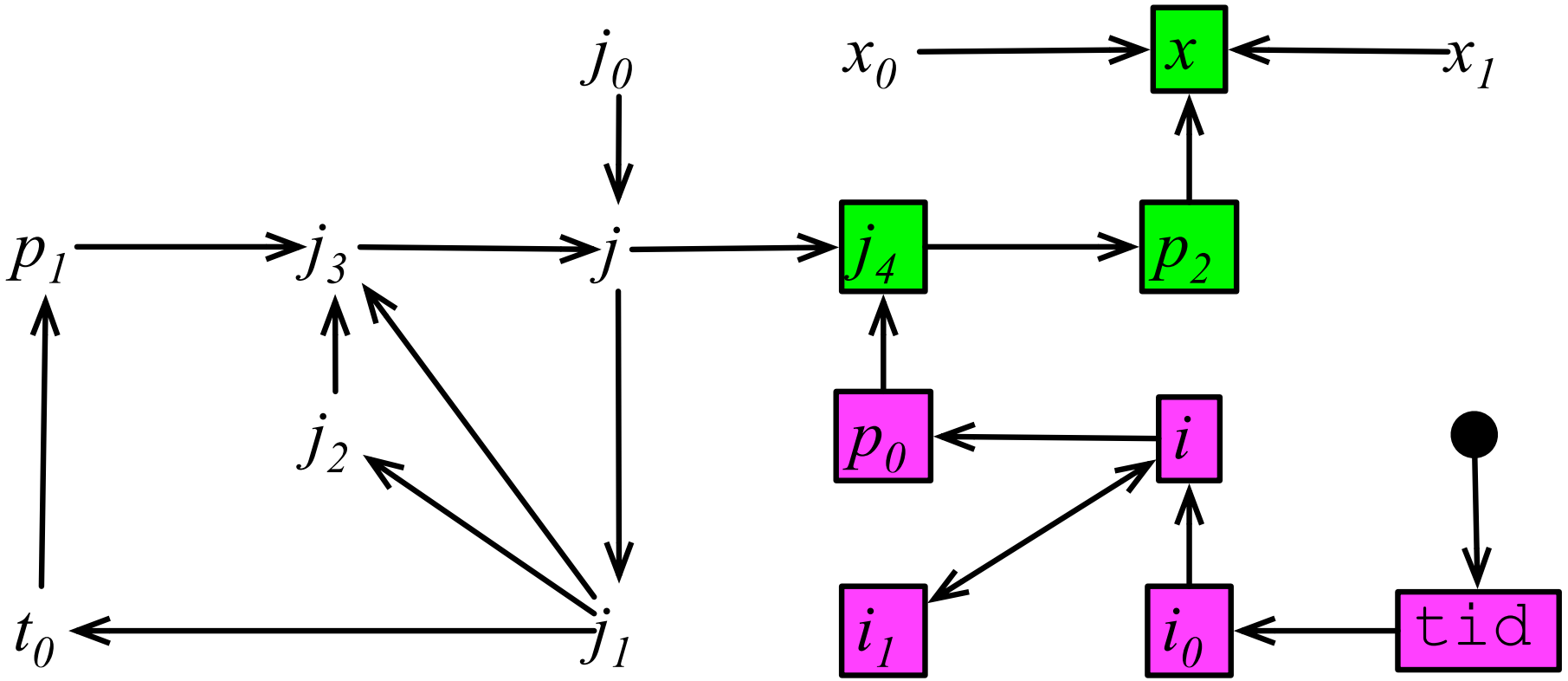
Gated Static Single Assignment Form

- The program representation that we just have produced has a name: *Gated Static Single Assignment* (GSA) form.
- It was invented on the early 90's♡, to take into account control dependences in static program analysis.
- Basically, the GSA form predicates each phi-function with the predicate that controls how that phi-function assigns its parameters to the variable that it defines.



