Program semantics meets architecture:

What if we did not have branches?

Soner Onder
Department of Computer Science
Michigan Technological University

MichiganTech

Prologue

Prologue

I am a dreamer. I dream a lot of things.

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did _not_ have branches?

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did _not_ have branches?

Pure speculation:

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did *not* have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

MichiganTech

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

… and … Yale would be lesser known …

MichiganTech

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

… and … Yale would be lesser known … for his work in branch prediction.

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

… and … Yale would be lesser known … for his work in branch prediction.
Nevertheless, no lesser known!

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

… and … Yale would be lesser known … for his work in branch prediction.
Nevertheless, no lesser known!

… as for me, I would still be giving this talk in honor of his 80-th birthday, except there would be a slight variation in my talk's title:

Prologue

I am a dreamer. I dream a lot of things.

Today, I ask the question:
What if we did <u>not</u> have branches?

Pure speculation:

… Daniel would be doing even better because he would stay in AI instead of working in our merciless field where no good work can escape rejection.

… TAGE predictor would still exist but it would be predicting something else.
(I am suspecting André came up with the name <u>before</u> he invented the predictor).

… and … Yale would be lesser known … for his work in branch prediction.
Nevertheless, no lesser known!

… as for me, I would still be giving this talk in honor of his 80-th birthday, except there would be a slight variation in my talk's title:

Program semantics meets architecture: ***What if we <u>had</u> branches?***

Is this a serious talk?

Is this a serious talk?

Possibly.

Is this a serious talk?

Possibly.

Are you going to be talking about predication?

Is this a serious talk?

Possibly.

Are you going to be talking about predication?

Nihil sub sole novum.

Is this a serious talk?

Possibly.

Are you going to be talking about predication?

Nihil sub sole novum.

Why are you giving this talk then?

Is this a serious talk?

Possibly.

Are you going to be talking about predication?

Nihil sub sole novum.

Why are you giving this talk then?

To give a new perspective.

Is this a serious talk?

Possibly.

Are you going to be talking about predication?

Nihil sub sole novum.

Why are you giving this talk then?

To give a new perspective.

So, let’s start with the question:

Why do we need to use branches in implementing the semantics of our programs?

Turing completeness

Turing completeness

In their seminal paper :

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules, Communications of the ACM (CACM), Volume 9 Issue 5, May 1966 (*),

Corrado Böhm and his student Giuseppe Jacopini proved that *sequencing*, *selection* and *iteration* are sufficient to simulate any Turing machine.

Corrado Böhm, professor emeritus at the University of Rome "La Sapienza", left us on October 23, 2017 at the age of 94.

He has been an exceptionally talented and creative researcher: his results have deeply influenced the development of theoretical computer science.

Turing completeness

In their seminal paper :

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules, Communications of the ACM (CACM), Volume 9 Issue 5, May 1966 (*),

Corrado Böhm and his student Giuseppe Jacopini proved that *sequencing*, *selection* and *iteration* are sufficient to simulate any Turing machine.

Corrado Böhm, professor emeritus at the University of Rome "La Sapienza", left us on October 23, 2017 at the age of 94.

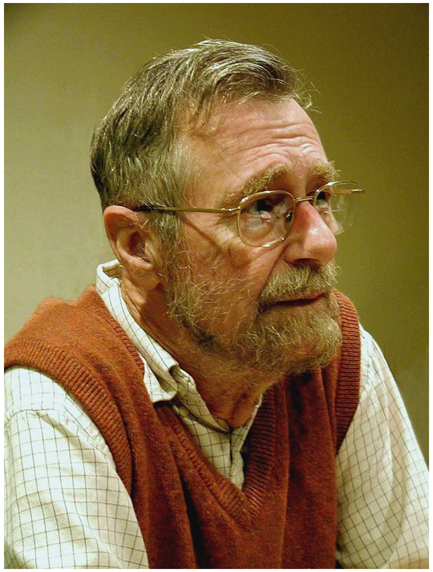
He has been an exceptionally talented and creative researcher: his results have deeply influenced the development of theoretical computer science.

(*) One of two references in Dijkstra's *Go To Statement Considered Harmful* paper.

Turing completeness

Edsger Wybe Dijkstra:

Guiseppe Jacopini seems to have proved the (logical) superfluousness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. *Then the resulting flow diagram cannot be expected to be more transparent than the original one.(*)*



Go To Statement Considered Harmful, *Communications of the ACM (CACM), Volume 11 Issue 3, March 1968.*

(*) I highlighted the text

Turing completeness

Turing completeness

Question:

Eliminating Go To statements through structured programming does not eliminate branches. We still need to implement structured constructs using branches at the low-level, right?

Turing completeness

Question:

Eliminating Go To statements through structured programming does not eliminate branches. We still need to implement structured constructs using branches at the low-level, right?

Answer:

No, we do not need branches even at the low-level, if we forgo controlling instructions and concentrate only on data values. We only need ***gating*** and ***recursion***.

Turing completeness

Question:

Eliminating Go To statements through structured programming does not eliminate branches. We still need to implement structured constructs using branches at the low-level, right?

Answer:

No, we do not need branches even at the low-level, if we forgo controlling instructions and concentrate only on data values. We only need *gating* and *recursion*.

In the rest of the talk, I'll show that we can ***efficiently*** implement:

1. Sequencing in a data-driven manner
2. Selection using gating functions
3. Iteration through recursion

All *without* branches!

Furthermore, we will end-up with a "*more transparent*" program than the original program!

MichiganTech

Revisiting control-dependence – if-then-else

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

The code

```
i:  If a < b then
j:     k = 5
    else
k:     k = 10
     = k
```

Selection is implemented in the fetch unit by changing PC.

MichiganTech

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

<u>The code</u>

```
i:  If a < b then
j:     k = 5
    else
k:     k = 10
     = k
```

<u>If-converted code</u>

```
 T :  P = a < b
 P :  k = 5
¬P :  k = 10
        = k
```

Selection is implemented in the fetch unit by changing PC.

Selection is implemented by controlling writes using predicates.

MichiganTech

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

The code

```
i:  If a < b then
j:     k = 5
    else
k:     k = 10
     = k
```

If-converted code

```
 T :  P = a < b
 P :  k = 5
¬P :  k = 10
          = k
```

Conditional move

```
k = 5
t  = 10
P = a < b
k = cmov ¬p, t
  = k
```

Selection is implemented in the fetch unit by changing PC.

Selection is implemented by controlling writes using predicates.

Selection is implemented by controlling whether cmov writes.

MichiganTech

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

<u>The code</u>

i: If a < b then
j: k = 5
else
k: k = 10
= k

Selection is implemented in the fetch unit by changing PC.

<u>If-converted code</u>

T : P = a < b
P : k = 5
$\neg$P : k = 10
= k

Selection is implemented by controlling writes using predicates.

<u>Conditional move</u>

k = 5
t = 10
P = a < b
k = cmov $\neg$p, t
= k

Selection is implemented by controlling whether cmov writes.

<u>Gating</u>

$k_0 = 5$
$k_1 = 10$
P = a < b
$k_2 = \Psi_p\ (k_0, k_1)$
$= k_2$

Selection is implemented by a MUX at the ALU inputs.

Revisiting control-dependence – if-then-else

It is well-known that branch instructions implement control-dependencies. They can be converted to data-dependencies through if-conversion (predication).

Formally:

An instruction j is control-dependent on i if the execution of j is controlled by i.

<u>The code</u>

i: If a < b then
j: k = 5
else
k: k = 10
= k

Selection is implemented in the fetch unit by changing PC.

<u>If-converted code</u>

T : P = a < b
P : k = 5
¬P : k = 10
= k

Selection is implemented by controlling writes using predicates.

<u>Conditional move</u>

k = 5
t = 10
P = a < b
k = cmov ¬p, t
= k

Selection is implemented by controlling whether cmov writes.

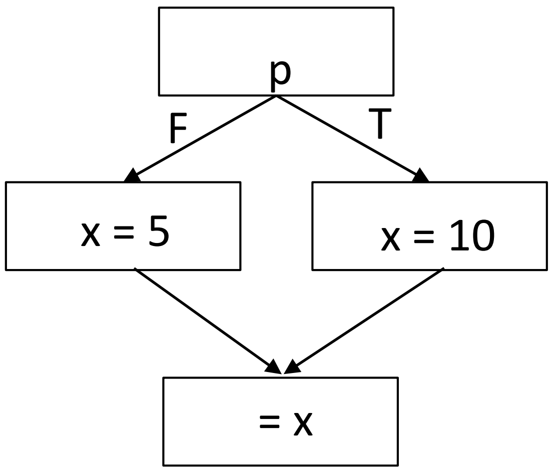
<u>Gating</u>

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p (k_0, k_1)$
= k_2

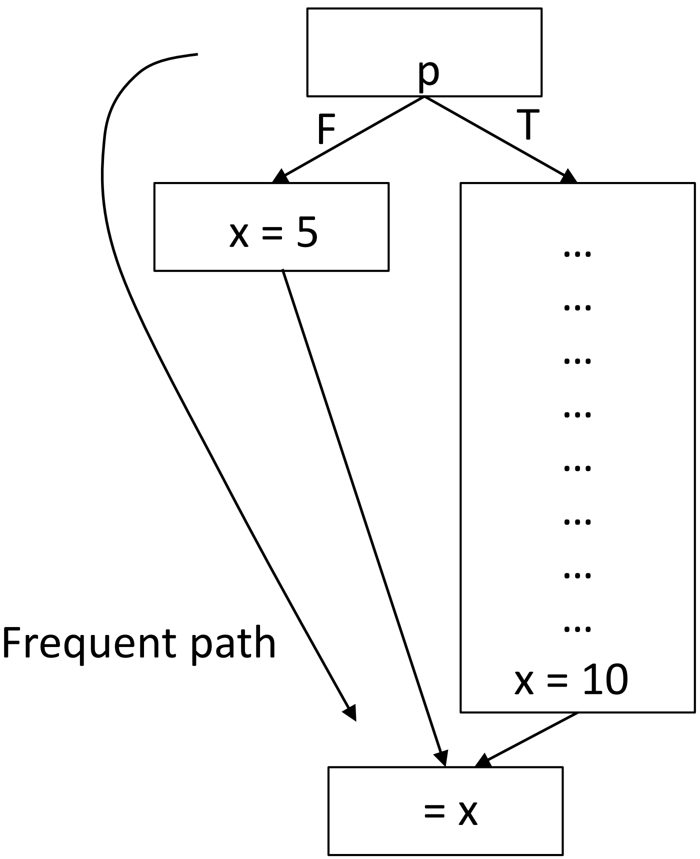
Selection is implemented by a MUX at the ALU inputs.

