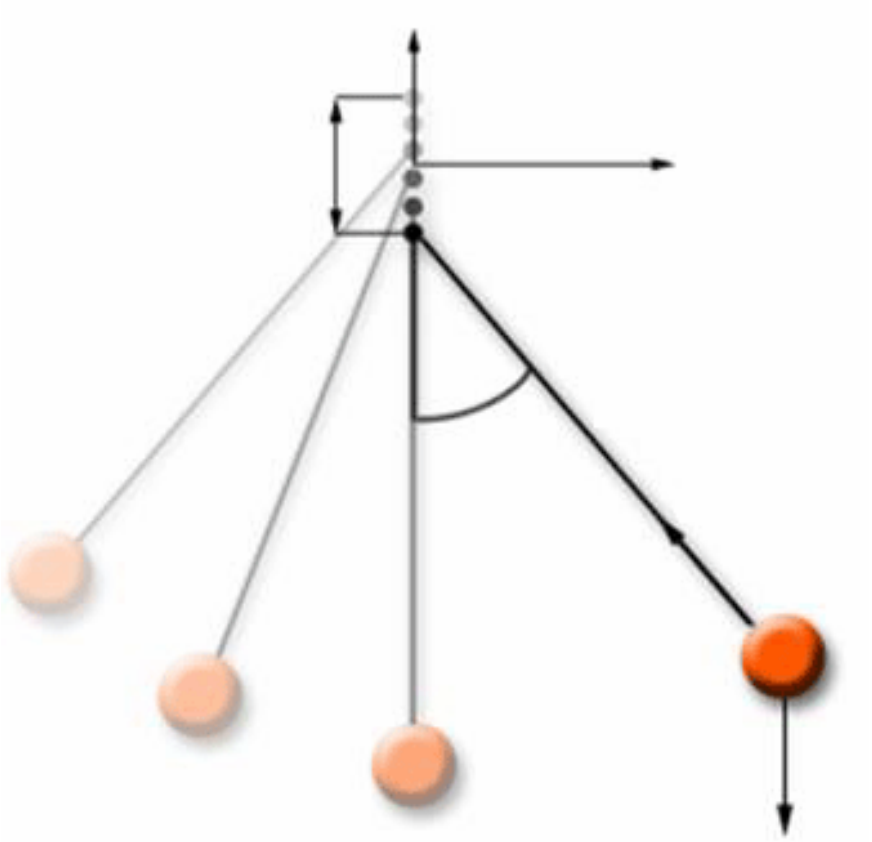
Towards Designing a Reversible Processor Architecture

Michael Kirkedal Thomsen
Robert Glück
Holger Bock Axelsen

RC2009

York, March 2009



Conventional Logic

- Computation today depends on logic operations that destroy information

Example: AND gate

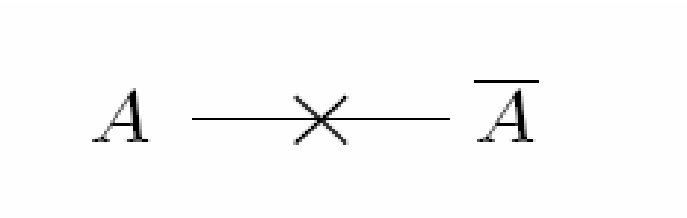
In		Out
A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

Reversible Logic Gates

Solution: Avoid information loss

1. The number of input lines is equal to the number of output lines (written $n \times n$)
2. Its Boolean function $B^n \rightarrow B^n$ is bijective

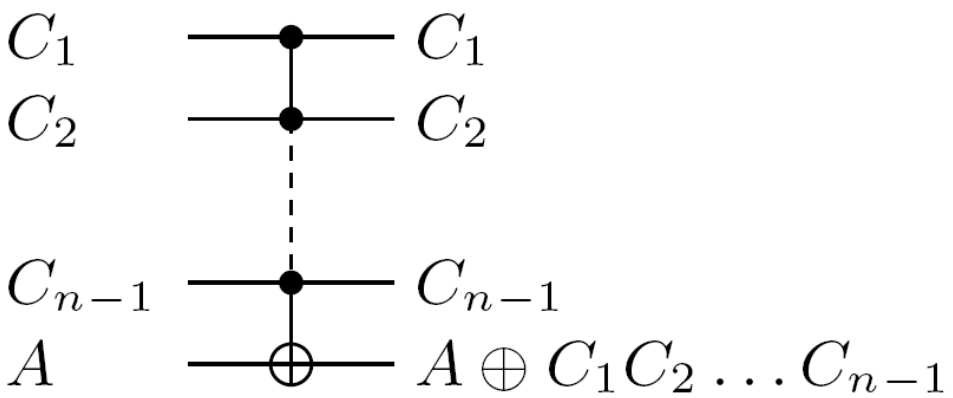
Not Gate



In	Out
A	¬A
0	1
1	0

- Simplest reversible gate
- Only classical logic gate used in reversible circuits

n-bit Controlled-Not Gate

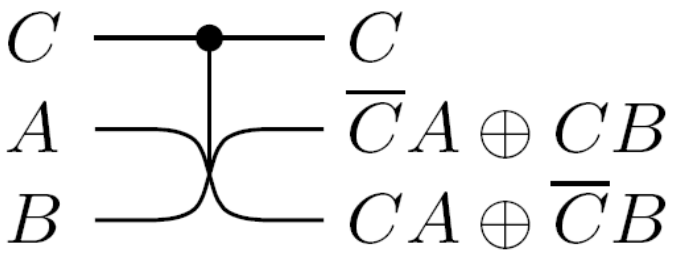


[Toffoli '80]