♡: The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages, PLDI pp 257-271 (1990)

Control dependence = data dependence



Compilers Laboratory

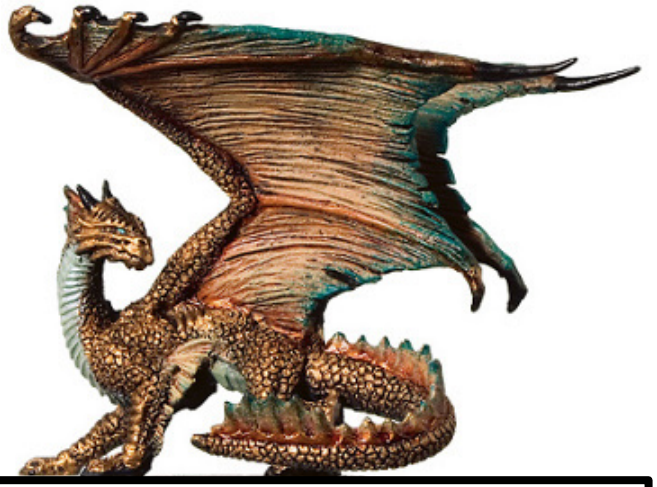
Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Divergence Optimizations

DCC 888

Many Optimizations Rely on Divergence Information

- Barrier elimination.
- Thread re-location.
- Register Allocation.
- Branch fusion.



Let developers worry about their *algorithms*, while the **compiler** takes care of *divergences*.

Elimination of Synchronization Barriers

"All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the **`uni`** suffix. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent."

PTX programmer's manual

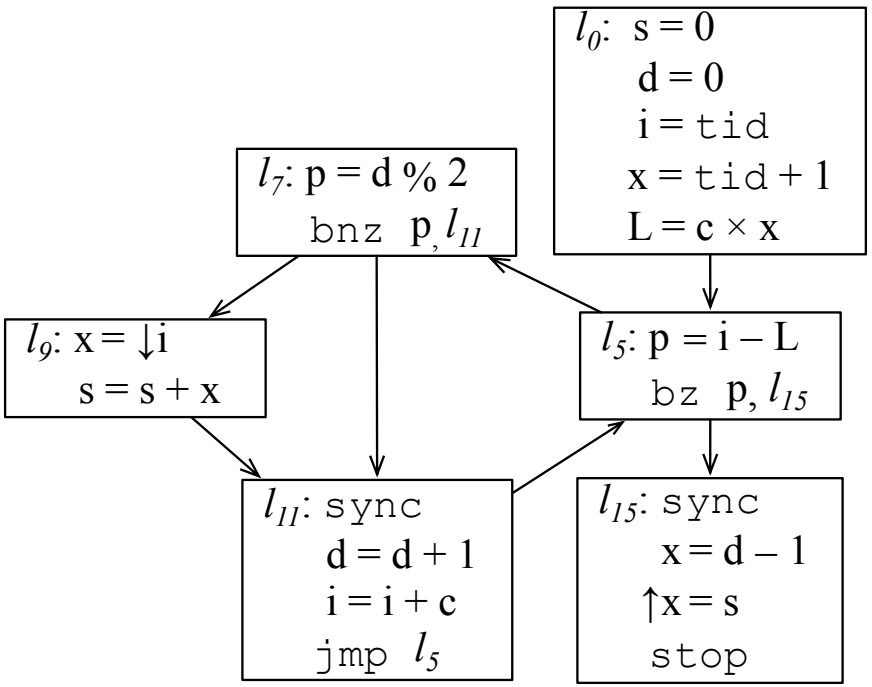
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Register Allocation

- Register allocation is the problem of finding storage locations to the variables used in a program.
- Variables can be kept in *registers*, or they can be mapped to *memory*.
- Registers are very fast, but only a handful of them is available to each GPU thread.
- The memory is plenty, but its access is slow.
- Variables that need to be mapped into memory are called *spills*.