Both paths are fetched and executed.

Executing both paths



(a)



(b)

Revisiting control-dependence - Gating

Revisiting control-dependence - Gating

<u>If-converted code</u>

T : P = a < b
P : k = 5
$\neg$P : k = 10
       = k

In eliminating branches through if-conversion, regardless of the type of the conversion, the ability to point to the instruction whose result will be used is lost.

<u>Gating</u>

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,}k_1)$
    = k_2

As a result, instructions providing the value for each path must be unconditionally executed, in anticipation that its value might be needed.

Revisiting control-dependence - Gating

<u>If-converted code</u>

T : P = a < b
P : k = 5
┐P : k = 10
= k

In eliminating branches through if-conversion, regardless of the type of the conversion, the ability to point to the instruction whose result will be used is lost.

<u>Gating</u>

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p (k_0, k_1)$
= k_2

As a result, instructions providing the value for each path must be unconditionally executed, in anticipation that its value might be needed.

Corollary:

1. Branch instructions route data through memory by selecting instructions. Only the fetched (i.e., selected) instruction can update memory. The word <u>*selection*</u> in Böhm and Jacopini's paper must be understood as <u>*selecting among data values*</u> (even though that is not exactly what they had shown).

2. Predication controls who can update memory, i.e., who should write, but it cannot select instructions.

3. Gating selects among data values, i.e., who should be read, but it cannot select instructions – (is that true?)

MichiganTech

Executing in “Reverse”

Executing in “Reverse”

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,} k_1)$
= k_2 + …

This code is in *single-assignment form*, i.e., every variable is assigned only once.

Executing in "Reverse"

k_0 = 5
k_1 = 10
$P = a < b$
$k_2 = \Psi_p\ (k_{0,} k_1)$
$\quad = k_2 + \ldots$

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,}k_1)$
= k_2 + …

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

k_0() { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
k_2() { return $\Psi_{p()}\ (k_0(), k_1())$ }
… { return k_2() + … }

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_0, k_1)$
= k_2 + …

This code is in *single-assignment form*, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}\ (k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

MichiganTech

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,} k_1)$
    = k_2 + …

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}\ (k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

MichiganTech

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,}k_1)$
   = k_2 + …

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}$ $(k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

MichiganTech

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\,(k_{0,}k_1)$
= k_2 + …

This code is in *<u>single-assignment form</u>*, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}$ $(k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

MichiganTech

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_0, k_1)$
= k_2 + …

This code is in *single-assignment form*, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return Ψ T $(k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_0, k_1)$
= k_2 + …

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}$ T $(k_0(), k_1())$ }
… { return $k_2()$ + … }

Evaluating the value of k_2

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\,(k_{0,}k_1)$
    = k_2 + …

This code is in *single-assignment form*, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return Ψ T (5), $k_1()$) }

… { return $k_2()$ + … }

Evaluating the value of k_2

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,} k_1)$
= k_2 + …

This code is in ***single-assignment form***, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

k_0() { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

k_2() { return $\Psi_{p()}\ (k_0(), k_1())$ }

… { return 5) + … }

MichiganTech

Executing in "Reverse"

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\,(k_{0,}k_1)$
  = k_2 + …

This code is in *single-assignment form*, i.e., every variable is assigned only once.

Instead of thinking each of the statements as assignments to memory locations, if we think of them to be **single instruction functions**, the variable name becomes the name of the function, and a reference to the name becomes an instruction pointer. Hence we obtain:

$k_0()$ { return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}\,(k_0(), k_1())$ }
 … { return 5) + … }

This is a functional program and the execution you witnessed is Lazy evaluation!

Only the necessary path has been fetched and executed, and there are no branches, only "function calls".

Two function calls per instruction

E/F	Value

Instruction
k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,}k_1)$
$=k_2$ + …

Two function calls per instruction

E/F	Value	Instruction
		k_0 = 5
		k_1 = 10
		P = a < b
		$k_2 = \Psi_p\ (k_{0,} k_1)$
		$= k_2$ + …

k_0() { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

k_2() { return $\Psi_{p()}\ (k_0(), k_1())$ }

… { return $k_2()$ + … }

Two function calls per instruction

E/F	Value	Instruction
		$k_0 = 5$
		$k_1 = 10$
		$P = a < b$
		$k_2 = \Psi_p\ (k_0, k_1)$
		$= k_2 + \ldots$

$k_0() \{\ \text{return } 5\ \}$

$k_1\ () \{\ \text{return } 10\ \}$

$P() \quad \{\ \text{return } a() < b()\ \}$

$k_2() \ \{\ \text{return } \Psi_{p()}\ (k_0(), k_1())\ \}$

$\ldots \quad \{\ \text{return } k_2() + \ldots \quad \}$

Evaluating the value of k_2

Two function calls per instruction

E/F	Value	Instruction
		k_0 = 5
		k_1 = 10
		$P = a < b$
		$k_2 = \Psi_p\ (k_{0,} k_1)$
		$= k_2 + \ldots$

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return $a() < b()$ }

$k_2()$ { return $\Psi_{p()}\ (k_0(), k_1())$ }

… { return $k_2()$ + … }

Evaluating the value of k_2

Two function calls per instruction

E/F	Value	Instruction
		k_0 = 5
		k_1 = 10
		P = a < b
		$k_2 = \Psi_p\ (k_{0,} k_1)$
		$= k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return $\Psi_{p()}$ $(k_0(), k_1())$ }

… { return $k_2()$ + … }

Evaluating the value of k_2

MichiganTech

Two function calls per instruction

E/F	Value	Instruction
		k_0 = 5
		k_1 = 10
1	1	P = a < b
		$k_2 = \Psi_p\ (k_{0,} k_1)$
		$=k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return $\Psi_{p()}$ $(k_0(), k_1())$ }

… { return $k_2()$ + … }

Evaluating the value of k_2

Two function calls per instruction

E/F	Value	Instruction
		k_0 = 5
		k_1 = 10
1	1	P = a < b
		$k_2 = \Psi_p\ (k_{0,} k_1)$
		$= k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return $\Psi_p()$ T $(k_0(), k_1())$ }

… { return $k_2()$ + … }

Evaluating the value of k_2

Two function calls per instruction

E/F	Value
1	1

Instruction
k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,} k_1)$
$=k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return Ψ T $(k_0(), k_1())$ }

… { return $k_2()$ + … }

Evaluating the value of k_2

Two function calls per instruction

E/F	Value	Instruction
1	5	k_0 = 5
		k_1 = 10
1	1	P = a < b
		$k_2 = \Psi_p\ (k_0, k_1)$
		$= k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return $\Psi_p()$ [T] ($k_0()$ [5]), $k_1()$) }

… { return $k_2()$ + … }

Evaluating the value of k_2

Two function calls per instruction

E/F	Value	Instruction
1	5	k_0 = 5
		k_1 = 10
1	1	P = a < b
1	5	$k_2 = \Psi_p\ (k_{0,} k_1)$
		$= k_2$ + …

$k_0()$ { return 5 }

k_1 () { return 10 }

P() { return a() < b() }

$k_2()$ { return $\Psi_{p()}\ (k_0(), k_1())$ }

… { return 5) + … }

Two function calls per instruction

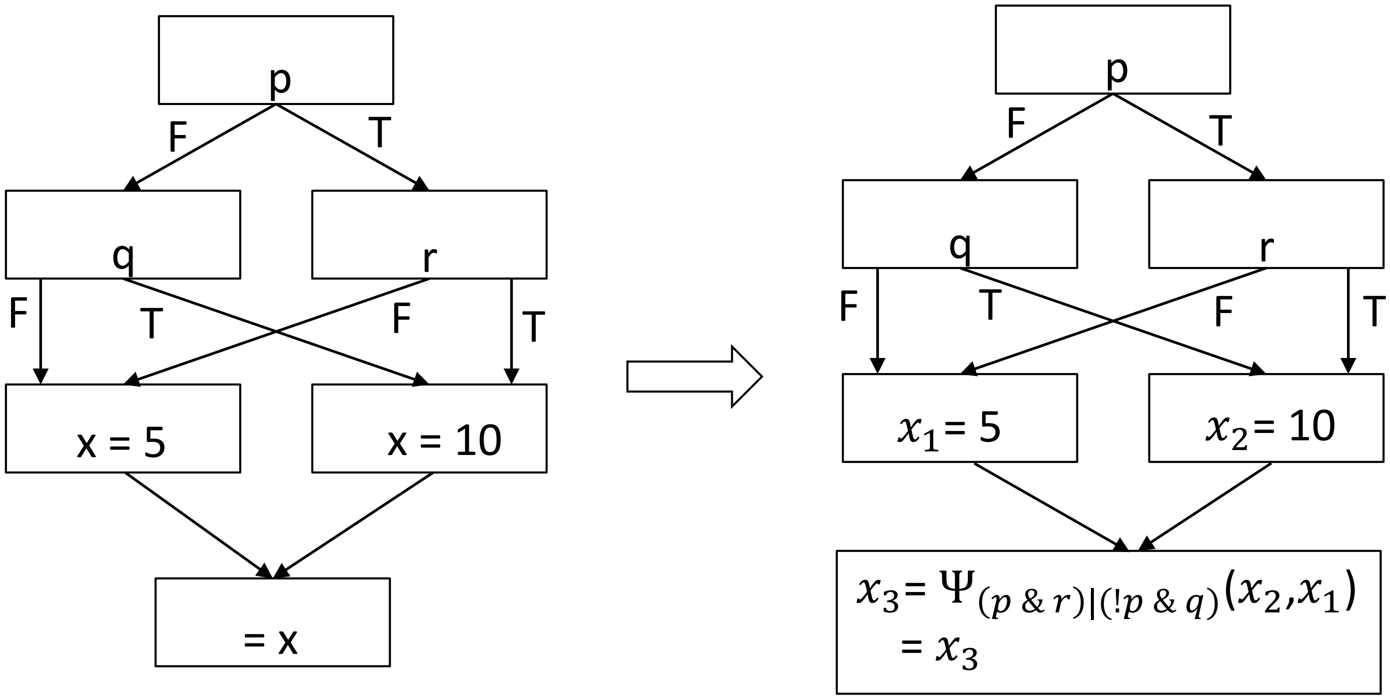
E/F	Value	Instruction
1	5	k_0 = 5
		k_1 = 10
1	1	P = a < b
1	5	$k_2 = \Psi_p\ (k_0, k_1)$
		$=k_2$ + …

