DataFlow Architectures / Languages (1975)

Kahn Process Networks (1974)

Communicating Sequential Processes (CSP) (1978)

Actor Model (1973/1978)

Message Passing Interface (MPI) (1992)

INFLUENTIAL MESSAGE PASSING COMPUTE MODELS AND FRAMEWORKS

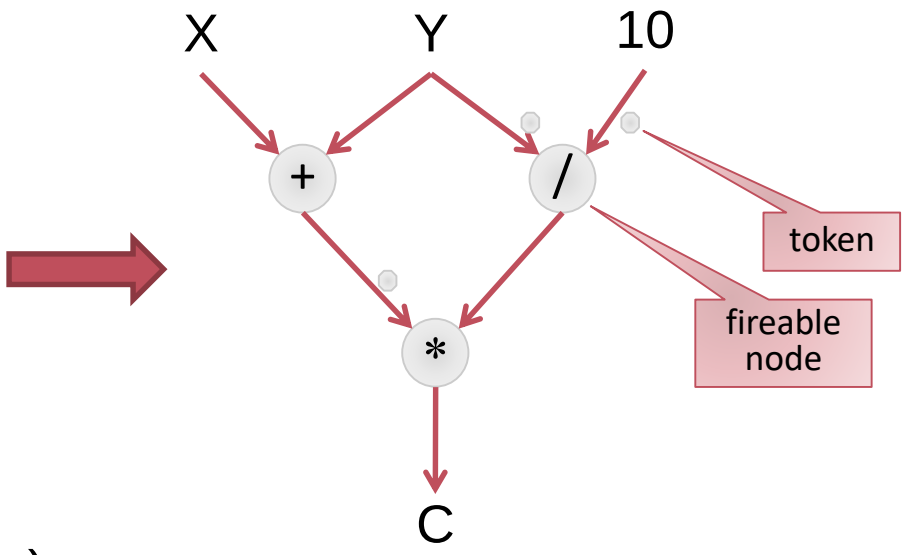
Dataflow Architectures and Languages*

- Take advantage of massive parallelism.
- Von Neumann Architecture unsuitable for parallelism. Bottlenecks:
  - Global program counter and
  - Global updatable memory
- Alternative proposal: dataflow architecture
  - Local memory
  - Execute instructions as soon as operands are available
  - Program in a dataflow computer is a directed graph and data flows between instructions along its edges

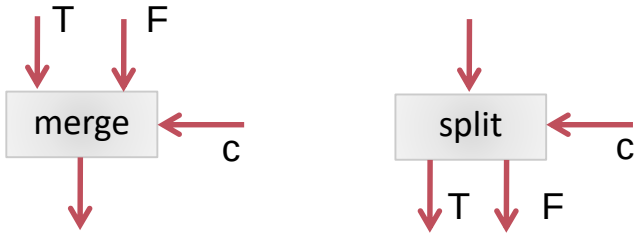
*following „Advances in Dataflow Programming Languages“, Johnston, Hanna, Millar, ACM Computing Surveys Vol36, No.1, 2004
Dataflow Programming Languages invented in the mid 1970s

Example

- A := X + Y
- B := Y / 10
- C := A * B



special control nodes (gates):

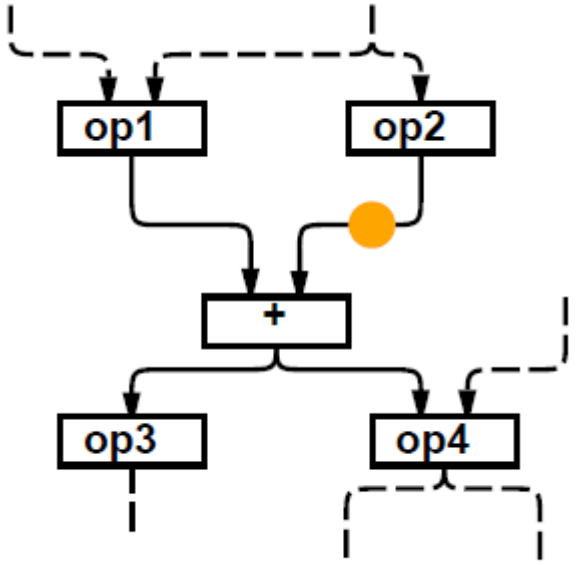


Early Dataflow Hardware Architectures

- Static Architecture (Jack Dennis /David Misunas 1975)
  - Each arc can hold only one token
  - Firing rule: token available on all input nodes and no token on output nodes
  - Single token per arc → second loop cannot begin until the previous one has ended – parallelism boils down to pipelining
- Dynamic Architecture (Watson/Gurd 1979)
  - Multiple incovations of a subgraph allowed
  - Each arc a bag of tokens with different tags (destinations, value)
  - Node fireable when on each input edge the same tag is available
  - Can take full advantage of pipelining and out of order execution.

MIT Tagged Token Dataflow Architecture

Conceptual



Encoding of graph

Program memory:

Opcode Destination(s)

109	op1	120L
113	op2	120R
120	+	141, 159L
141	op3	...
159	op4	... , ...

Encoding of token:

A "token" contains

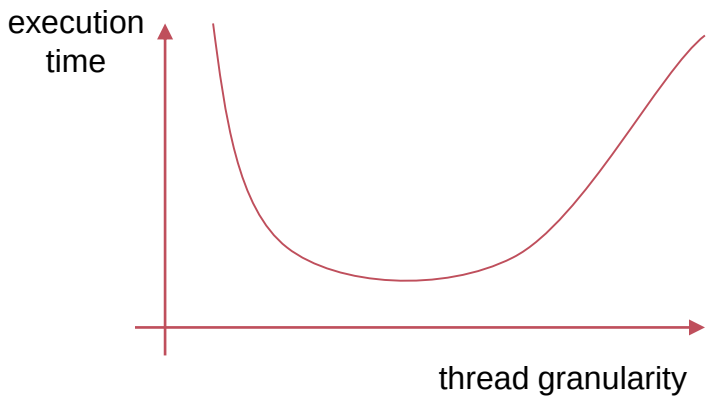
120R, 6.847 Destination instruction address, Left/Right port, Value

Possible reasons for the failure of early dataflow

- Totally new programming paradigm not accepted
- Dataflow languages almost invariably functional
- Programs in imperative languages hard to compile to a dataflow architecture
- Dataflow architecture operated on a too fine grained level
  - Von Neumann: *process* level granularity
  - Early dataflow: *instruction* level granularity

Hybrid Dataflow

Realization in the 1990s:
Dataflow and von Neumann architectures are not mutually exclusive but the two extremes of a continuum of possible computer architectures



→ Large-grain dataflow: each node contains an entire function expressed in a sequential language

Kahn Process Networks

- Seminal Paper „The Semantics of a Simple Language for Parallel Programming“ by Gilles Kahn, 1974.
- „Formal approach to the design of programming languages and system programming“
- Programming language based on Algol.