We have roughly three kinds of memory in a GPU: shared, local and global. Where are spills usually placed?

Linear Scan Register Allocation

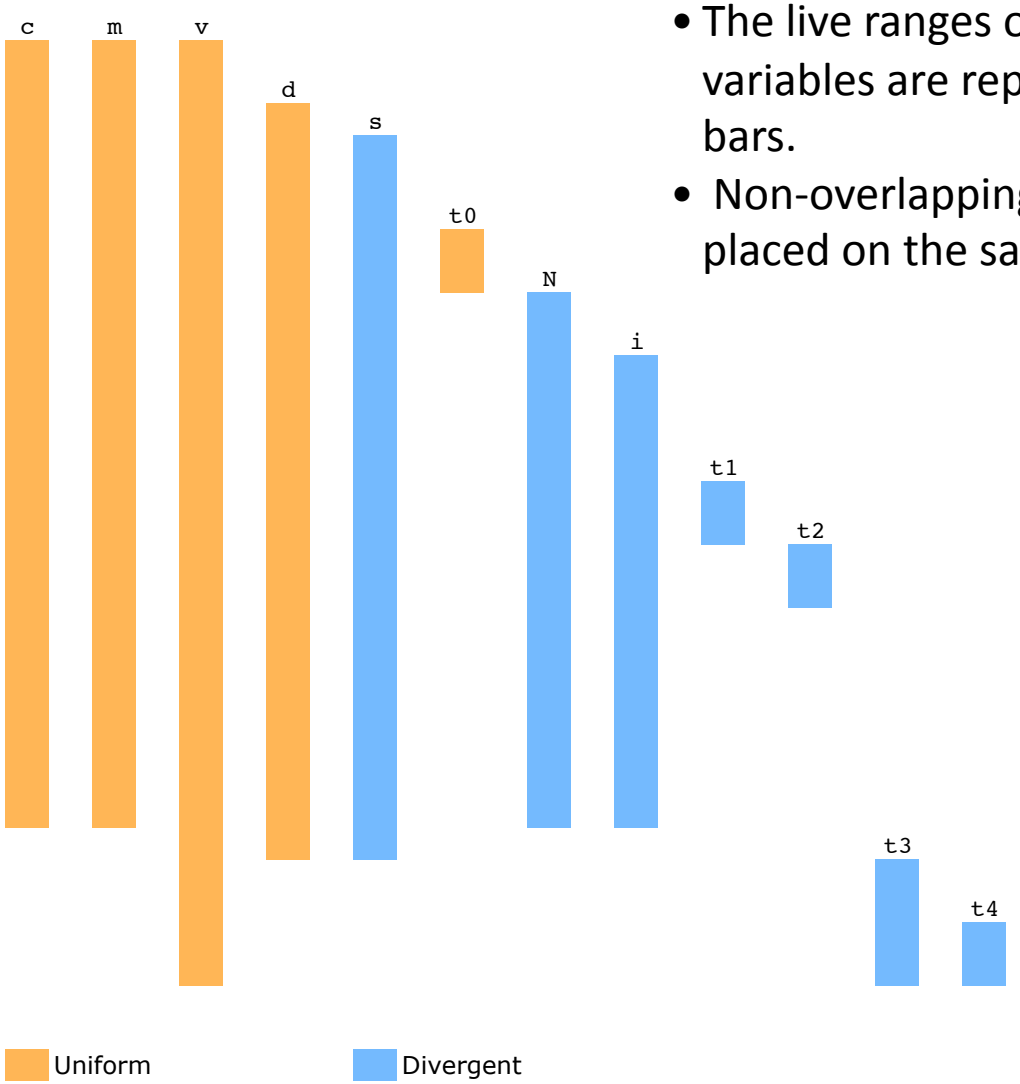


How can we do register allocation in the program above?