$k_0() \{$ return 5 }
k_1 () { return 10 }
P() { return a() < b() }
$k_2()$ { return $\Psi_{p()}\ (k_0(), k_1())$ }
… { return 5) + … }

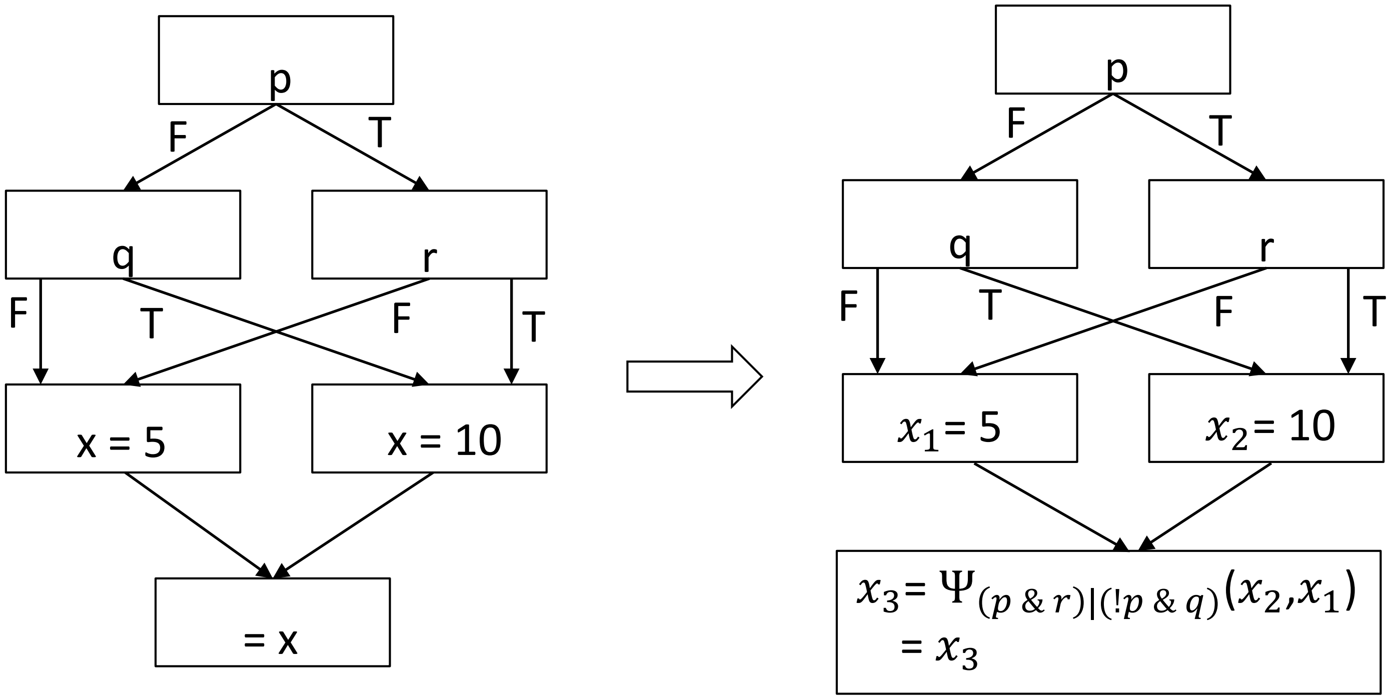
A "function call" is nothing but reading a memory location. If the location is full, it returns the data value. If it is empty, it evaluates the instruction, stores the value and makes the location full.

MichiganTech

A more complicated example



A more complicated example



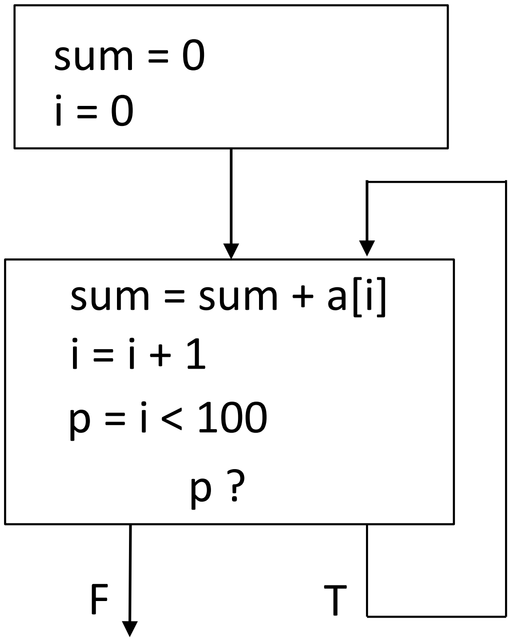
Gating:

1. It does not matter how complicated the control-flow structure is. Unlike branches, predicate expressions can be evaluated in parallel.

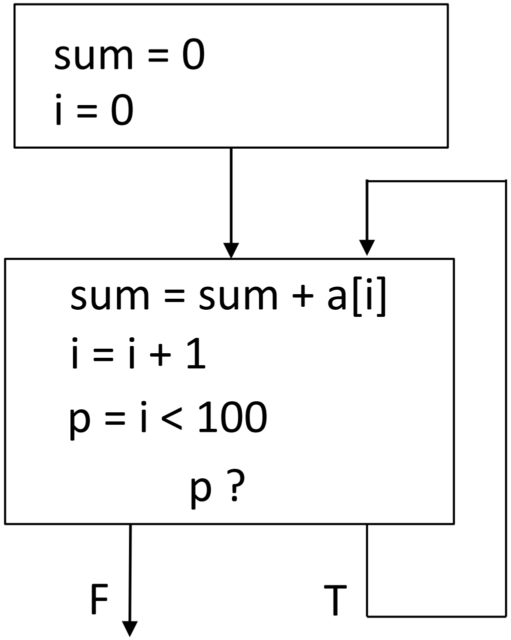
2. If (p &r) | (!p & q) is true, we'll fetch and execute x_2= 10, otherwise we'll fetch x_1= 5.

How about loops?

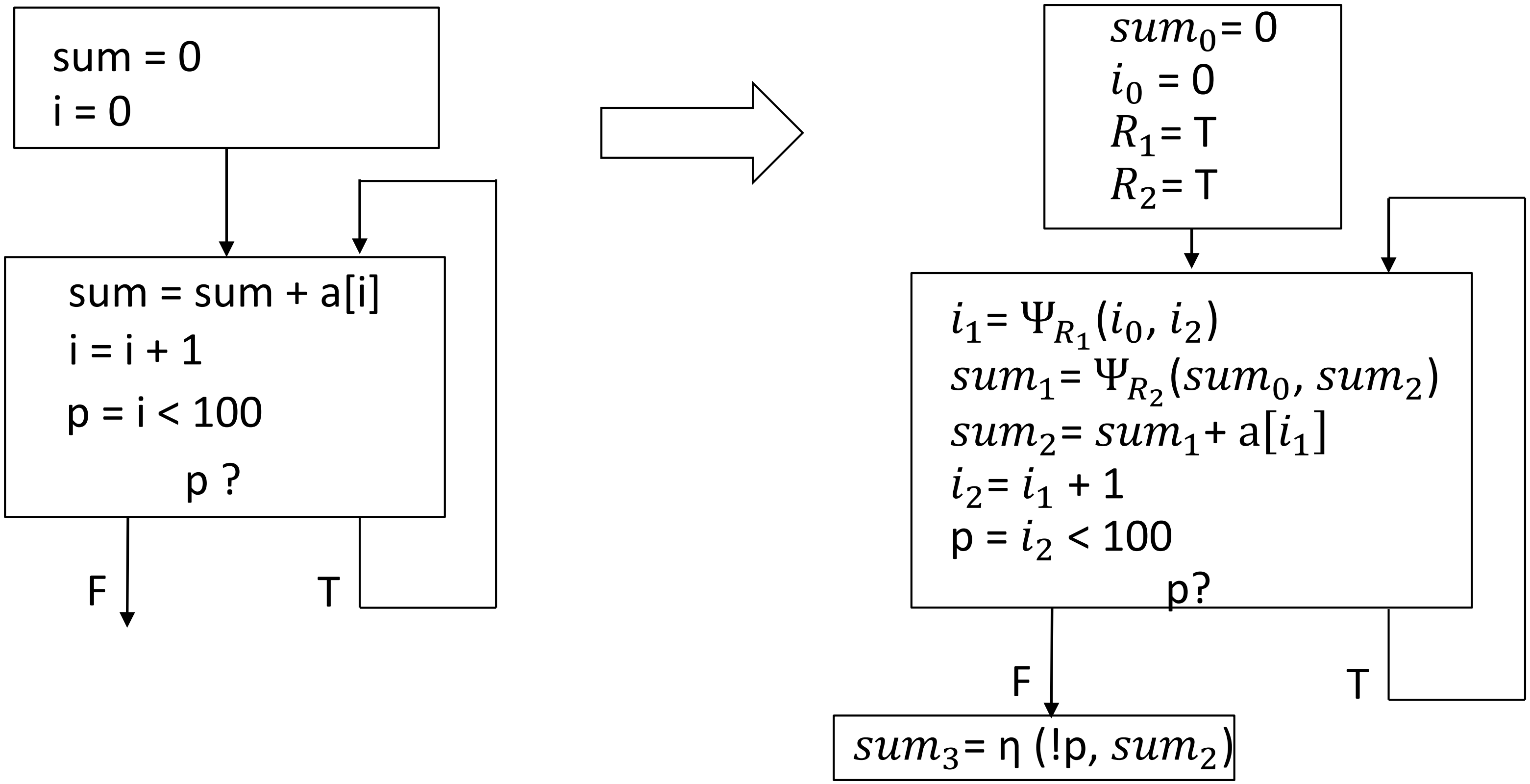
How about loops?



How about loops?