In			Out		
C_1	C_2	A	C_1	C_2	R_A
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

- 3-bit: Toffoli gate
  - Universal gate
- 2-bit: Feynman gate
- Self-inverse

3-bit Controlled-Swap (Fredkin) Gate



[Fredkin, Toffoli '82]

In			Out		
C	A	B	C	R_A	R_B
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	1

- Universal gate
- Self-inverse

- Can be generalized to a n-bit controlled-swap gate

Reversible Circuits

Important restriction:

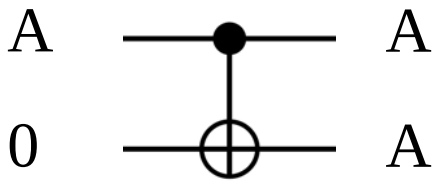
- Fan-out is not permitted – it is an irreversible construction

Reversible Circuits

Important restriction:

- Fan-out is not permitted – it is an irreversible construction

Simulate fan-out using one Feynman-gate



Reversible Circuits

Important restriction:

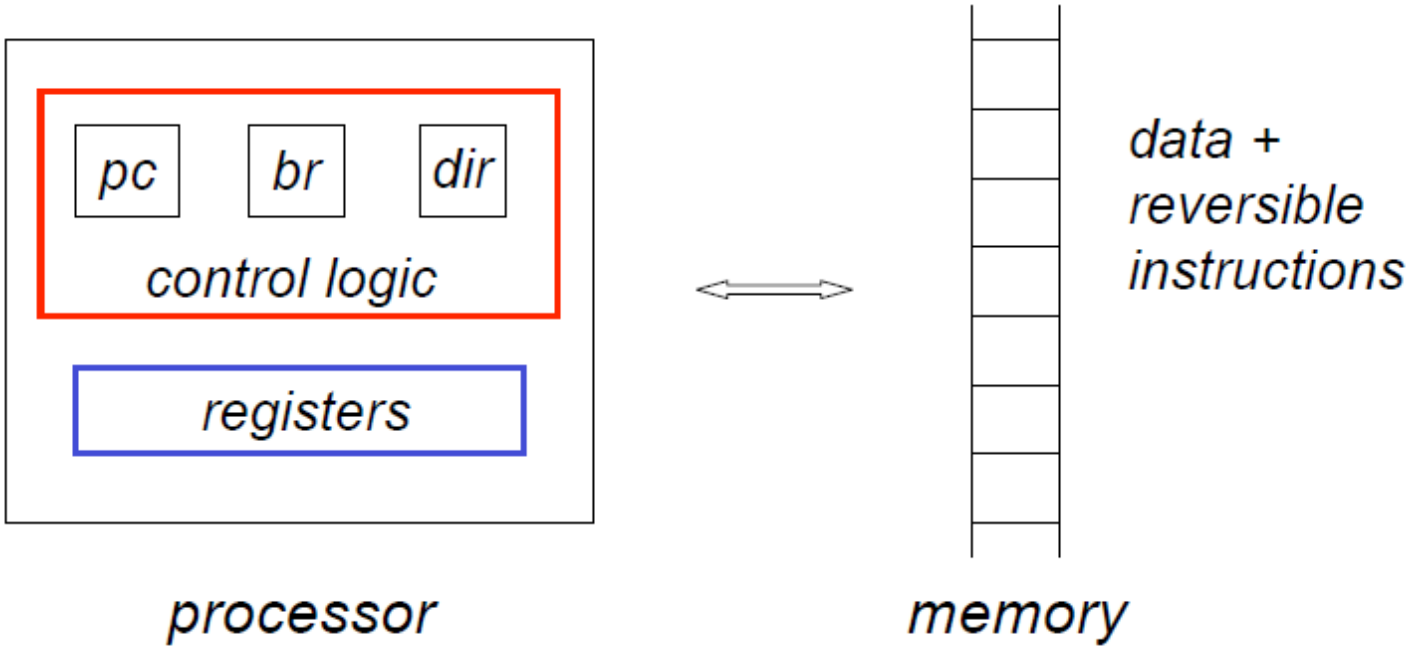
- Fan-out is not permitted – it is an irreversible construction

- Garbage bit
  - Non-constant output that is not part of the desired result
  - Problem in reverse execution
- Ancilla bit
  - Bit that is constant at both input and output

Ancillae is preferable to garbage due to reuse

Reversible von Neumann Architecture

- *Program counter* - points to current instruction
- *Branch register* - contains the offset of a jump
- *Direction bit* - specifies the execution direction



***Design*: [Frank '99]**
***Formalized*: [Axelsen, Glück, Yokoyama '07]**

Design Criteria and Goals for ISA

Requirements

- Forward and backward determinism
- Locally invertible **[Yokoyama, Glück '07]**
- Use reversible updates **[Axelsen, Glück, Yokoyama '07]**
- r-Turing complete **[Morita *et al.* '01]**
  - Turing completeness for reversible computing