- KPNs describe a signal processing system:
  Processes communicate by passing data tokens through unidirectional FIFO channels
- KPN provide a distributed model of computation
- KPNs consist of a set of arbitrary deterministic sequential processes

Concepts

- Channels
- Processes
- Wait
  (***Blocking*** Receive)
- Send
  (Non-Blocking, unbounded fifos)
- Parallel invocation
  of processes in
  program body

```
Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
    Begin integer I ; logical B ;
          B := true ;
          Repeat Begin
(4)         I := if B then wait(U) else wait(V) ;
(7)         print (I) ;
(5)         send I on W ;
            B := ¬B ;
            end ;
    End ;
  Process g(integer in U ; integer out V, W) ;
    Begin integer I ; logical B ;
      B := true ;
      Repeat Begin
        I := wait (U) ;
        if B then send I on V else send I on W ;
        B := ¬B ;
        End ;
    End ;
(3) Process h(integer in U;integer out V; integer INIT);
    Begin integer I ;
      send INIT on V ;
      Repeat Begin
        I := wait(U) ;
        send I on V ;
        End ;
    End ;
  Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
  End ;
```

Fig.1. Sample parallel program S.

Determinism

Execution Model

- Channels are the only way for communication
- Communication for each line takes unpredictable but finite time
- Each process is either computing or waiting on ***one*** of its input lines.Processes are not allowed to test input channels for existence of tokens without consuming them (reads are blocking)
- Each process is a sequential process (given a specific input history for a process, the process must be determinstic). Timing / execution order may not influence the result

→ Determinism

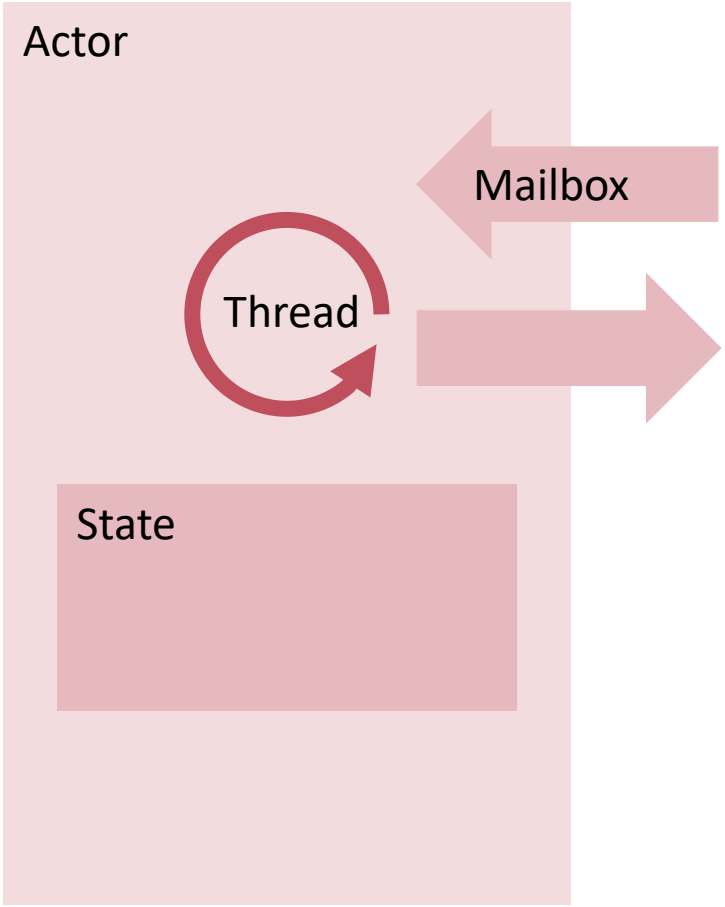
- The history of tokens produced on communication channel does not depend on execution order
- Every execution order that obeys the semantic of the process network produces the same result

The Actor Model*

Actor = Computational agent that maps communication to

- a finite set of communications sent to other actors (messages)
- a new behavior (state)
- a finite set of new actors created (dynamic reconfigurability)

- Undefined global ordering
- Asynchronous Message Passing
- Invented by Carl Hewitt 1973**



*Gul Agha (1986). Actors: A Model of Concurrent Computation in Distributed Systems. Doctoral Dissertation. MIT Press

**Carl Hewitt; Peter Bishop and Richard Steiger (1973). *A Universal Modular Actor Formalism for Artificial Intelligence*. IJCAI.

The Actor Model

Actor model provides a dynamic interconnection topology

- dynamically configure the graph during runtime (add channels)
- dynamically allocate resources

An actor sends messages to other actors using "direct naming", without indirection via port / channel / queue / socket (etc.)

Implemented in various languages such as Erlang, Scala, Ruby and in frameworks such as Akka (for Scala and Java)

Example: Erlang

Functional Programming Language

Developed by Ericsson for distributed fault-tolerant applications

- if no state is shared, recovering from errors becomes much easier

Open source

Concurrent, follows the actor model


```
-module(pingpong).
-export([start/1,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    register(pong, spawn(tut18, pong, [])),
    spawn(Ping_Node, tut18, ping, [3, node()]).
```

Erlang example

```
Start() ->
       Pid = spawn(fun() -> hello() end),

       Pid ! Hello,
       Pid ! bye.

hello() ->
       receive
           hello ->
              io:fwrite("Hello world\n"),
                  hello();
           bye ->
             io:fwrite("Bye cruel world\n"),
                     ok
       end.
```

new task (actor) that will execute the hello function
spawn returns address (Pid) of new task

Address (Pid) can be used to send messages to task

Erlang example

```
Start() ->
       Pid = spawn(fun() -> hello() end),

       Pid ! Hello,
       Pid ! bye.

hello() ->
       receive
           hello ->
              io:fwrite("Hello world\n"),
                  hello();
           bye ->
             io:fwrite("Bye cruel world\n"),
                     ok
       end.
```

new task (actor) that will execute the hello function
spawn returns address (Pid) of new task

Address (Pid) can be used to send messages to task

Messages sent to a task are put in a mailbox

Receive reads the first message in the mailbox, which is matched against patterns (similar to a switch statement)

Event-driven programming:
code is structured as reactions to events

Communicating Sequential Processes

Sir Charles Antony Richard Hoare (aka C.A.R. / Tony Hoare) (1978, 1985)

Formal language defining a process algebra for concurrent systems.

Operators seq (sequential) and par (parallel) for the hierarchical composition of processes.