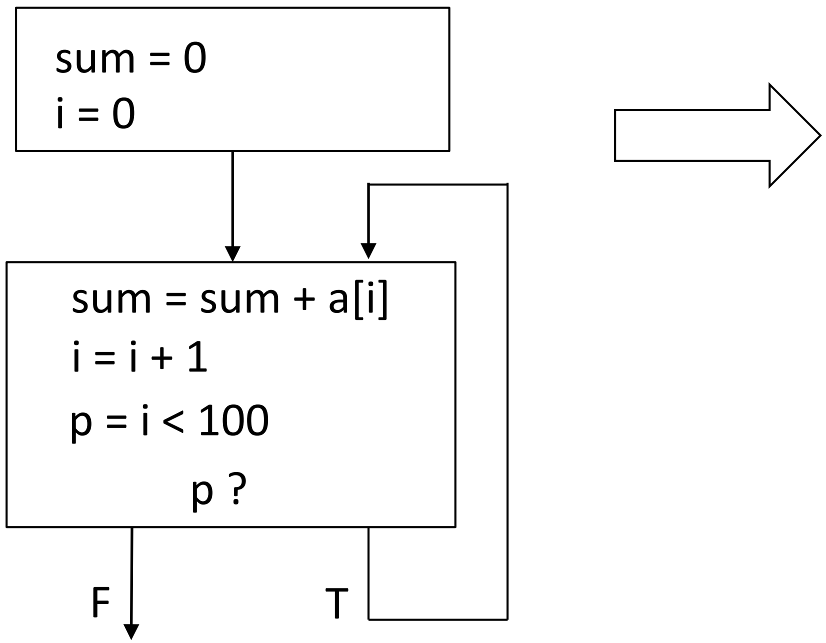


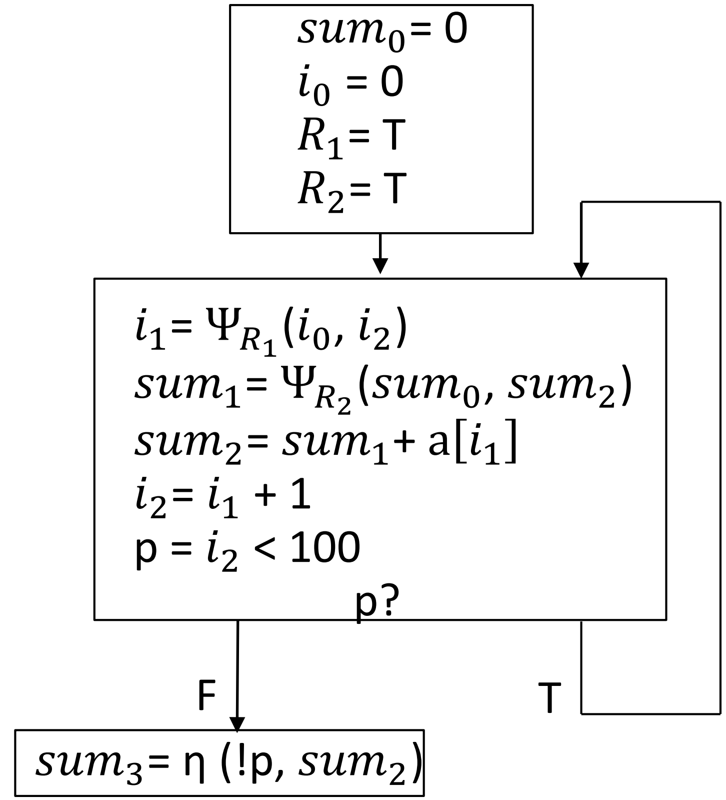
How about loops?



MichiganTech

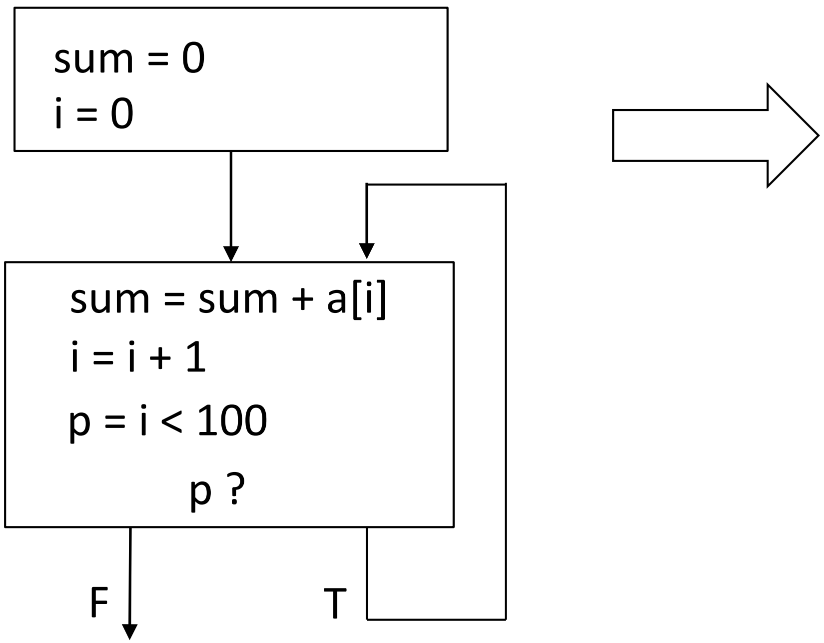
How about loops?

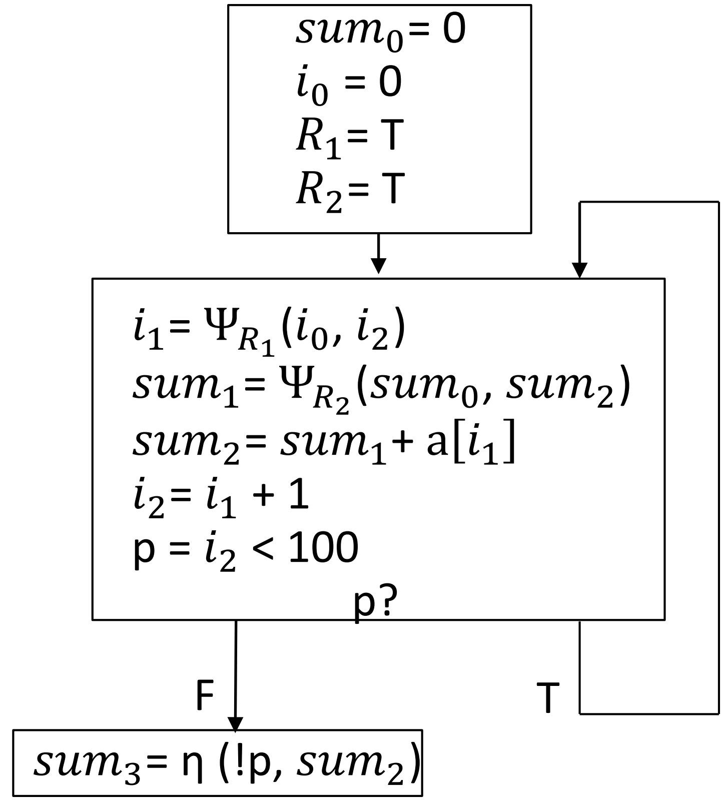




R_1 and R_2 are read-once predicates

How about loops?





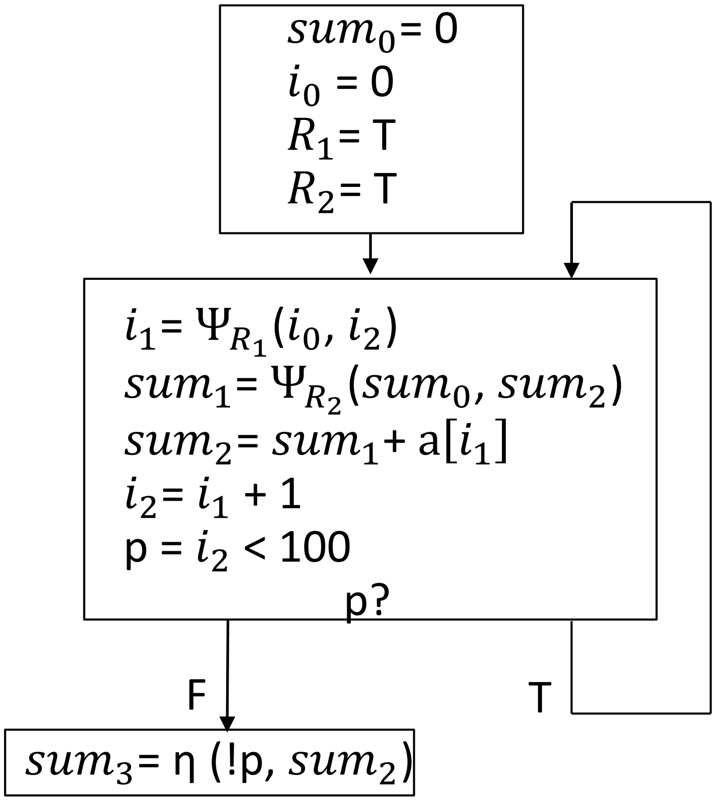
R_1 and R_2 are read-once predicates

The loop in single-assignment form facilitates the conversion to recursion:

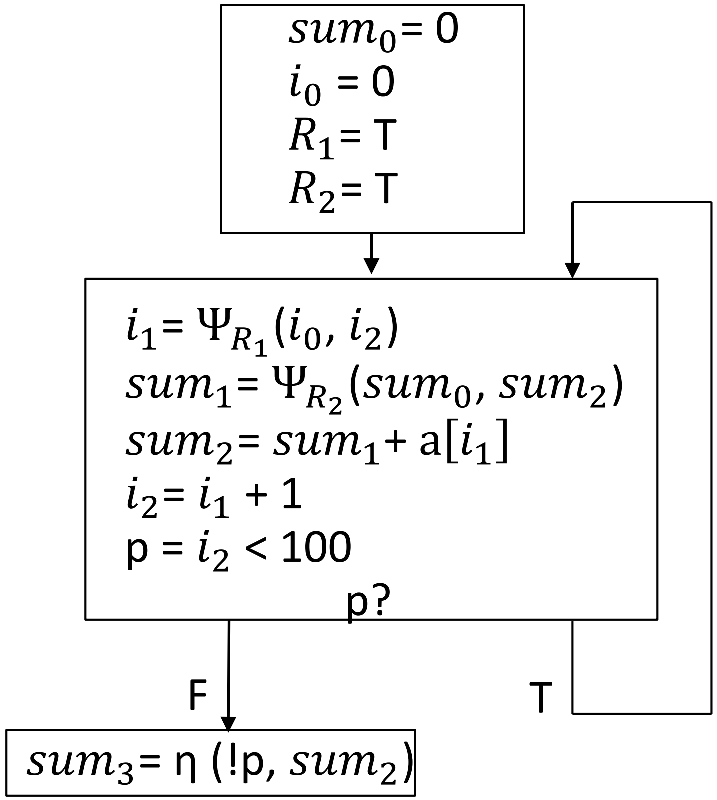
1. If we keep branches, the code is forward executable.
2. Alternatively, we can drop the branches and call the exit function of the loop.
3. The exit function "iterates" until the predicate becomes true.