Goals

- RISC design strategy
- Simple circuit implementation
- No history

Injective in 1st Argument

If an operator $\odot$ is **injective in its 1st argument**,

$a \odot \mathbf{c} = b \odot \mathbf{c} \Rightarrow a = b$ (if $a \odot \mathbf{c}$ and $b \odot \mathbf{c}$ defined),

then an operator $\oslash$ exists to **reconstruct 1st argument**:

$(a \odot \mathbf{c}) \oslash \mathbf{c} = a$

Example:

$n + 3 = m + 3 \Rightarrow n = m$ and $(n + 3) - 3 = n$

Non-Example:

$n * 0 = m * 0 \not\Rightarrow n = m$

Reversible Update

Given ($\odot$, $\oslash$) and a partial function **f**, **then** function

$g(x, y) =^{def} (x \odot \mathbf{f(y)}, y)$

is a **reversible update of x**, and there **exists**

$g^{-1}(x, y) =^{def} (x \oslash \mathbf{f(y)}, y)$

Example:

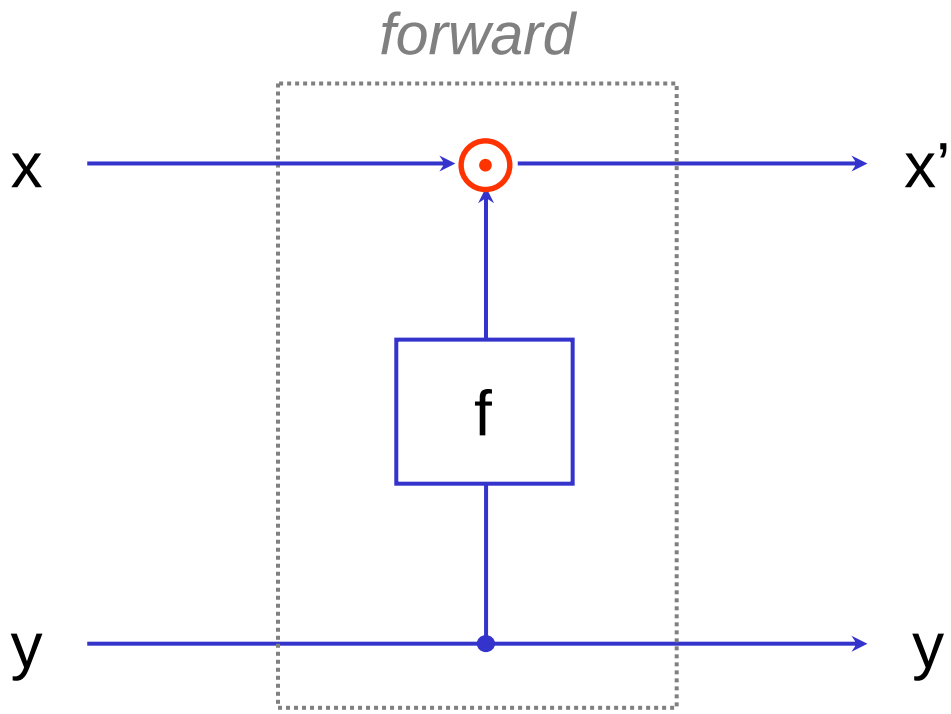
$g\ (x, y) = (x + \mathbf{f(y)}, y)$ … reversible update of x

$g^{-1}(x, y) = (x - \mathbf{f(y)}, y)$ … inverse function

Fct **f** can be non-injective; x,y can be tuples $(x_1,\ldots,x_n), (y_1,\ldots,y_m)$.

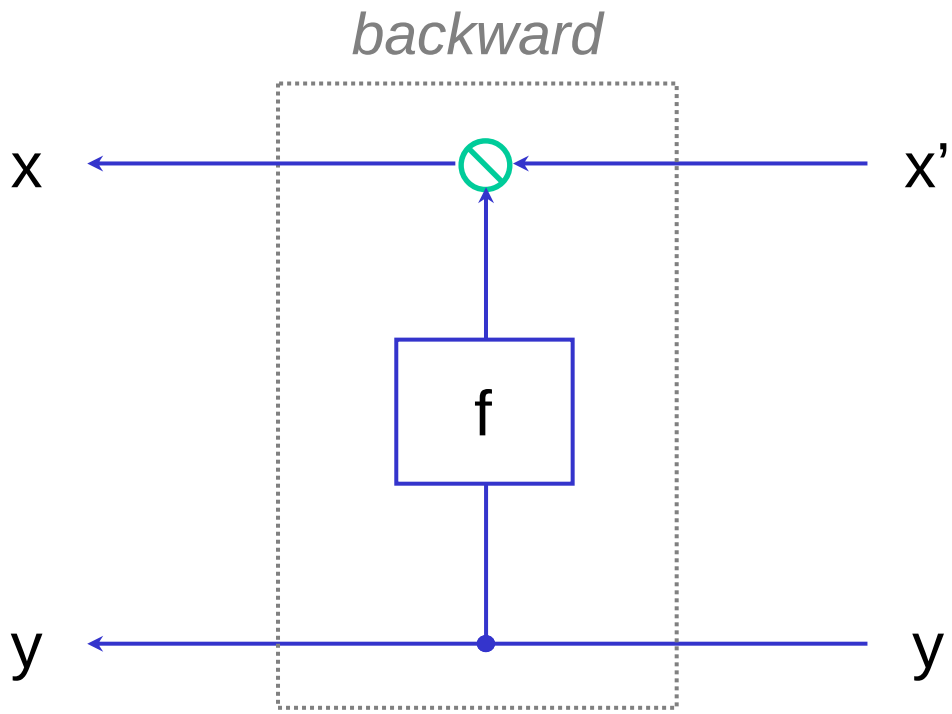
Every reversible update g is an injective function.