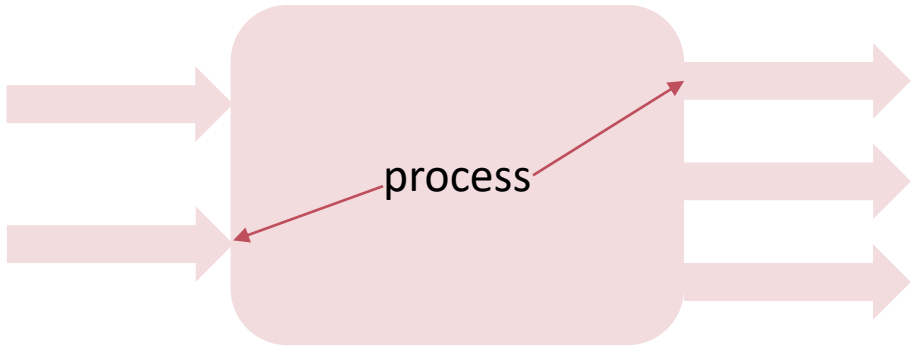
Synchronisation and Communication between parallel processes with Message Passing.

- Symbolic channels between sender and receiver
- Read and write requires a rendevouz (synchronous!)

CSP was firstly implemented in Occam.

CSP: Indirect Naming

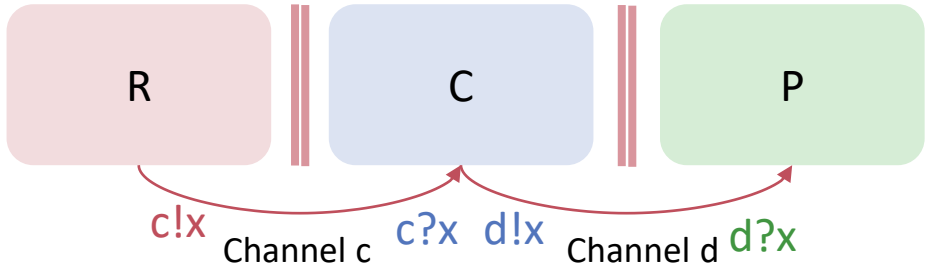
- Most message passing architectures (including CSP) include an intermediary entity (*port / channel*) to address send destination
- Process issuing send() specifies the port to which the message is sent
- Process issuing receive() specifies a port number and waits for the first message that arrives at the port



CSP Example (from Hoare's seminal Paper)

Conway's Problem

- Write a program that transforms a series of cards with 80-character columns in a series of printing lines with 125 characters each. Replace each "**" by "^"
- Separation into processes (Threads)
  `R par C par P`
  - R: Reading process reading 80-character records
  - C: Converting process converting "**" into "^"
  - W: Writing process: write records with 125 characters



CSP Example (from Hoare's seminal Paper)

[west :: DISASSEMBLE] || **X :: SQUASH** || east :: ASSEMBLE]

SQUASH

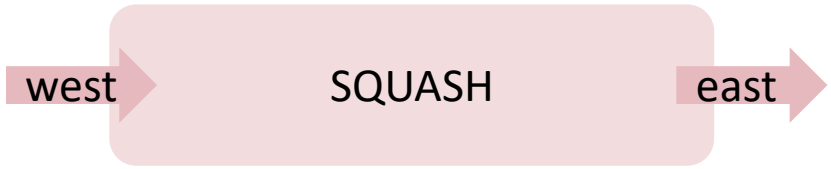
```
X ::
*[c:character; west?c →
        [c # asterisk → east!c
        |c = asterisk → west?c;
                [c # asterisk → east!asterisk; east!c
                |c = asterisk → east!upward arrow
                ]
        ]
]
```

Repetition of guarded command

Guarded receive

Blocking send

Guarded alternatives



OCCAM

First programming language to implement CSP (1983)

```
ALT
  count1 < 100 & c1 ? data
    SEQ
      count1 := count1 + 1
      merged ! data
  count2 < 100 & c2 ? data
    SEQ
      count2 := count2 + 1
      merged ! data
  status ? request
    SEQ
      out ! count1
      out ! count2
```

Superpascal (Per Brinch Hansen (1994))

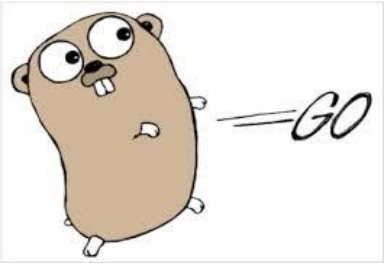
Typed channels, processes, parallel statements, message passing

```
type channel= *(boolean, number);

procedure ring(a: number; var prime: boolean);
var left, right: channel;
begin
  open(left, right);
  parallel
  pipeline(left, right) | master(a, prime, left, right)
  end
end;

procedure node(i: integer;
        left, right: channel);
var a: number; j: integer;
        composite: boolean;
begin
        receive(left, a);
        if i < p then send(right, a);
        test(a, i, composite);
        send(right, composite);
        for j := 1 to i - 1 do
        begin
                receive(left, composite);
                send(right, composite)
        end
end;
```

```
procedure master(
        a: number; var prime: boolean;
        left, right: channel);
var
        i: integer; composite: boolean;
begin
        send(left, a); prime := true;
        for i := 1 to p do
                begin
                        receive(right, composite);
                        if composite then
                                prime := false
                end
end;

procedure pipeline(left, right: channel);
        type row = array [0..p] of channel;
        var c: row; i: integer;
begin
        c[0] := left; c[p] := right;
        for i := 1 to p ⁄ 1 do
                open(c[i]);
        forall i := 1 to p do
                node(i, c[i-1], c[i])
end;
```

Go programming language

Concurrent programming language from Google



Language support for:

- Lightweight tasks (called goroutines)
- Typed channels for task communications
  - channels are synchronous (or unbuffered) by default
  - support for asynchronous (buffered) channels

Inspired by CSP

Language roots in Algol Family: Pascal, Modula, Oberon [Prof. Niklaus Wirth, ETH]

[One of the inventors of Go: Robert Griesemer holding a PhD from ETH]

Go example

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs,done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

Go example

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs,done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

Create two channels:
- msgs: for strings
- done: for boolean values

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

Go example

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs,done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

Create a new task (goroutine), that will execute function hello with the given arguments

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

Go example

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs,done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

Hello takes two channels as arguments for communication

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

Go example

```
func main() {

        msgs := make(chan string)
        done := make(chan bool)

        go hello(msgs,done);

        msgs <- "Hello"
        msgs <- "bye"

        ok := <-done

        fmt.Println("Done:", ok);
}
```