Drop branches and (just for fun) shuffle instructions

Drop branches and (just for fun) shuffle instructions

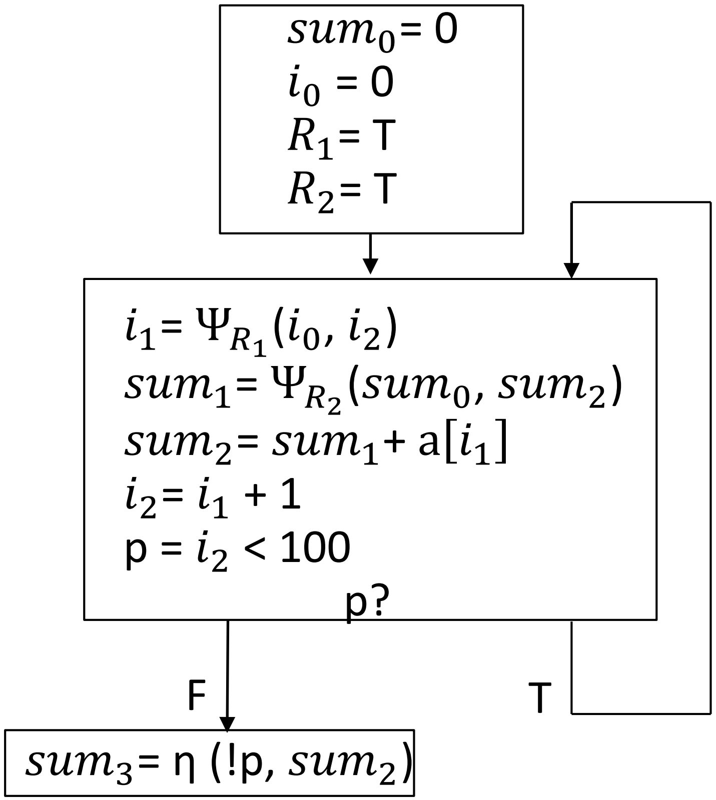


Drop branches and (just for fun) shuffle instructions



sum_2= sum_1+ a[i_1]
i_0 = 0
R_1= T
i_1= $\Psi_{R_1}(i_0, i_2)$
Entry point → sum_3= η (!p, sum_2)
p = i_2 < 100
i_2= i_1 + 1
sum_1= $\Psi_{R_2}(sum_0, sum_2)$
R_2= T
sum_0= 0

Drop branches and (just for fun) shuffle instructions



Entry point →

$sum_2 = sum_1 + a[i_1]$

$i_0 = 0$

$R_1 = T$

$i_1 = \Psi_{R_1}(i_0, i_2)$

$sum_3 = \eta\ (!p, sum_2)$

$p = i_2 < 100$

$i_2 = i_1 + 1$

$sum_1 = \Psi_{R_2}(sum_0, sum_2)$

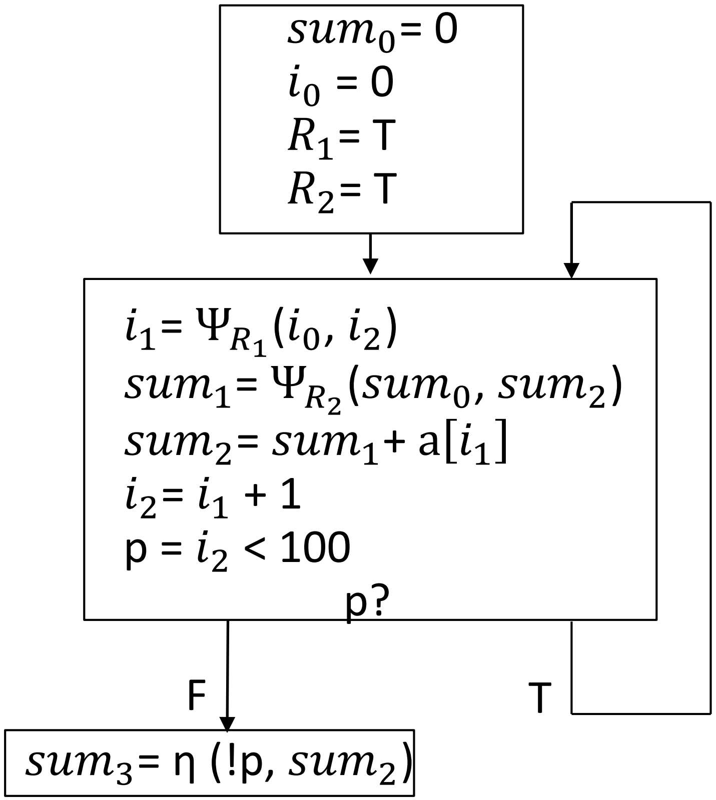
$R_2 = T$

$sum_0 = 0$

Data-driven sequencing

MichiganTech

Drop branches and (just for fun) shuffle instructions

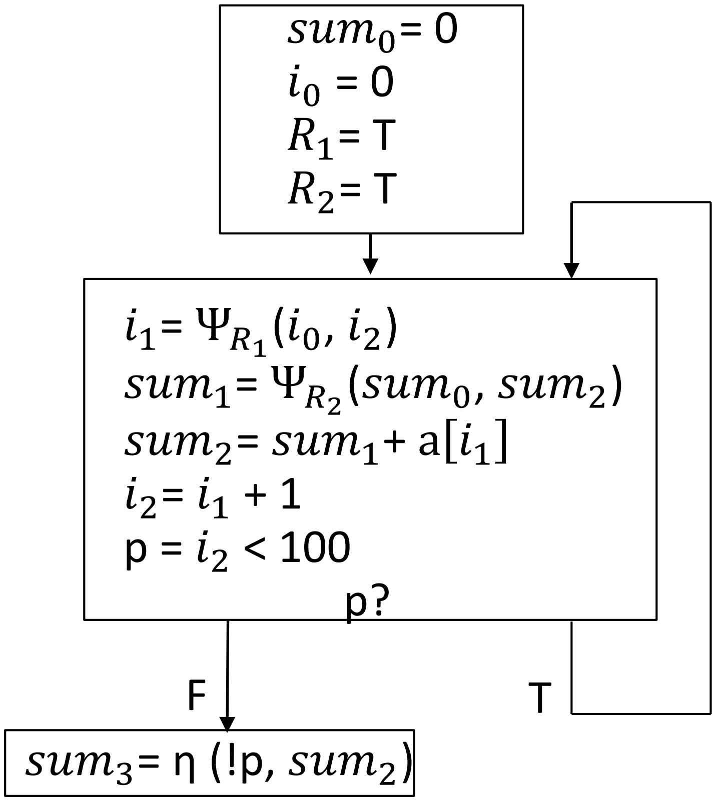


$sum_2 = sum_1 + a[i_1]$
$i_0 = 0$
$R_1 = T$
$i_1 = \Psi_{R_1}(i_0, i_2)$
Entry point → $sum_3 = \eta\ (!p, sum_2)$
$p = i_2 < 100$
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
$R_2 = T$
$sum_0 = 0$

$sum_3 = \eta\ (!p, sum_2)$

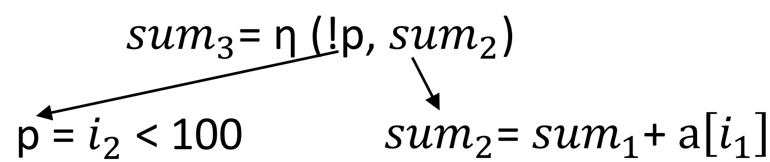
Data-driven sequencing

Drop branches and (just for fun) shuffle instructions



Entry point →

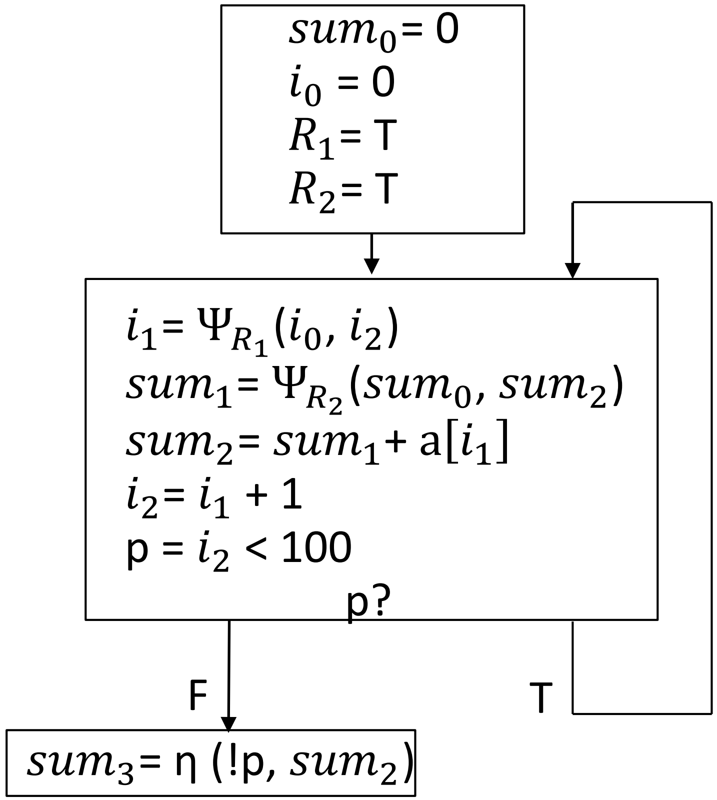
$sum_2 = sum_1 + a[i_1]$
$i_0 = 0$
$R_1 = T$
$i_1 = \Psi_{R_1}(i_0, i_2)$
$sum_3 = \eta\ (!p, sum_2)$
$p = i_2 < 100$
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
$R_2 = T$
$sum_0 = 0$