- This register allocator was invented by Polleto and Sarkar to suit the needs of just-in-time compilers.
- The allocator represents variables as intervals.
- Non-overlapping intervals can be placed in the same location.
- The coloring of the interval graph gives us a valid register allocation.

An Instance of the Register Allocation Problem

```
L0   d = 0
L1   s = 0.0F
L2   t0 = c * c
L3   N = tid + t0
L4   i = tid
L5   if i < N jp L12
L6   t1 = i * 4
L7   ld.global [m+t1] t2
L8   s = t2 + s
L9   d = d + 1
L10  i = i + c
L11  jp L5
L12  t3 = s / d
L13  t4 = tid*4
L14  st.global t3 [v+t4]
```



- The live ranges of the variables are represented as bars.
- Non-overlapping bars can be placed on the same register

Placing Spills in the Local Memory

Program	register file PE0 r0	r1	r2	register file PE1 r0	r1	r2	local PE0 0	1	2	local PE1 0	1	2	global 0	1	2
d = 0													c	m	v
st.local d [1]	d			d									c	m	v
s = 0.0F	d			d				d			d		c	m	v
st.local s [0]	d	s		d	s			d			d		c	m	v
ld.global [0] c	d	s		d	s		s	d		s	d		c	m	v
t0 = c * c	d	s	c	d	s	c	s	d		s	d		c	m	v
N = tid + t0	t0	s	c	t0	s	c	s	d		s	d		c	m	v
st.local N [2]	t0	s	N	t0	s	N	s	d		s	d		c	m	v
i = tid	t0	s	N	t0	s	N	s	d	N	s	d	N	c	m	v
ld.local [2] N	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
if i < N jp L24	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
t1 = i * 4	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
ld.global [1] m	i	s	t1	i	s	t1	s	d	N	s	d	N	c	m	v
ld.global [m+t1] t2	i	m	t1	i	m	t1	s	d	N	s	d	N	c	m	v
ld.local [0] s	i	m	t2	i	m	t2	s	d	N	s	d	N	c	m	v
s = t2 + s	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
st.local s [0]	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
ld.local [1] d	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
d = d + 1	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
st.local d [1]	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
ld.global [0] c	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
i = i + c	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
jp L9	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
ld.local [1] d	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
t3 = s / d	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
t4 = tid*4	t3	s	d	t3	s	d	s	d	N	s	d	N	c	m	v
ld.global [2] v	t3	t4	d	t3	t4	d	s	d	N	s	d	N	c	m	v
st.global t3 [v+t4]	t3	t4	v	t3	t4	v	s	d	N	s	d	N	c	m	v

Traditionally, variables are placed in the local memory, which is private to each thread.

Why are spills traditionally placed in the local memory?

There is some wasting here. How could we improve this allocation?