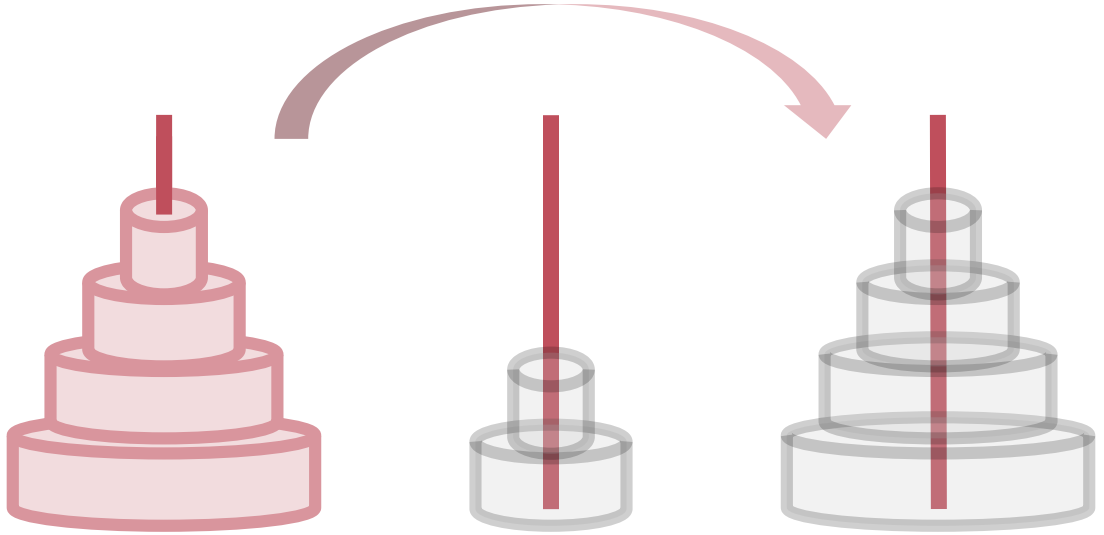
Write arguments to msgs channel

Read result via done channel

```
func hello(msgs chan string,
           done chan bool) {

        for {
                msg := <-msgs
                fmt.Println("Got:", msg)

                if msg == "bye" {
                        break
                }
        }

        done <- true;
}
```

Towers of Hanoi (sequential)

```
package main
import "fmt"

func Hanoi(n, f, t, u int) {
	if n<=1 {
		fmt.Println(f, "->", t)
	} else{
		Hanoi(n-1, f, u, t);
		fmt.Println(f, "->", t);
		Hanoi(n-1, u, t, f);
	}
}

func main() {
	Hanoi(4,1,3,2)
}
```

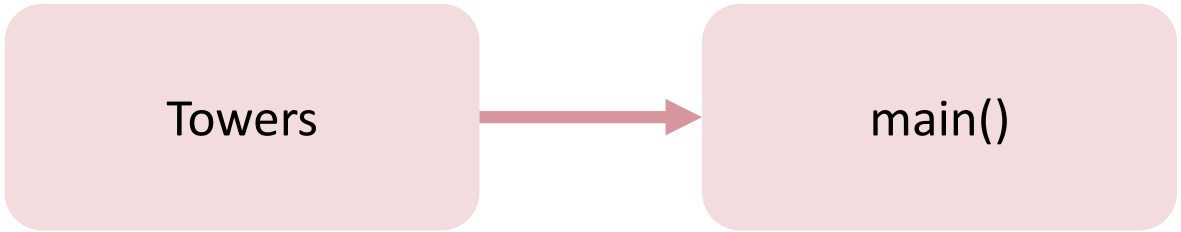


Q: How can I easily return the moves in this sequence to main()?

Towers of Hanoi with go-routine

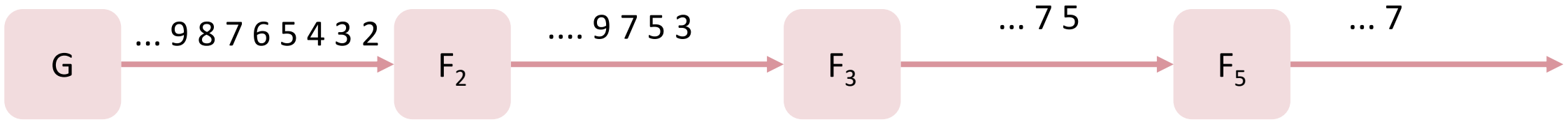
```
func Hanoi(ch chan<- int, n, f, t, u int) {
    if n<=1 {
        ch <- f
        ch <- t
    } else{
        Hanoi(ch, n-1, f, u, t);
        ch <- f
        ch <- t
        Hanoi(ch, n-1, u, t, f);
    }
}

func Towers(ch chan<- int, n, f, t, u int) {
    Hanoi(ch,n,f,t,u);
    ch <- -1
}
```

```
func main() {
    ch := make(chan int)
    go Towers(ch, 4,1,3,2)
    for ;; {
        i := <-ch
        if i<0 {return}
        j := <-ch
        fmt.Println(i,"<-",j)
    }
}
```



Concurrent prime sieve

Each station removes multiples of the first element received and passes on the remaining elements to the next station



source code copied from golang.org

Concurrent prime sieve

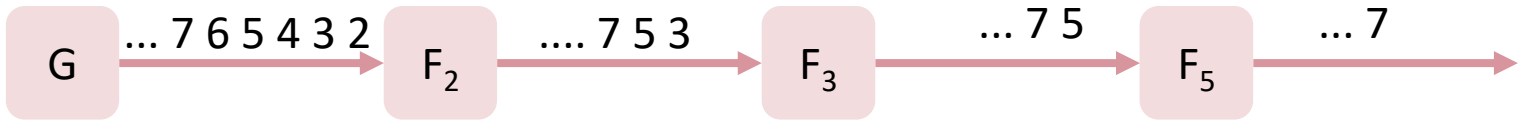
```
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}
```

G

```
func Filter(in <-chan int, out chan<- int, prime int) {
    for {
        i := <-in  // Receive value from 'in'.
        if i%prime != 0 {
            out <- i // Send 'i' to 'out'.
        }
    }
}
```

F_{prime}

```
func main() {
    ch := make(chan int)
    go Generate(ch)
    for i := 0; i < 10; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go Filter(ch, ch1, prime)
        ch = ch1
    }
}
```



source code copied from golang.org

Message Passing Interface (MPI)