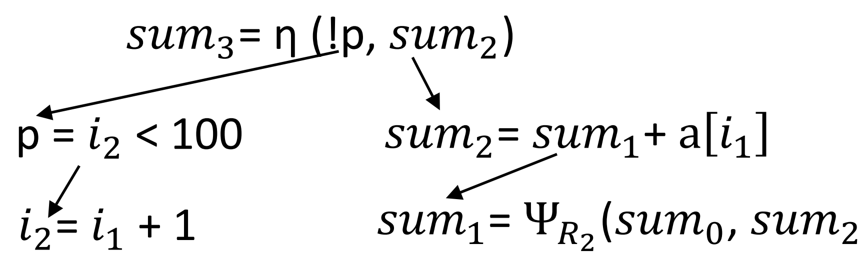
Data-driven sequencing

MichiganTech

Drop branches and (just for fun) shuffle instructions

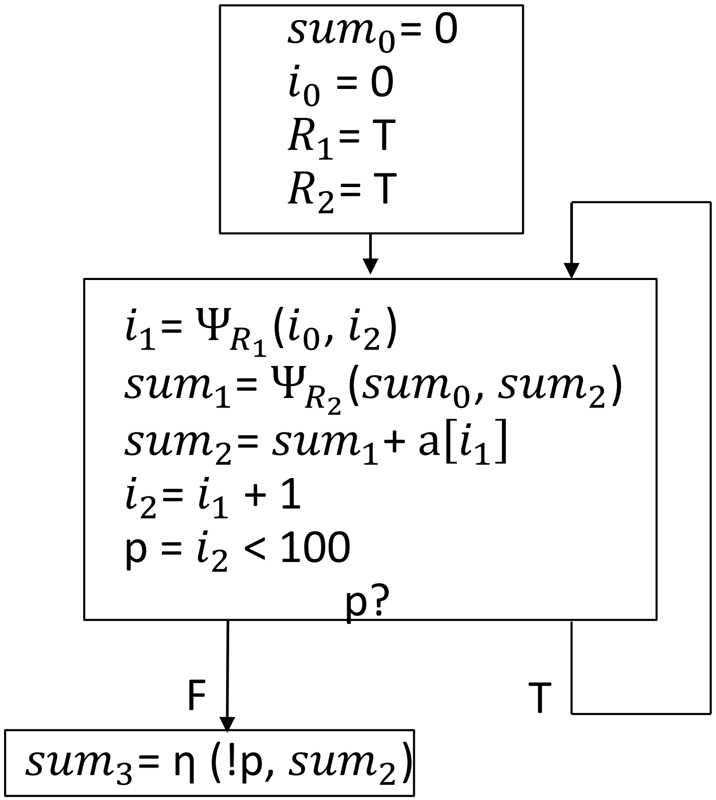


$sum_2 = sum_1 + a[i_1]$
$i_0 = 0$
$R_1 = T$
$i_1 = \Psi_{R_1}(i_0, i_2)$
Entry point → $sum_3 = \eta\ (!p, sum_2)$
$p = i_2 < 100$
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
$R_2 = T$
$sum_0 = 0$

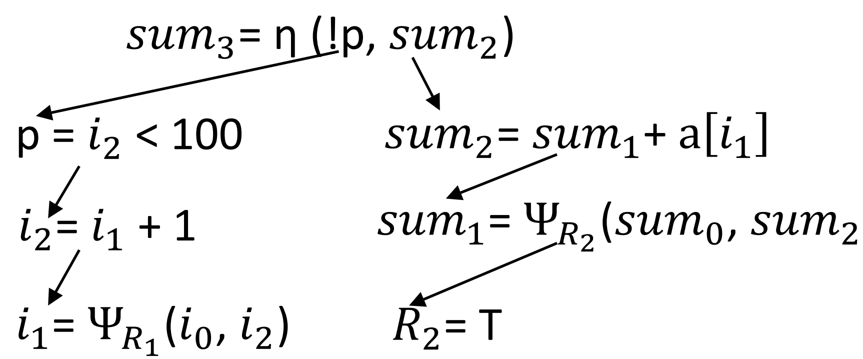


Data-driven sequencing

Drop branches and (just for fun) shuffle instructions

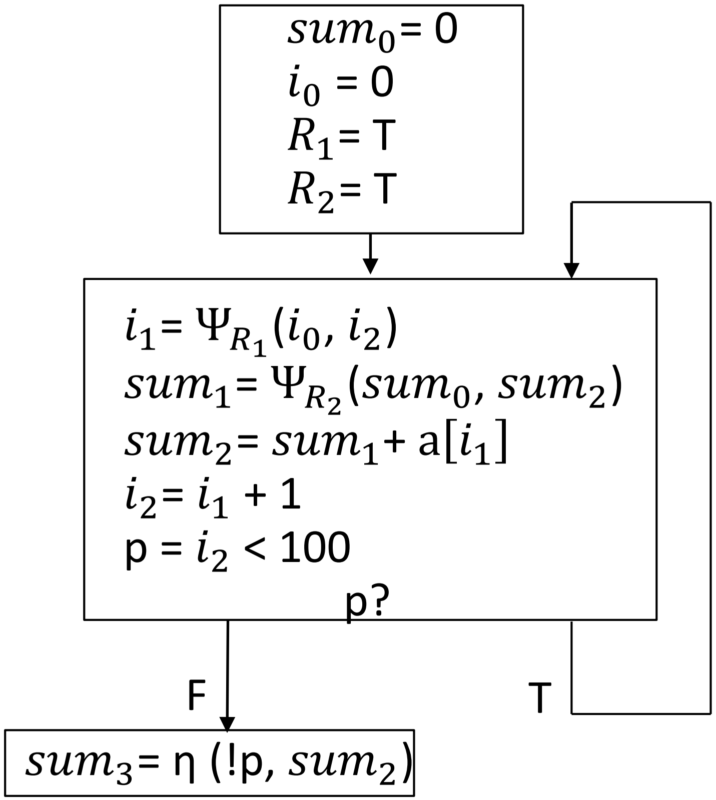


$sum_2 = sum_1 + a[i_1]$
i_0 = 0
R_1 = T
$i_1 = \Psi_{R_1}(i_0, i_2)$
Entry point → sum_3 = η (!p, sum_2)
p = i_2 < 100
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
R_2 = T
sum_0 = 0



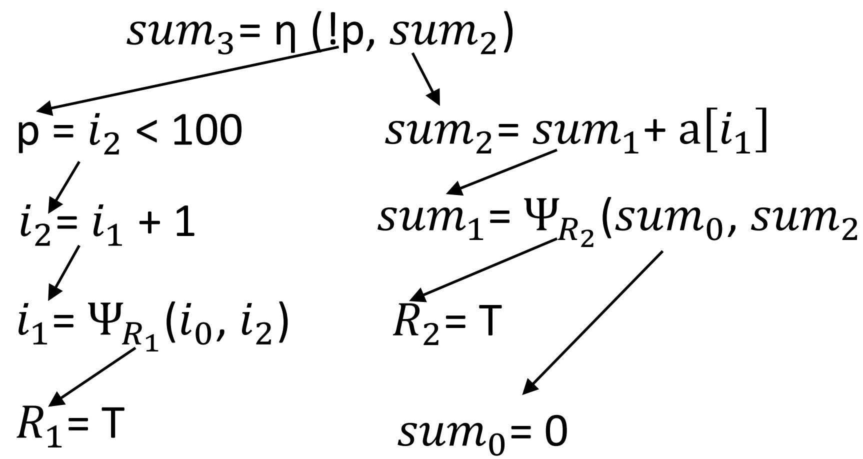
Data-driven sequencing

Drop branches and (just for fun) shuffle instructions



Entry point →

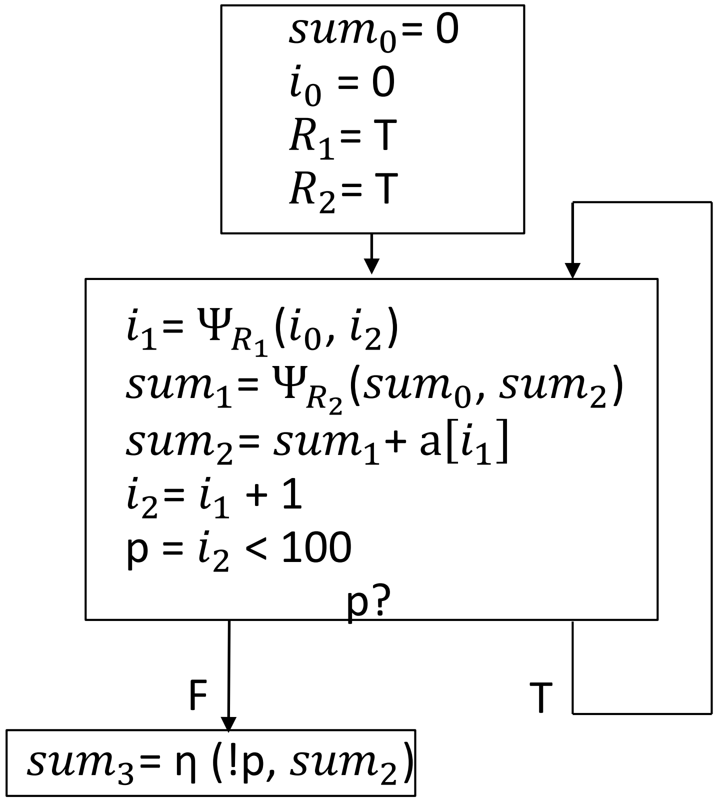
$sum_2 = sum_1 + a[i_1]$
$i_0 = 0$
$R_1 = T$
$i_1 = \Psi_{R_1}(i_0, i_2)$
$sum_3 = \eta\ (!p, sum_2)$
$p = i_2 < 100$
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
$R_2 = T$
$sum_0 = 0$