Placing Uniform Spills in the Shared Memory

	Program	register file PE0			register file PE1			Shared PE0	Local		global		
		r0	r1	r2	r0	r1	r2	0	0	1	0	1	2
L0	d = 0										c	m	v
L1	**st.shared d [0]**	d			d						c	m	v
L2	s = 0.0F	d			d			d			c	m	v
L3	**st.local s [0]**	d	s		d	s		d			c	m	v
L4	**ld.global [0] c**	d	s		d	s		d	s		c	m	v
L5	t0 = c * c	d	s	c	d	s	c	d	s		c	m	v
L6	N = tid + t0	t0	s	c	t0	s	c	d	s		c	m	v
L7	**st.shared t0 [1]**	t0	s	N	t0	s	N	d	s		c	m	v
L8	i = tid	t0	s	N	t0	s	N	d	s	N	c	m	v
L9	**ld.shared [1] t0**	i	s	N	i	s	N	d	s	N	c	m	v
L10	**N = tid + t0**	i	s	t0	i	s	t0	d	s	N	c	m	v
L11	if i < N jp L24	i	s	N	i	s	N	d	s	N	c	m	v
L12	t1 = i * 4	i	s	N	i	s	N	d	s	N	c	m	v
L13	**ld.global [1] m**	i	s	t1	i	s	t1	d	s	N	c	m	v
L14	ld.global [m+t1] t2	i	m	t1	i	m	t1	d	s	N	c	m	v
L15	**ld.local [0] s**	i	m	t2	i	m	t2	d	s	N	c	m	v
L16	s = t2 + s	i	s	t2	i	s	t2	d	s	N	c	m	v
L17	**st.local s [0]**	i	s	t2	i	s	t2	d	s	N	c	m	v
L18	**ld.shared [0] d**	i	s	t2	i	s	t2	d	s	N	c	m	v
L19	d = d + 1	i	s	d	i	s	d	d	s	N	c	m	v
L20	**st.shared d [0]**	i	s	d	i	s	d	d	s	N	c	m	v
L21	**ld.global [0] c**	i	s	d	i	s	d	d	s	N	c	m	v
L22	i = i + c	i	s	c	i	s	c	d	s	N	c	m	v
L23	jp L9	i	s	c	i	s	c	d	s	N	c	m	v
L24	**ld.shared [0] d**	i	s	c	i	s	c	d	s	N	c	m	v
L25	t3 = s / d	i	s	d	i	s	d	d	s	N	c	m	v
L26	t4 = tid*4	t3	s	d	t3	s	d	d	s	N	c	m	v
L27	**ld.global [2] v**	t3	t4	d	t3	t4	d	d	s	N	c	m	v
L28	st.global t3 [v+t4]	t3	t4	v	t3	t4	v	d	s	N	c	m	v

What are the advantages of placing spills in the shared memory?

Is there any disadvantage in this approach?

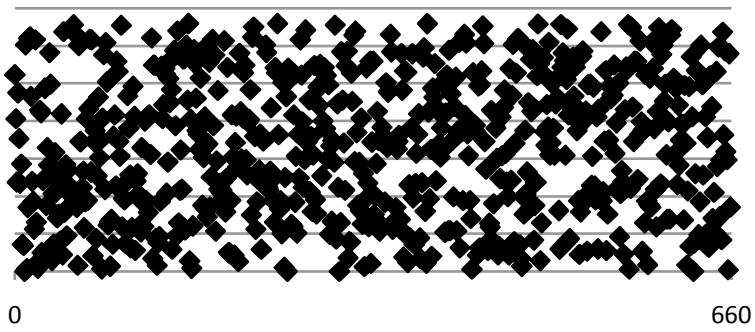
Data Re-location

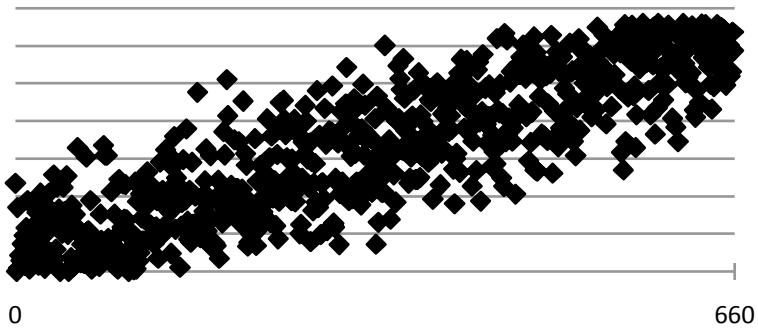
- Sometimes it pays off to spend some time to reorder the data before processing it.
  - Or then we can try to “quasi-sort” these data.
- What could we do to improve the execution of this old acquaintance of ours?

```
__global__ void dec2zero(int* v, int N) {
  int xIndex = blockIdx.x*blockDim.x+threadIdx.x;
  if (xIndex < N) {
    while (v[xIndex] > 0) {
      v[xIndex]--;
    }
  }
}
```

Quasi-Sorting

- Copy data to shared memory
- Each thread sorts small chunks of data, say, four cells
- Scatter the data around the shared memory:
  - Each thread puts its smallest element in the first ¼ of the array, the second smallest in the second ¼ of the array, and so forth.