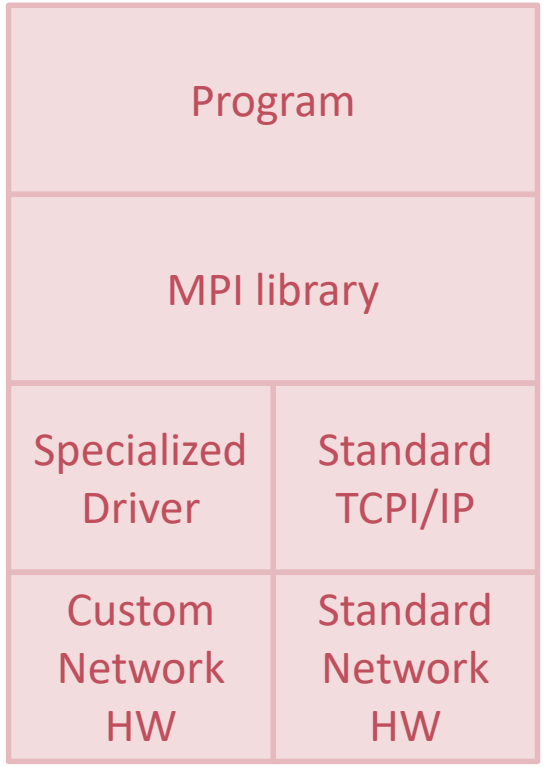
Message passing **libraries**:

- PVM (Parallel Virtual Machines) 1980s
- **MPI** (Message Passing Interface) 1990s

MPI = Standard API

- Hides Software/Hardware details
- Portable, flexible
- Implemented as a library

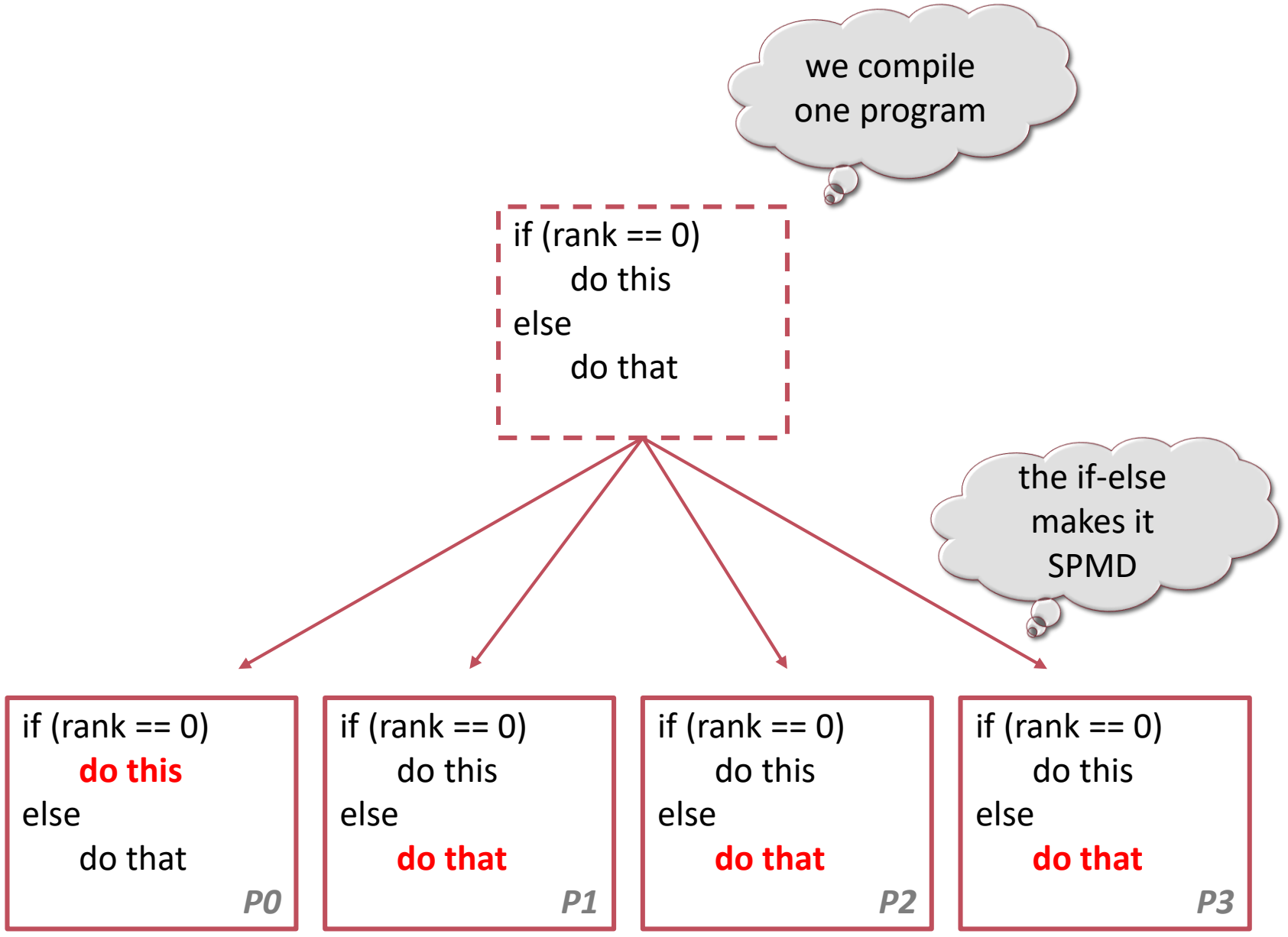
Largely used for distributed HPC.



SPMD

Single **P**rogram

Multiple **D**ata
(Multiple Instances)



Synchronous / Asynchronous vs Blocking / Nonblocking

Synchronous / Asynchronous

- about communication between sender and receiver

Blocking / Nonblocking

- about local handling of data to be sent / received

MPI Send and Receive Defaults

Send

- blocking,
- synchrony **implementation dependent**
  - depends on existence of buffering, performance considerations etc

Danger of Deadlocks.
Don't make any assumptions!

Receive

- blocking

There are a lot of different variations of this in MPI.