Data-driven sequencing

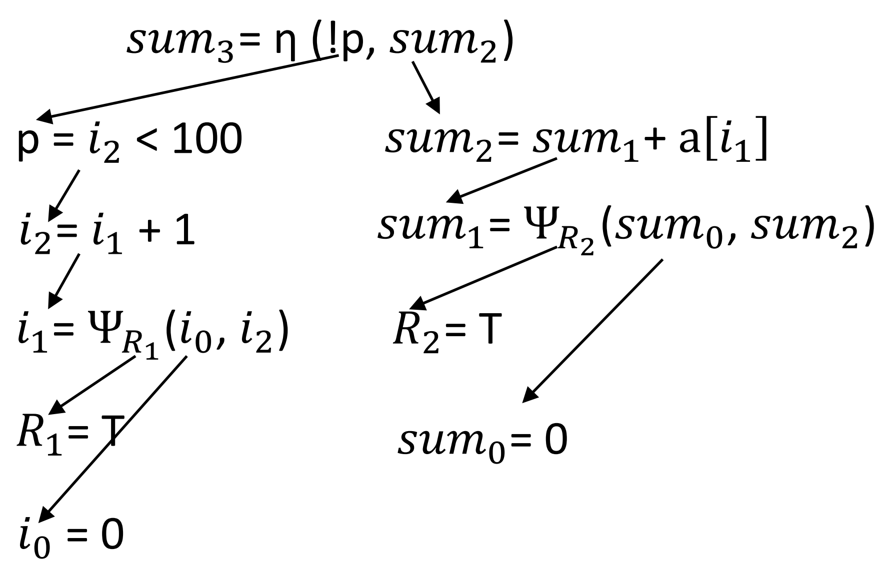
MichiganTech

Drop branches and (just for fun) shuffle instructions



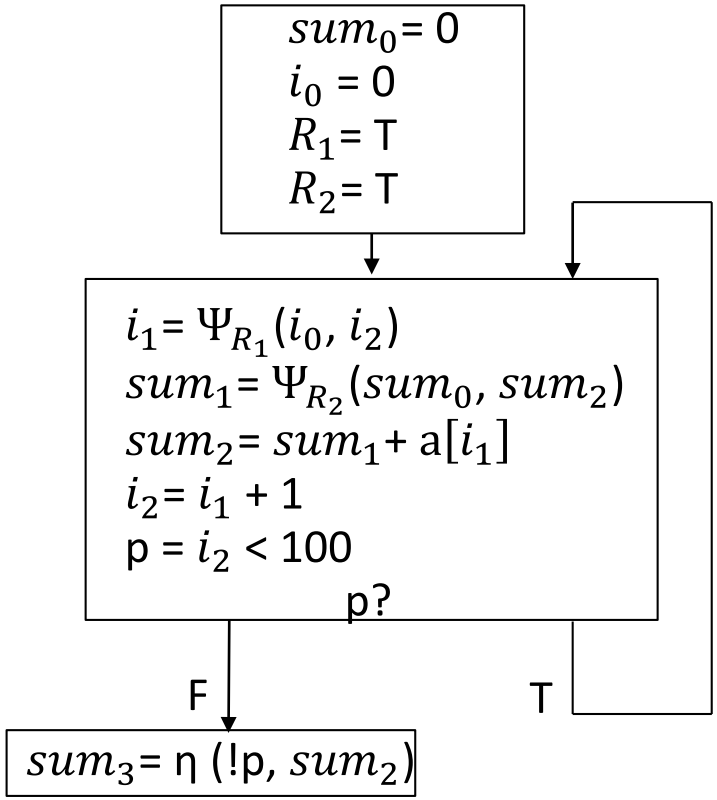
Entry point →

$$
\begin{aligned}
&sum_2 = sum_1 + a[i_1] \\
&i_0 = 0 \\
&R_1 = T \\
&i_1 = \Psi_{R_1}(i_0, i_2) \\
&sum_3 = \eta\ (!p, sum_2) \\
&p = i_2 < 100 \\
&i_2 = i_1 + 1 \\
&sum_1 = \Psi_{R_2}(sum_0, sum_2) \\
&R_2 = T \\
&sum_0 = 0
\end{aligned}
$$



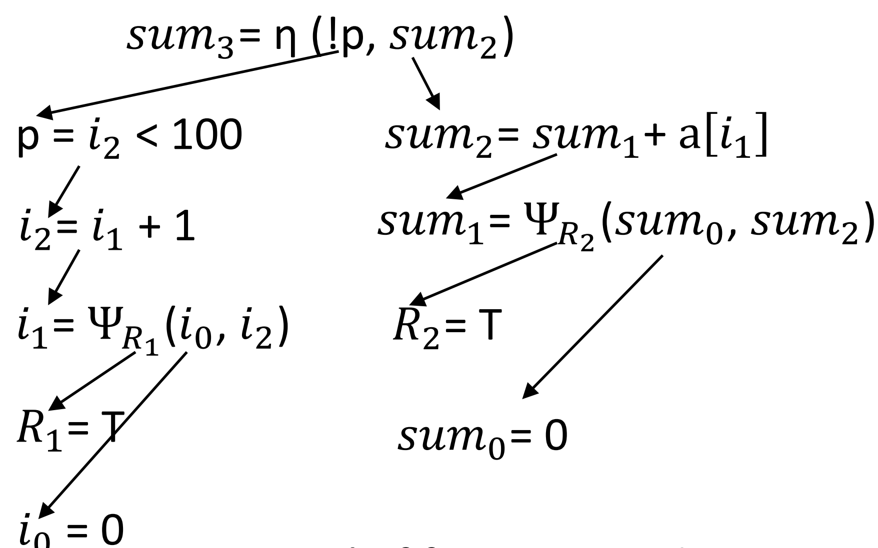
Data-driven sequencing

Drop branches and (just for fun) shuffle instructions



Data-driven sequencing

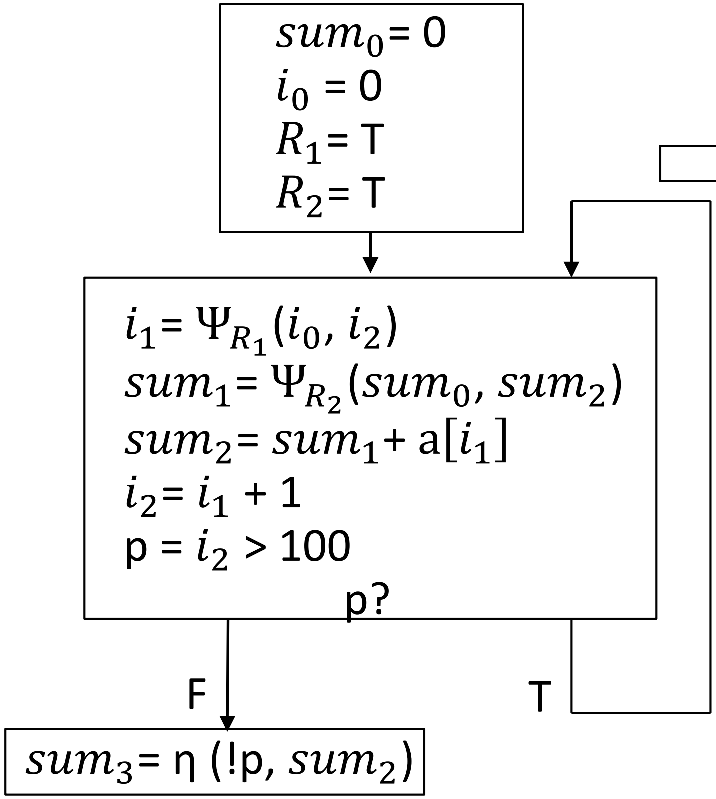
$sum_2 = sum_1 + \mathrm{a}[i_1]$
$i_0 = 0$
$R_1 = \mathrm{T}$
$i_1 = \Psi_{R_1}(i_0, i_2)$
Entry point → $sum_3 = \eta\ (!\mathrm{p}, sum_2)$
$\mathrm{p} = i_2 < 100$
$i_2 = i_1 + 1$
$sum_1 = \Psi_{R_2}(sum_0, sum_2)$
$R_2 = \mathrm{T}$
$sum_0 = 0$



End of first iteration!

MichiganTech

Conversion to a “functional program”



loop (i_1, sum_1)

{

$sum_2 = sum_1 + a[i_1]$

$i_2 = i_1 + 1$

$p = i_2 > 100$

return $\Psi_p\ (sum_2, loop(i_2, sum_2))$

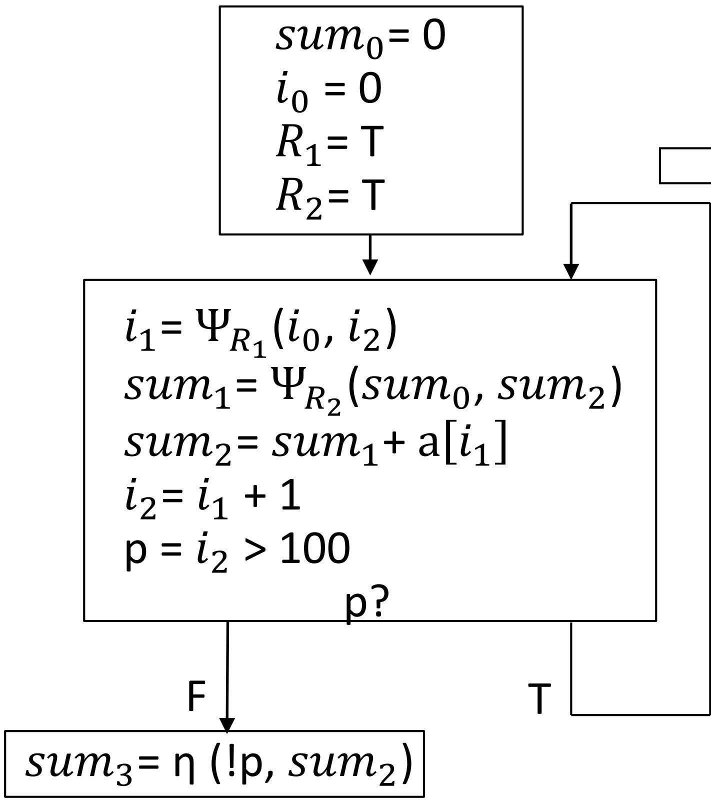
}

$sum_0 = 0$

$i_0 = 0$

$sum_3 =$ loop (i_0, sum_0)

Conversion to a “functional program”

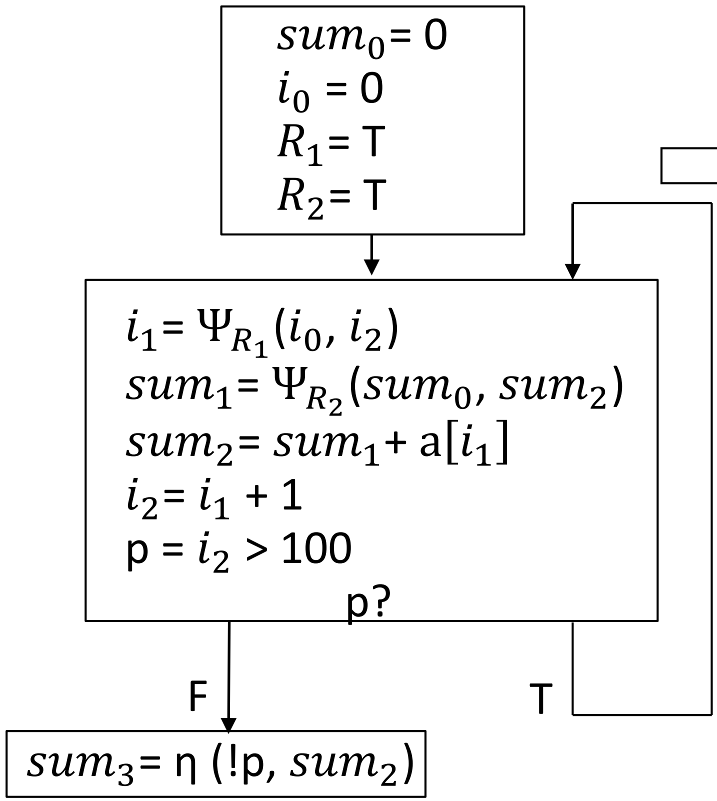


loop (i_1, sum_1)
{
  $sum_2 = sum_1 + a[i_1]$
  $i_2 = i_1 + 1$
  $p = i_2 > 100$
  return Ψ_p (sum_2,loop(i_2, sum_2))
}

$sum_0 = 0$
$i_0 = 0$
sum_3 = loop (i_0, sum_0)

Isn’t this tail-recursion?