Reversible Update of x



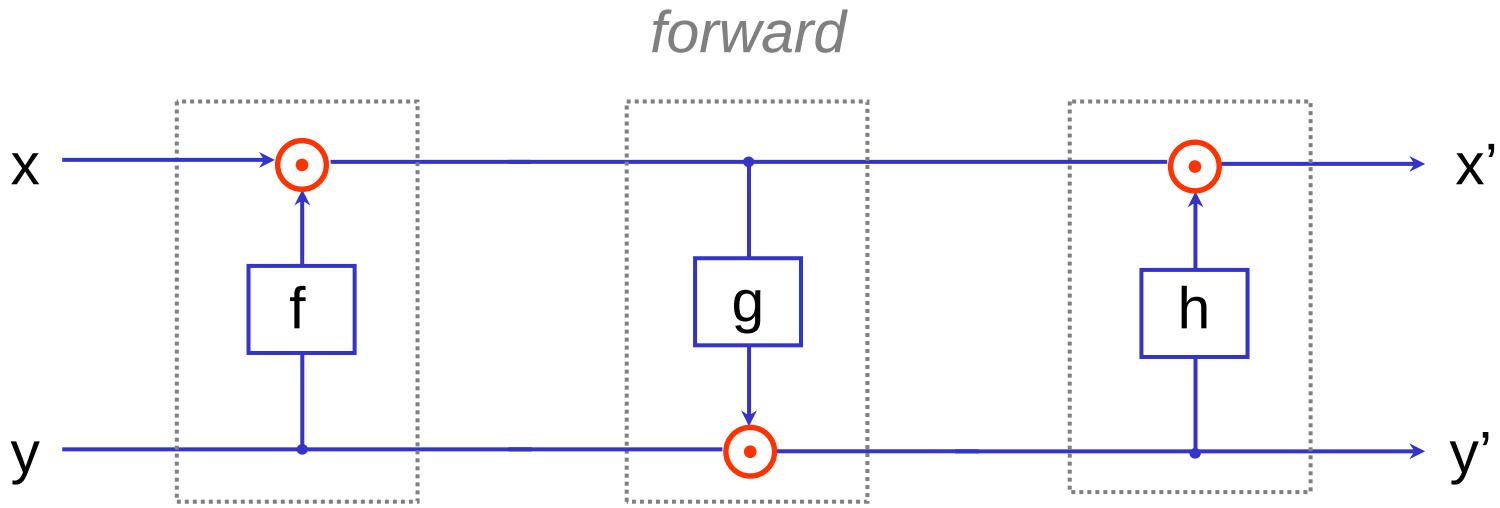
⊙ … injective in x-argument

Reversible Update of x



⊘ … injective in x-argument

Composition of Reversible Updates



sequence of reversible updates
in any combination is reversible

Reversible Updates

2 + 3 → 5

1 + 4 → 5

? + ? ← 5

- Addition is irreversible
- We need a reversible way to add numbers

- NEW SLIDES FROM RG

Reversible Binary Addition

2 + 3 → 5

1 + 4 → 5

? + ? ← 5

- Addition is irreversible
- Solution:

$$+_n(A, B) \mapsto (A, B + A \bmod 2^n)$$

- Reversible updates
- Keeps one operant

Arithmetic and Memory Instructions

i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$

Arithmetic and Memory Instructions

i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$
ADD1 reg_d	SUB1 reg_d	$reg_d \leftarrow reg_d +_n 1$
SUB1 reg_d	ADD1 reg_d	$reg_d \leftarrow reg_d -_n 1$

Arithmetic and Memory Instructions

i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$
ADD1 reg_d	SUB1 reg_d	$reg_d \leftarrow reg_d +_n 1$
SUB1 reg_d	ADD1 reg_d	$reg_d \leftarrow reg_d -_n 1$
NEG reg_d	NEG reg_d	$reg_d \leftarrow 0 -_n reg_d$

Arithmetic and Memory Instructions

i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$
ADD1 reg_d	SUB1 reg_d	$reg_d \leftarrow reg_d +_n 1$
SUB1 reg_d	ADD1 reg_d	$reg_d \leftarrow reg_d -_n 1$
NEG reg_d	NEG reg_d	$reg_d \leftarrow 0 -_n reg_d$
MUL2 reg_d	DIV2 reg_d	$reg_d \leftarrow mul2(reg_d)$
DIV2 reg_d	MUL2 reg_d	$reg_d \leftarrow div2(reg_d)$