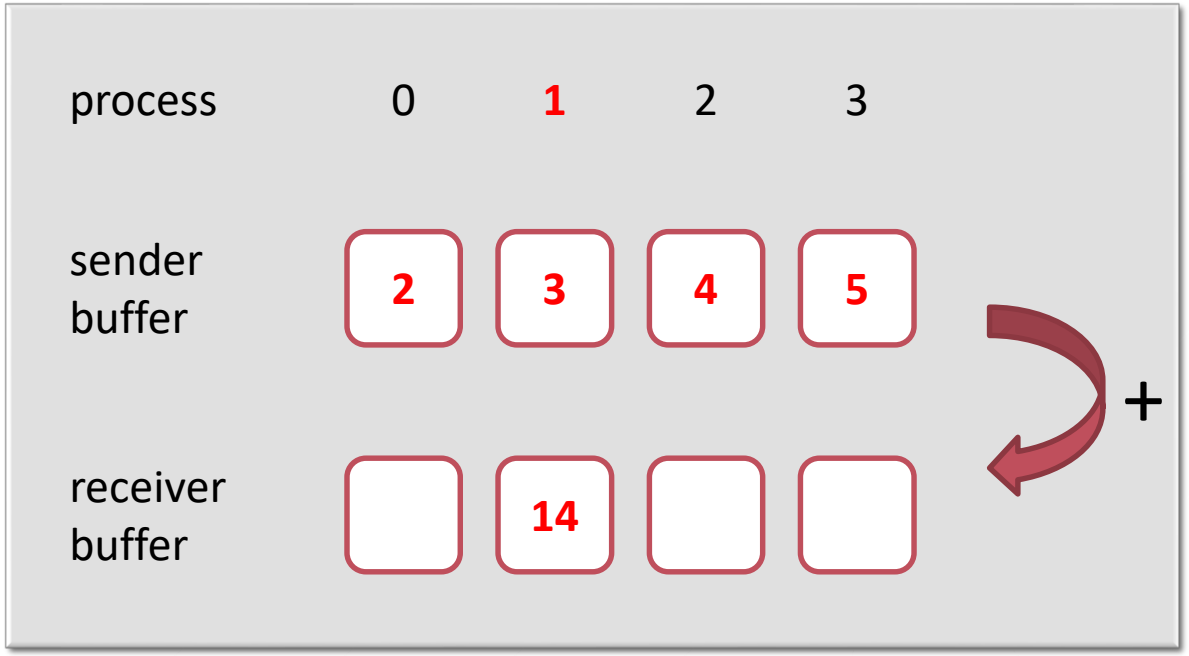
Group Communication

MPI supports sending messages between groups of processors

- not absolutely necessary for programming
- but essential for performance

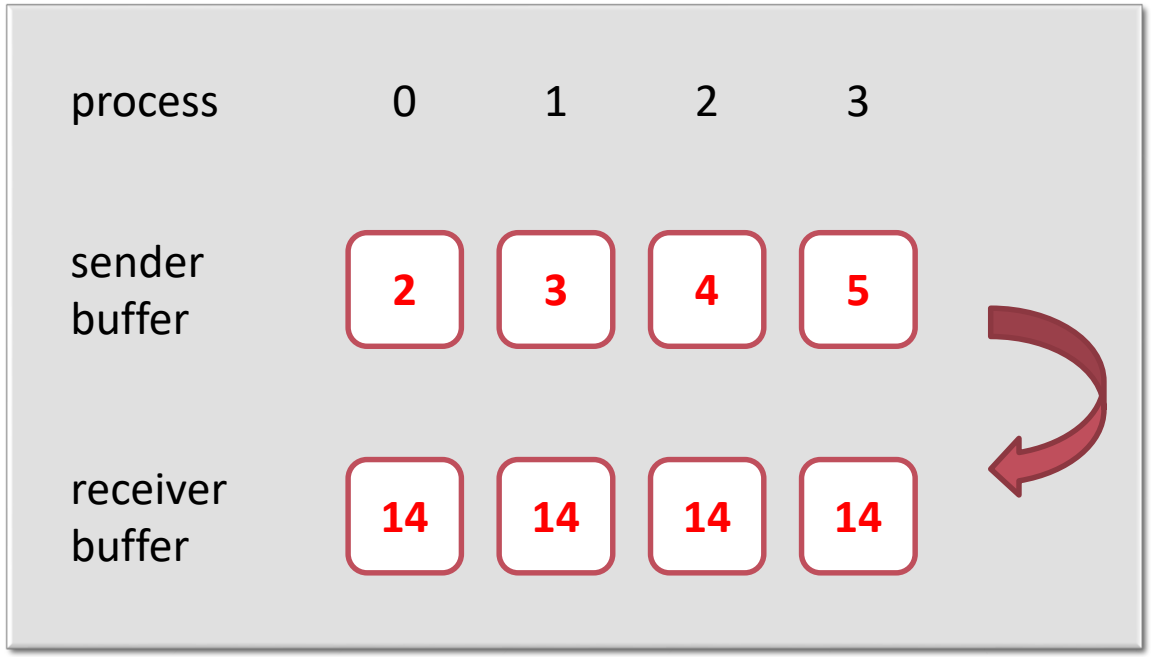
Examples: broadcast, gather, scatter, reduce, barrier