Quasi-Sorting

```
__global__ static void maxSort1(int * values, int N) {
  // 1) COPY-INTO: Copy data from the values vector into
  // shared memory:

  // 2) SORT: each thread sorts its chunk of data with a
  // small sorting net.

  // 3) SCATTER: the threads distribute their data along
  // the array.

  // COPY-BACK: Copy the data back from the shared
  // memory into the values vector:
}
```

1) Why is it necessary to copy data to the shared memory?
2) Why is the sorting net so good here?
3) How should the data be distributed back?

Quasi-Sorting

```
__global__ static void maxSort1(int * values, int N) {
  // 1) COPY-INTO: copy data from global memory to the shared memory.
   __shared__ int shared[THREAD_WORK_SIZE * NUM_THREADS];
   for (unsigned k = 0; k < THREAD_WORK_SIZE; k++) {
     unsigned loc = k * blockDim.x + threadIdx.x;
     if (loc < N) {
       shared[loc] = values[loc + blockIdx.x * blockDim.x];
     }
   }
  __syncthreads();

  // 2) SORT: each thread sorts its chunk of data with a small sorting net.
  // 3) SCATTER: the threads distribute their data along the array.

  // 4) COPY-BACK: copy data from shared memory to the global memory.
  for (unsigned k = 0; k < THREAD_WORK_SIZE; k++) {
     unsigned loc = k * blockDim.x + threadIdx.x;
     if (loc < N) {
       values[loc + blockIdx.x * blockDim.x] = scattered[loc];
     }
   }
}
```

What is the PRAM complexity of each of these loops?

We manipulate the data in the shared memory to mitigate the large costs of accessing the global memory.

Quasi-Sorting

```
__global__ static void maxSort1(int * values, int N) {
  // 1) COPY-INTO: copy data from global memory to the shared memory.
  // 2) SORT: each thread sorts its chunk of data with a small sorting net.
    int index1 = threadIdx.x * THREAD_WORK_SIZE;
    int index2 = threadIdx.x * THREAD_WORK_SIZE + 1;
    int index3 = threadIdx.x * THREAD_WORK_SIZE + 2;
    int index4 = threadIdx.x * THREAD_WORK_SIZE + 3;
    if (index4 < N) {
      swapIfNecessary(shared, index1, index3);
      swapIfNecessary(shared, index2, index4);
      swapIfNecessary(shared, index1, index2);
      swapIfNecessary(shared, index3, index4);
      swapIfNecessary(shared, index2, index3);
    }
    __syncthreads();

  // 3) SCATTER: the threads distribute their data along the array.
  // 4) COPY-BACK: copy data from shared memory to the global memory.
}
```

What is the PRAM complexity of this algorithm?

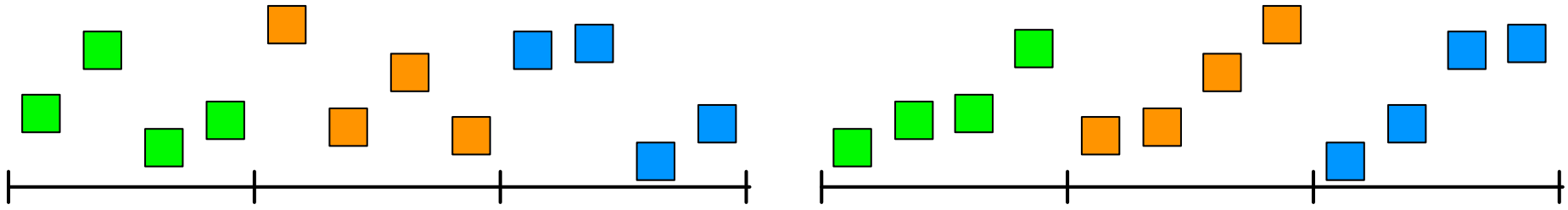
We use a sorting network because all the different threads can execute the same set of comparisons.