Arithmetic and Memory Instructions

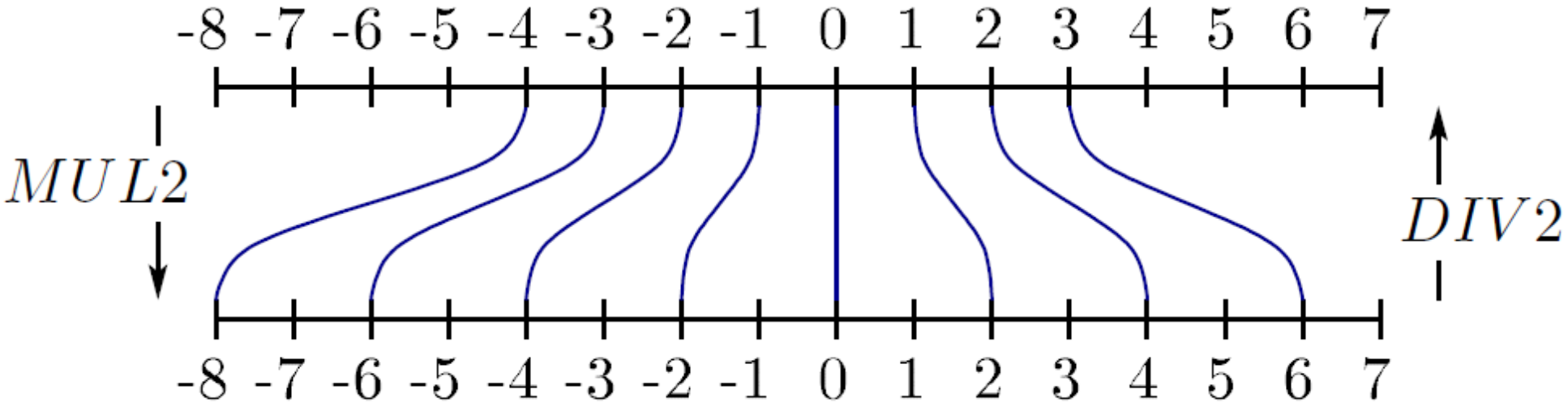
i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$
ADD1 reg_d	SUB1 reg_d	$reg_d \leftarrow reg_d +_n 1$
SUB1 reg_d	ADD1 reg_d	$reg_d \leftarrow reg_d -_n 1$
NEG reg_d	NEG reg_d	$reg_d \leftarrow 0 -_n reg_d$
MUL2 reg_d	DIV2 reg_d	$reg_d \leftarrow mul2(reg_d)$
DIV2 reg_d	MUL2 reg_d	$reg_d \leftarrow div2(reg_d)$
XOR reg_d reg_s	XOR reg_d reg_s	$reg_d \leftarrow reg_d \oplus reg_s$
XORI reg_d *imm*	XORI reg_d *imm*	$reg_d \leftarrow reg_d \oplus imm$

Arithmetic and Memory Instructions

i	*inv(i)*	*effect(i)*
ADD reg_d reg_s	SUB reg_d reg_s	$reg_d \leftarrow reg_d +_n reg_s$
SUB reg_d reg_s	ADD reg_d reg_s	$reg_d \leftarrow reg_d -_n reg_s$
ADD1 reg_d	SUB1 reg_d	$reg_d \leftarrow reg_d +_n 1$
SUB1 reg_d	ADD1 reg_d	$reg_d \leftarrow reg_d -_n 1$
NEG reg_d	NEG reg_d	$reg_d \leftarrow 0 -_n reg_d$
MUL2 reg_d	DIV2 reg_d	$reg_d \leftarrow mul2(reg_d)$
DIV2 reg_d	MUL2 reg_d	$reg_d \leftarrow div2(reg_d)$
XOR reg_d reg_s	XOR reg_d reg_s	$reg_d \leftarrow reg_d \oplus reg_s$
XORI reg_d *imm*	XORI reg_d *imm*	$reg_d \leftarrow reg_d \oplus imm$
EXCH reg_d reg_a	EXCH reg_d reg_a	$reg_d \leftrightarrow M(reg_a)$