Reduce

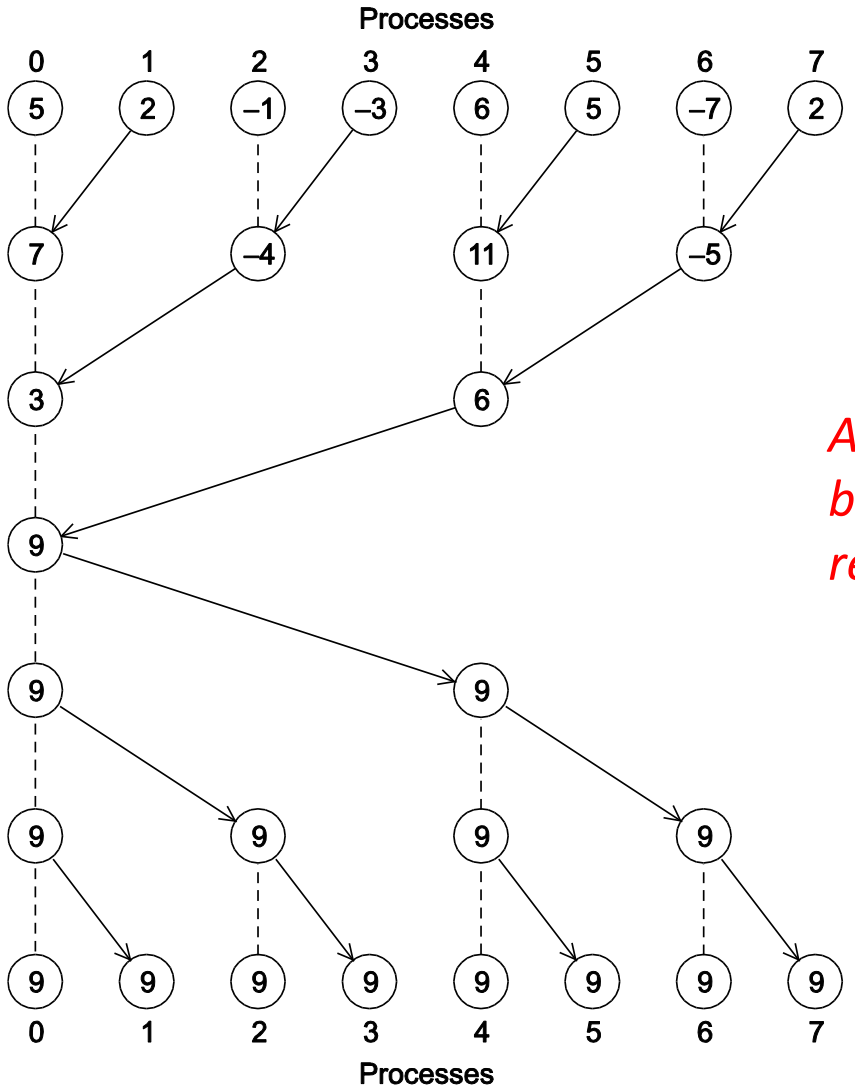


Allreduce

Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

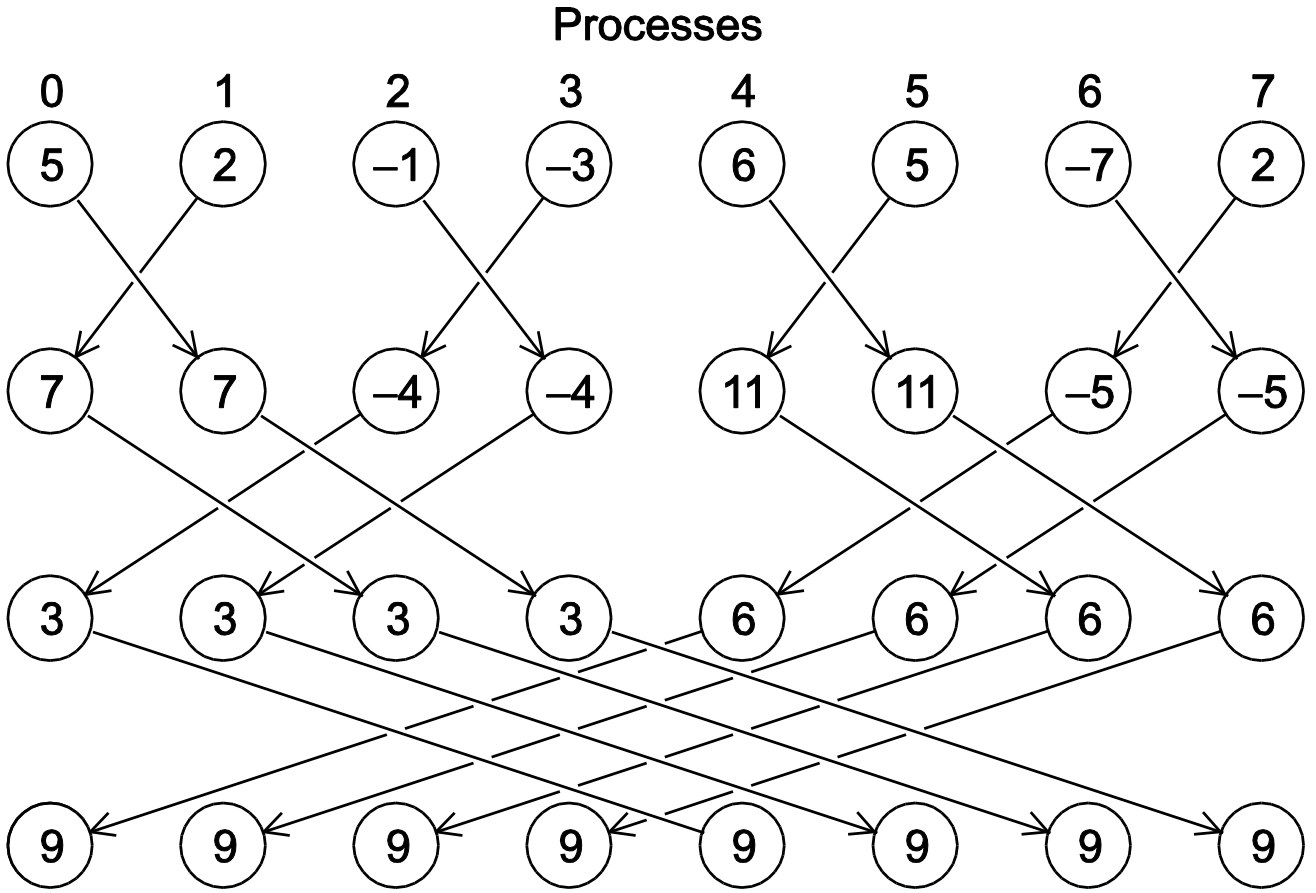


Allreduce = Reduce + Broadcast?



A global sum followed by distribution of the result.