Which tradeoffs are involved in the choice of the size of the sorting network?

Quasi-Sorting

```
__global__ static void maxSort1(int * values, int N) {
  // 2) SORT: each thread sorts its chunk of data with a small sorting net.
  int index1 = threadIdx.x * THREAD_WORK_SIZE;
  int index2 = threadIdx.x * THREAD_WORK_SIZE + 1;
  int index3 = threadIdx.x * THREAD_WORK_SIZE + 2;
  int index4 = threadIdx.x * THREAD_WORK_SIZE + 3;
  if (index4 < N) {
    swapIfNecessary(shared, index1, index3);
    swapIfNecessary(shared, index2, index4);
    swapIfNecessary(shared, index1, index2);
    swapIfNecessary(shared, index3, index4);
    swapIfNecessary(shared, index2, index3);
  }
  __syncthreads();
}
```

This step happens in a constant number of steps per thread. After the five comparisons, each chunk of data is sorted.



Quasi-Sorting

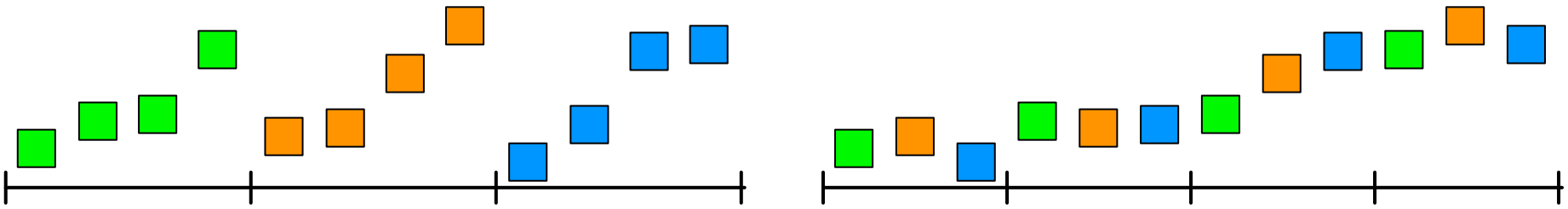
```
__global__ static void maxSort1(int * values, int N) {
  // 1) COPY-INTO: copy data from global memory to the shared memory.

  // 2) SORT: each thread sorts its chunk of data with a small sorting net.

  // 3) SCATTER: the threads distribute their data along the array.
    __shared__ int scattered[THREAD_WORK_SIZE * 300];
     unsigned int nextLoc = threadIdx.x;
     for (unsigned i = 0; i < THREAD_WORK_SIZE; i++) {
       scattered[nextLoc] = shared[threadIdx.x*THREAD_WORK_SIZE + i];
       nextLoc += blockDim.x;
     }
     __syncthreads();

   // 4) COPY-BACK: copy data from shared memory to the global memory.
}
```

Again: what is the PRAM complexity of this algorithm?



A Bit of History

- General purpose graphics processing units come into the spotlight in 2005/06
- Divergence analyses have been independently described by several research groups.
- The core divergence analysis that we describe was invented by Coutinho *et al*.
- The Divergence Aware Register Allocator was designed by Sampaio *et al*.

- Ryoo, S. Rodrigues, C. Baghsorkhi, S. Stone, S. Kirk, D. and Hwu, Wen-Mei. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", PPoPP, p 73-82 (2008)
- Coutinho, B. Diogo, S. Pereira, F and Meira, W. "Divergence Analysis and Optimizations", PACT, p 320-329 (2011)
- Sampaio, D. Martins, R. Collange, S. and Pereira, F. "Divergence Analysis", TOPLAS, 2013.