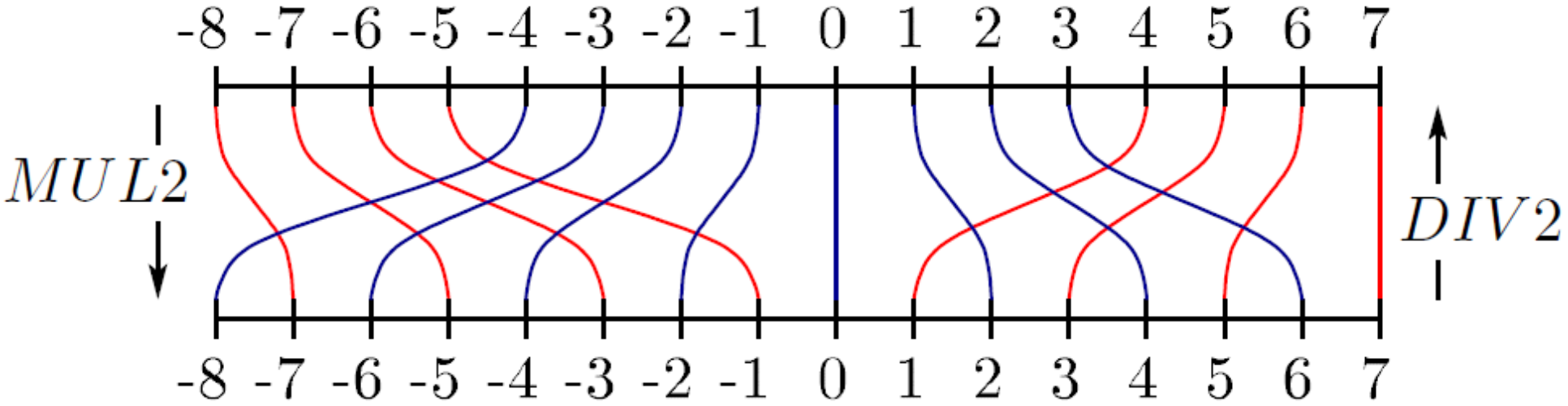
Division/Multiplication by 2

- Shift left/right and truncate is *not* reversible
- Well-behaved for numbers that are “in scope”



Division/Multiplication by 2

- Shift left/right is *not* reversible
- Well-behaved for numbers that are “in scope”
- “Out of scope” must ensure reversibility and local invertibility



Branch Instructions

i	*condition(i)*	*effect(i)*
BGEZ reg_d *off*	$reg_d >= 0$	$br \leftarrow br +_n off \cdot dir$
BLZ reg_d *off*	$reg_d < 0$	$br \leftarrow br +_n off \cdot dir$

- Updates only BR and DIR
- Relative offset

Branch Instructions

i	*condition(i)*	*effect(i)*
BGEZ reg_d *off*	$reg_d >= 0$	$br \leftarrow br +_n off \cdot dir$
BLZ reg_d *off*	$reg_d < 0$	$br \leftarrow br +_n off \cdot dir$
BEVN reg_d *off*	$even(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BODD reg_d *off*	$odd(reg_d)$	$br \leftarrow br +_n off \cdot dir$

- Updates only BR and DIR
- Relative offset

Branch Instructions

i	*condition(i)*	*effect(i)*
BGEZ reg_d *off*	$reg_d >= 0$	$br \leftarrow br +_n off \cdot dir$
BLZ reg_d *off*	$reg_d < 0$	$br \leftarrow br +_n off \cdot dir$
BEVN reg_d *off*	$even(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BODD reg_d *off*	$odd(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BRA *off*		$br \leftarrow br +_n off \cdot dir$

- Updates only BR and DIR
- Relative offset

Branch Instructions

i	*condition(i)*	*effect(i)*
BGEZ reg_d *off*	$reg_d >= 0$	$br \leftarrow br +_n off \cdot dir$
BLZ reg_d *off*	$reg_d < 0$	$br \leftarrow br +_n off \cdot dir$
BEVN reg_d *off*	$even(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BODD reg_d *off*	$odd(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BRA *off*		$br \leftarrow br +_n off \cdot dir$
RBRA *off*		$br \leftarrow -(br +_n off \cdot dir)$; $dir \leftarrow -dir$

- Updates only BR and DIR
- Relative offset

Branch Instructions

i	*condition(i)*	*effect(i)*
BGEZ reg_d *off*	$reg_d >= 0$	$br \leftarrow br +_n off \cdot dir$
BLZ reg_d *off*	$reg_d < 0$	$br \leftarrow br +_n off \cdot dir$
BEVN reg_d *off*	$even(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BODD reg_d *off*	$odd(reg_d)$	$br \leftarrow br +_n off \cdot dir$
BRA *off*		$br \leftarrow br +_n off \cdot dir$
RBRA *off*		$br \leftarrow br +_n off \cdot dir$; $dir \leftarrow -dir$
SWBR reg_d		$br \leftrightarrow reg_d$

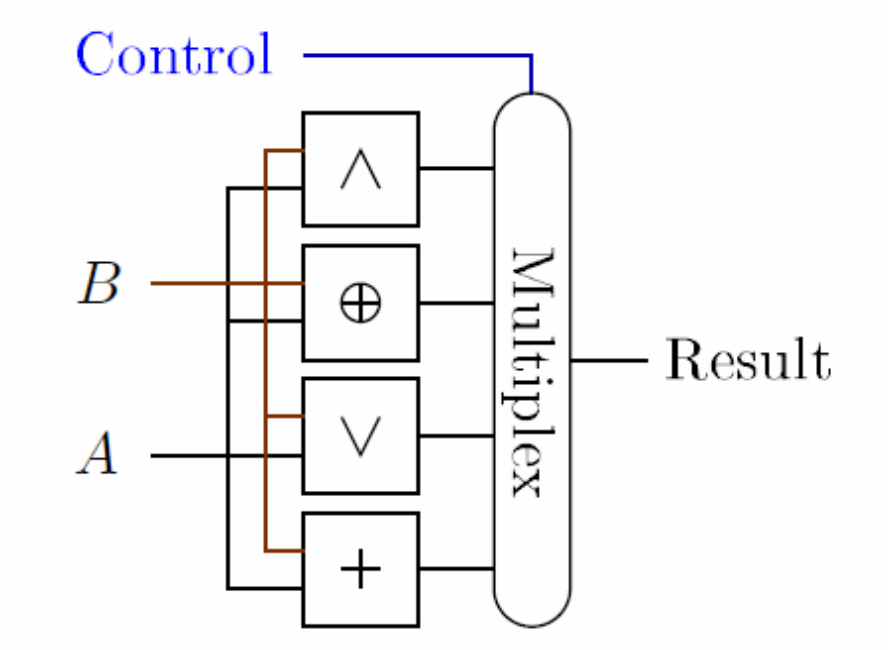
- Updates only BR and DIR
- Relative offset