Conversion to a “functional program”



$\text{loop}\ (i_1, sum_1)$
{
$sum_2 = sum_1 + a[i_1]$
$i_2 = i_1 + 1$
$p = i_2 > 100$
$\text{return}\ \Psi_p\ (sum_2, \text{loop}(i_2, sum_2))$
}

$sum_0 = 0$
$i_0 = 0$
$sum_3 = \text{loop}\ (i_0, sum_0)$

Isn’t this tail-recursion?

The program on the right is a functional program, generated from the imperative program, following a completely mechanical procedure. In this program, sequencing is data-driven, selection is provided by gating functions and iteration is implemented using a special form of tail-recursion, we call “cut-tail” recursion.

Cut-tail is a mirror image of *continuation-passing*, only in reverse (we forward return points).

MichiganTech

“Branch” Prediction

Gating

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\ (k_{0,} k_1)$
$= k_2$

Under Lazy evaluation, there is a pipeline delay between the time the gating function gets the predicate and the appropriate path is fetched. The same is true for the recursive iterator.

The only way to remedy this delay is to *predict* the predicate. In this case, the function can simultaneously evaluate the predicted path and the predicate expression, provided that evaluation of the predicted path is side-effect free.

“Branch” Prediction

Gating

k_0 = 5
k_1 = 10
P = a < b
$k_2 = \Psi_p\,(k_{0,}k_1)$
$= k_2$

Under Lazy evaluation, there is a pipeline delay between the time the gating function gets the predicate and the appropriate path is fetched. The same is true for the recursive iterator.

The only way to remedy this delay is to *predict* the predicate. In this case, the function can simultaneously evaluate the predicted path and the predicate expression, provided that evaluation of the predicted path is side-effect free.

Corollary:

1. It is more appropriate to talk about control-dependence prediction, rather than “branch prediction”.

2. Correlation manifests itself now among predicates. It is more appropriate to talk about correlation in control-dependence as well.

3. **Control-dependence prediction, and exploiting control-dependence correlation are lasting contributions**. They won’t go away whether or not programs are expressed imperatively or functionally (or by using branches or not).

What we have

What we have

We have a “graph solver” which can take any imperative program and generate a functional version of it in the form of a program representation, called Future Gated Single Assignment (FGSA) form:

Shuhan Ding, John Earnest, and Soner Önder. 2014. *Single Assignment Compiler, Single Assignment Architecture: Future Gated Single Assignment Form*; *Static Single Assignment with Congruence Classes.* In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14). ACM, New York, NY, USA, , Pages 196 , 12 pages. DOI=10.1145/2544137.2544158

What we have

We have a “graph solver” which can take any imperative program and generate a functional version of it in the form of a program representation, called Future Gated Single Assignment (FGSA) form:

Shuhan Ding, John Earnest, and Soner Önder. 2014. *Single Assignment Compiler, Single Assignment Architecture: Future Gated Single Assignment Form*; *Static Single Assignment with Congruence Classes.* In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14). ACM, New York, NY, USA, , Pages 196 , 12 pages. DOI=10.1145/2544137.2544158

We have a complete instruction set in which FGSA programs can be encoded at the machine-level:

Omkar Javeri and Zhaoxiang Jin, and Soner Onder (2018). *A Demand-Driven Instruction Set Architecture.* Technical Report, Department of Computer Science, Michigan Technological University, CS-TR-18-01.

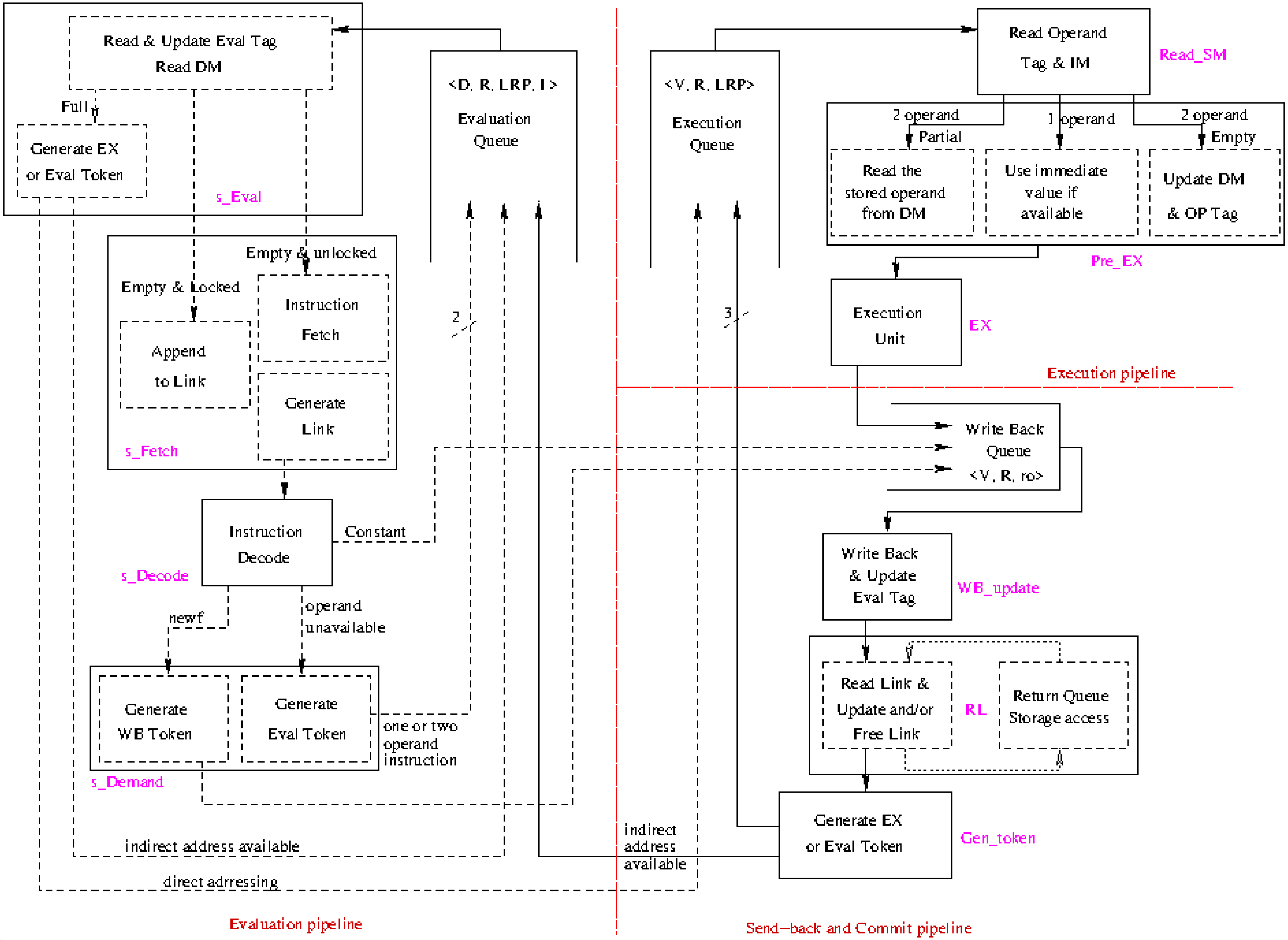
What we have

What we have

We have a functioning pipelined processor implementation written in ADL language which gives us a cycle-accurate simulator:

Omkar Javeri and Tino Moore, and Soner Onder (2018). *Demand-Driven Execution Pipeline.* Technical Report, Department of Computer Science, Michigan Technological University, CS-TR-19-00.

Demand-driven Execution Pipeline



MichiganTech

What we have

What we have

Finally, we have a compiler that can compile C programs and generate demand-driven ISA code for inner-most loops.

Open Problems

Open Problems

1. Existing branch predictors rely on existence of *sequential sequencing* (i.e., global branch history). We do not know how to exploit "predicate histories" which are data-driven.

2. It seems that in programs with sufficient ILP, prediction of "forward" branches in this domain may not be necessary, or a non-correlating simple predictor would suffice. On the other hand, we MUST predict the loop back-edges. We do not know how to best do this.

Diversity

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

MichiganTech

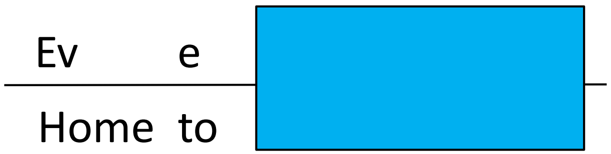
Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :



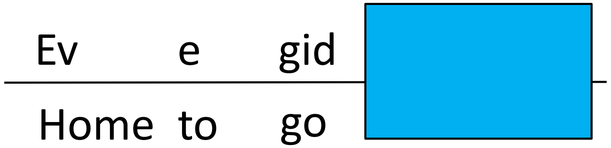
Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :



Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

Ev	e	gid	eceğ
Home	to	go	will

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

Ev	e	gid	eceğ	im
Home	to	go	will	I

MichiganTech

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

Ev	e	gid	eceğ	im
Home	to	go	will	I

Diversity

"Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer."

— Edsger W. Dijkstra

A simple Turkish sentence: Eve gideceğim.

Dissecting :

Ev	e	gid	eceğ	im
Home	to	go	will	I

Turkish is primarily a suffix based, "postfix language". In Turkish, it is natural to say everything in "reverse"!

MichiganTech

Acknowledgement

Graduate and Graduated Students

Dr. Shuhan Ding
(MTU-CS)
FGSA
(Qualcomm)

Tino Moore
(MTU-CS)
Graph-solver

Omkar Javeri
(MTU-ECE)
Processor design

Ryan Baird
(FSU-CS)
Compiler Framework

XPS: Full: FP: Collaborative Research: Sphinx: Combining Data and Instruction Level Parallelism through Demand Driven Execution of Imperative Programs, CCF 1533828, MTU Lead institution, PI:Soner Onder (Michigan Tech), PI:David Whalley (Florida State University)

EAGER: Combining Data and Instruction Level Parallelism through Demand Driven Execution of Imperative Programs, CCF 1450062, PI:Soner Onder

SHF: Small: Single Assignment Architecture / Single Assignment Compiler, CCF 1116551, PI: Soner Onder

MichiganTech