Architecture and Instr Encoding

- 16 bit architecture
- 4 bit register names gives 16 registers
- Two's complement number representation
  - Range from -32768 through 32767
- Memory can index 2^{16} words of 16 bits - max: 128 KB
- Branch jumps can be up to 127 lines
- Immediate values from -128 through 127
- Garbage free and ancillae less design

bits:	15 12	11 8	7 4	3 0
Arithmetic & memory	*opcode*	reg_d	reg_s	*arith*
Branch & load immediate	*opcode*	reg_d	*off /imm*	

Conventional ALU

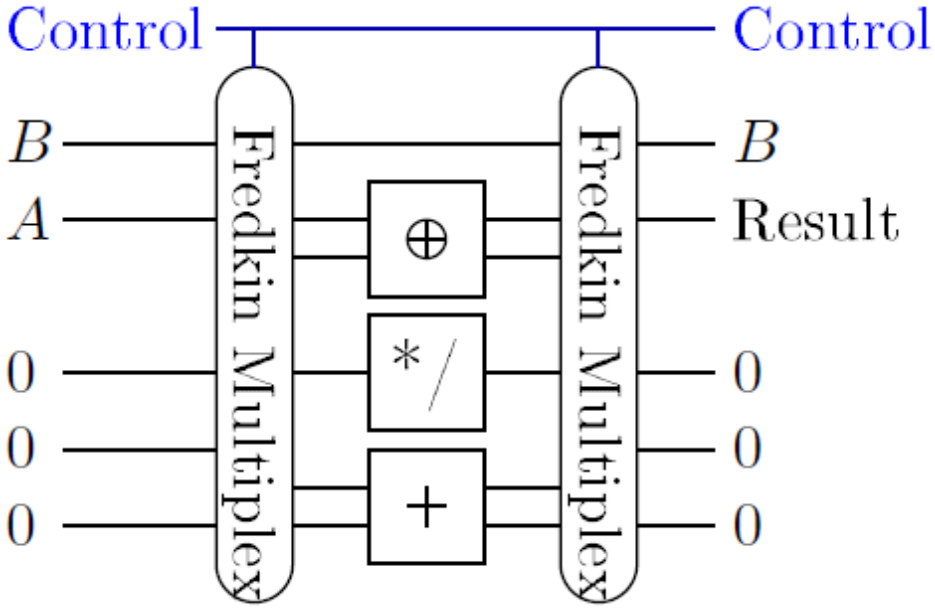


- Calculates all operations in parallel and chose only the desired result
- Not reversible design

Reversible ALU Designs

Parallel design

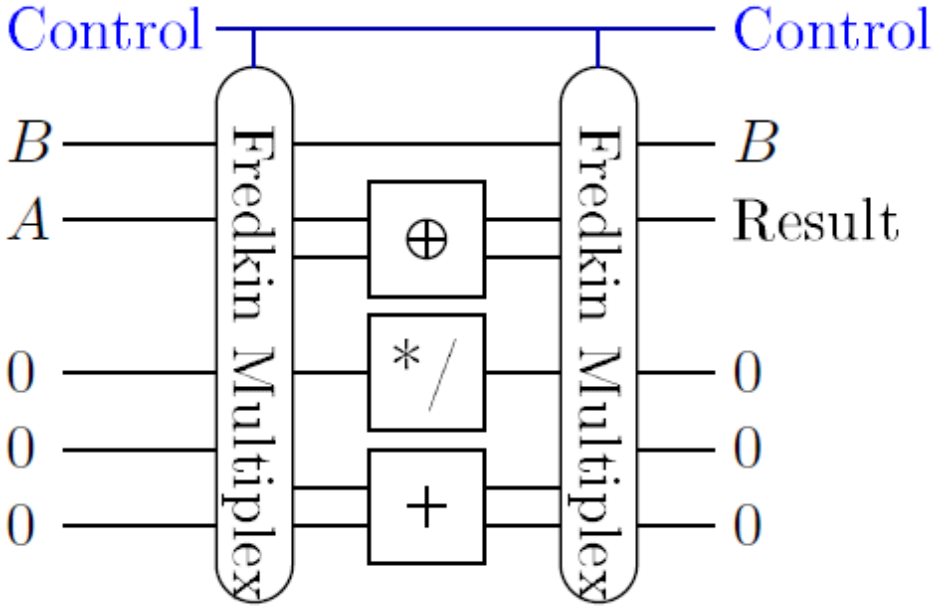
- Similar to conventional
- Directs *A* and *B* to the desired operation
- Many ancillae bits



Reversible ALU Designs

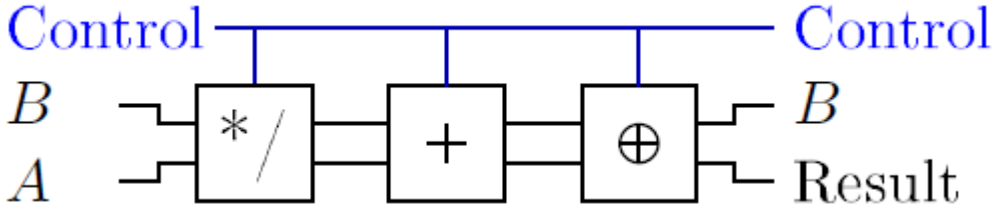
Parallel design

- Similar to conventional
- Directs *A* and *B* to the desired operation
- Many ancillae bits



Sequential design

- Ensures that only the desired operation changes *A and B*



Related Work

Reversible Computer Apparatus by Cezzar ’91 (Hampton, USA)

- Use Landauer’s embedding – puts garbage on a stack
- Not locally invertible design

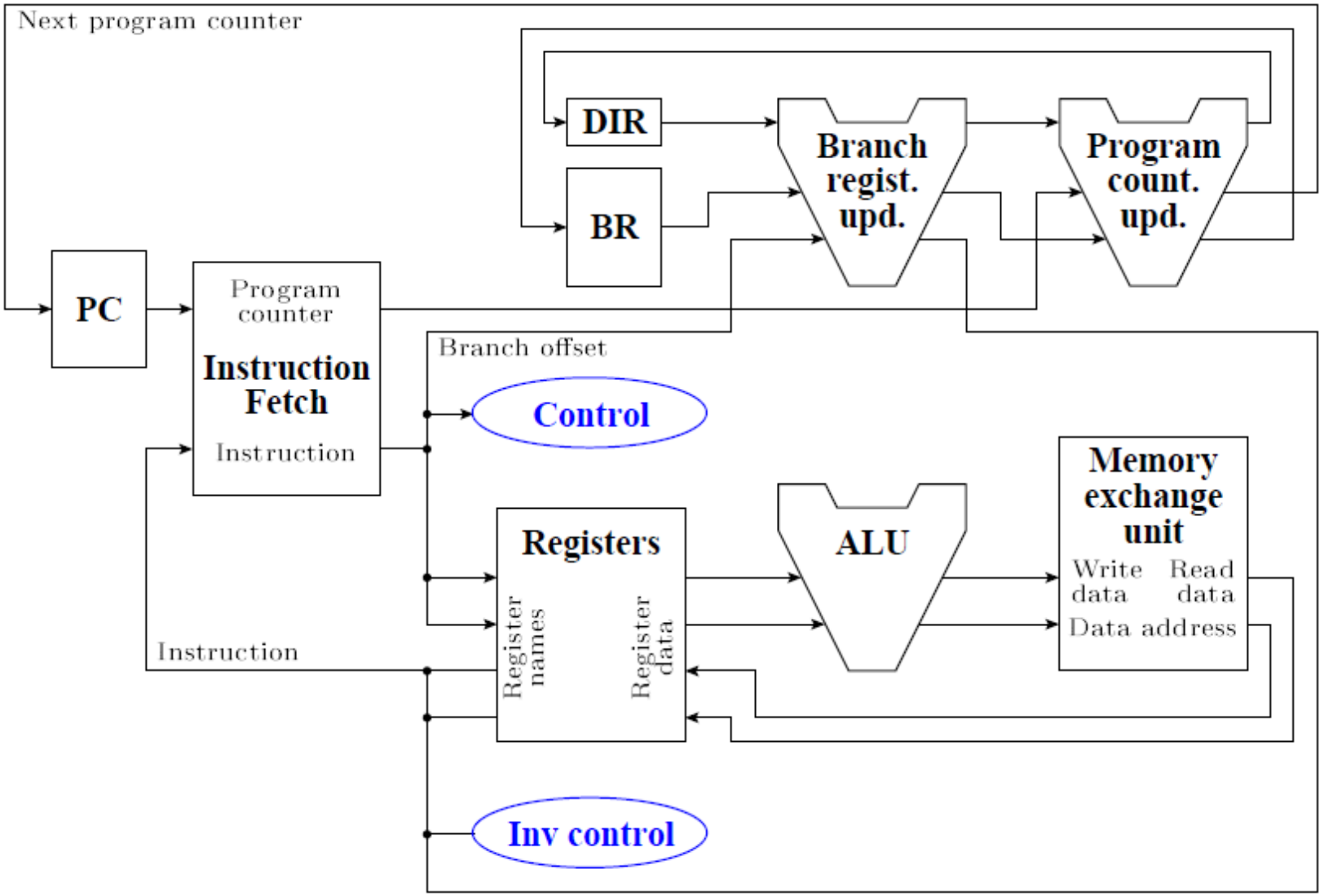
Pendulum and *PISA* by Vieri and Frank ’99 (MIT, USA)

- PISA is a reversible instructions set
- The Pendulum processor is not reversible
  - Implementation is SCRL that is not reversible logic

Reversible CPL by De Vos *et al.* ’03 (Gent, Belgium)

- Implementation of reversible gates in static CMOS
- Has been fabricated

Thank You!


Next program counter
DIR
BR
Branch regist. upd.
Program count. upd.
PC
Program counter
Instruction Fetch
Instruction
Branch offset
Control
Registers
Register names
Register data
ALU
Memory exchange unit
Write data
Read data
Data address
Instruction
Inv control