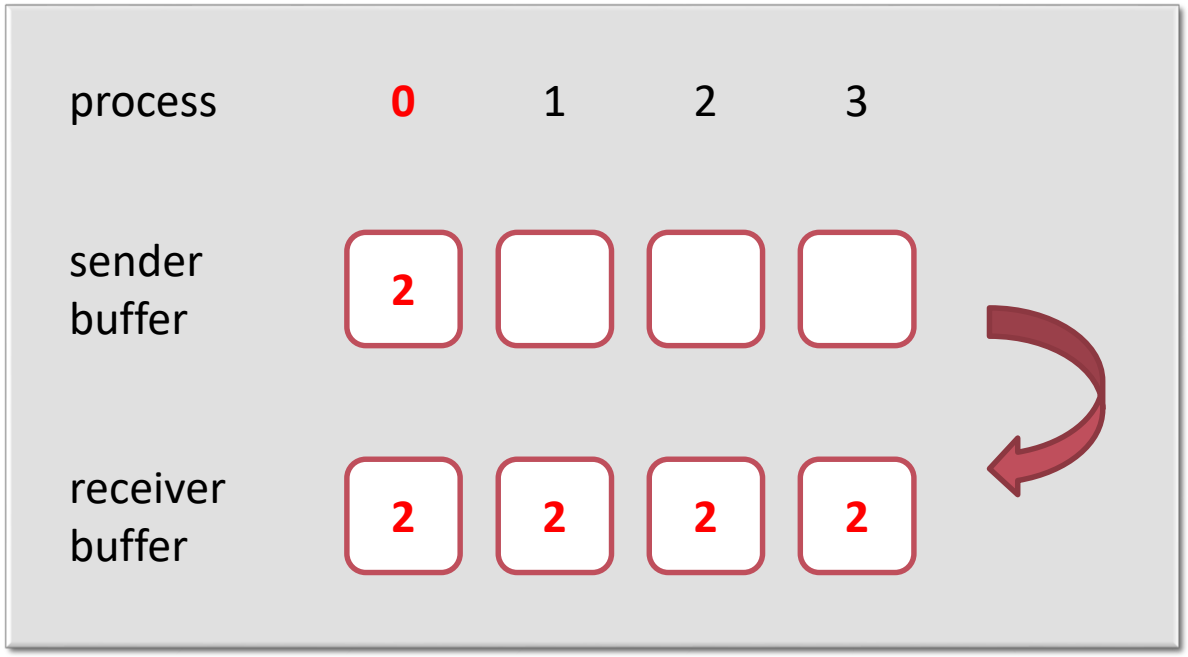
Allreduce ≠ Reduce + Broadcast



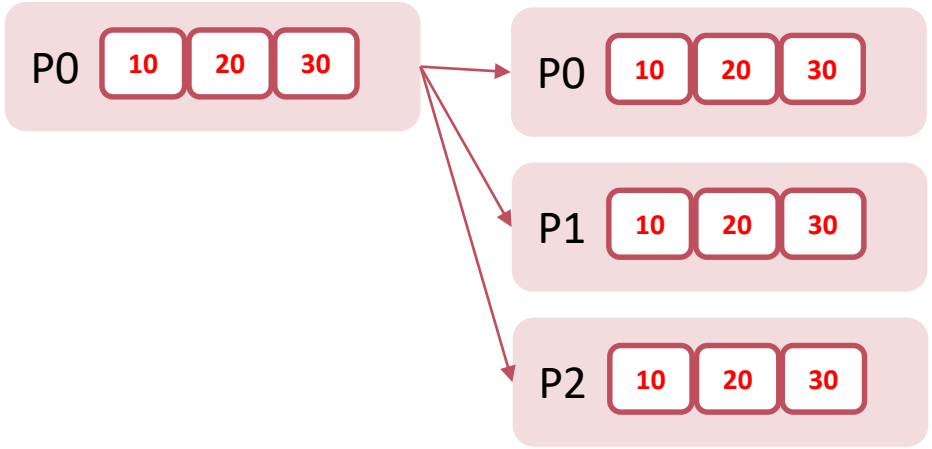
A butterfly-structured global sum.

Broadcast

Data belonging to a single process is sent to all of the processes in the communicator.

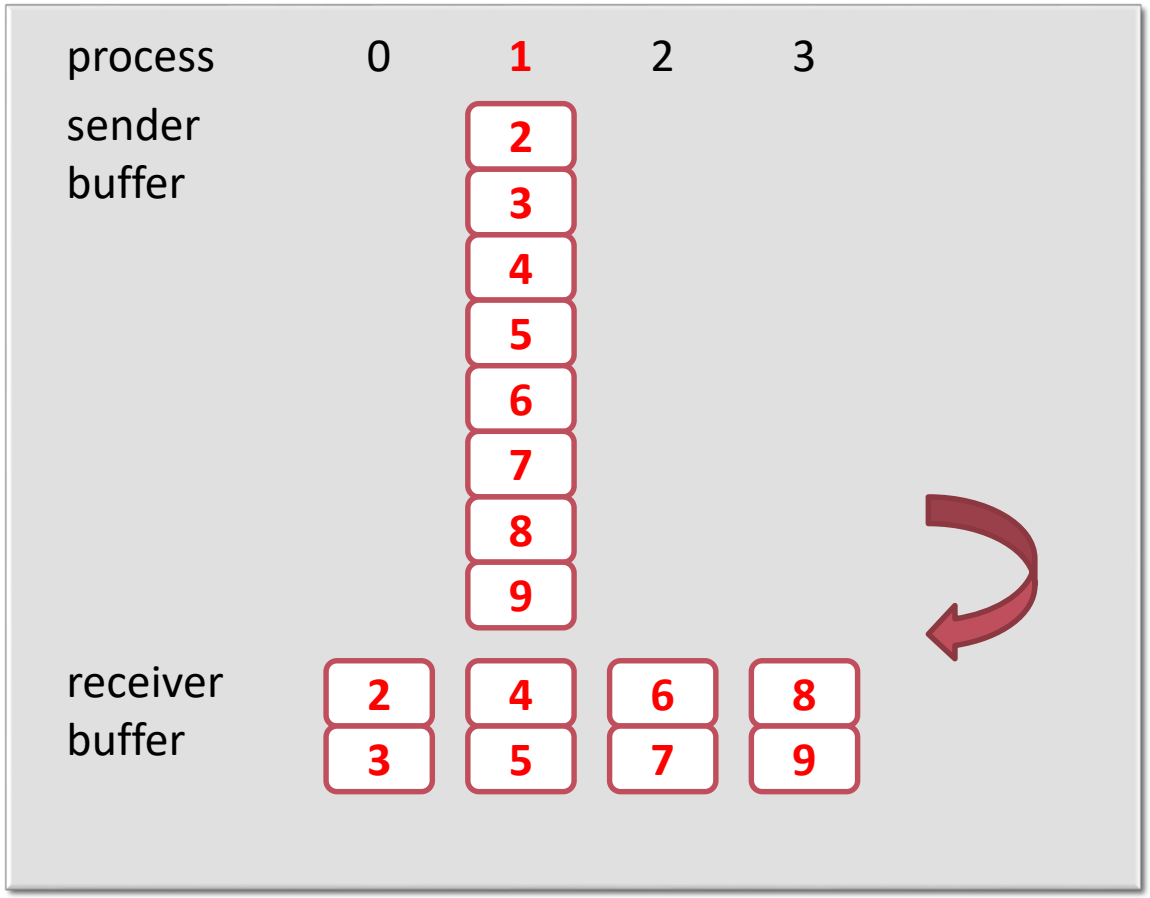


$$y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{10} \\ \mathbf{20} \\ \mathbf{30} \end{bmatrix}$$

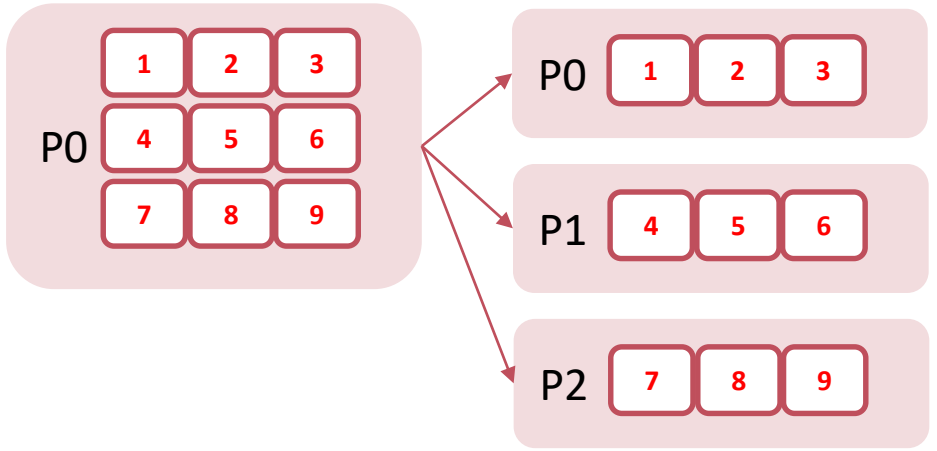


Scatter

Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

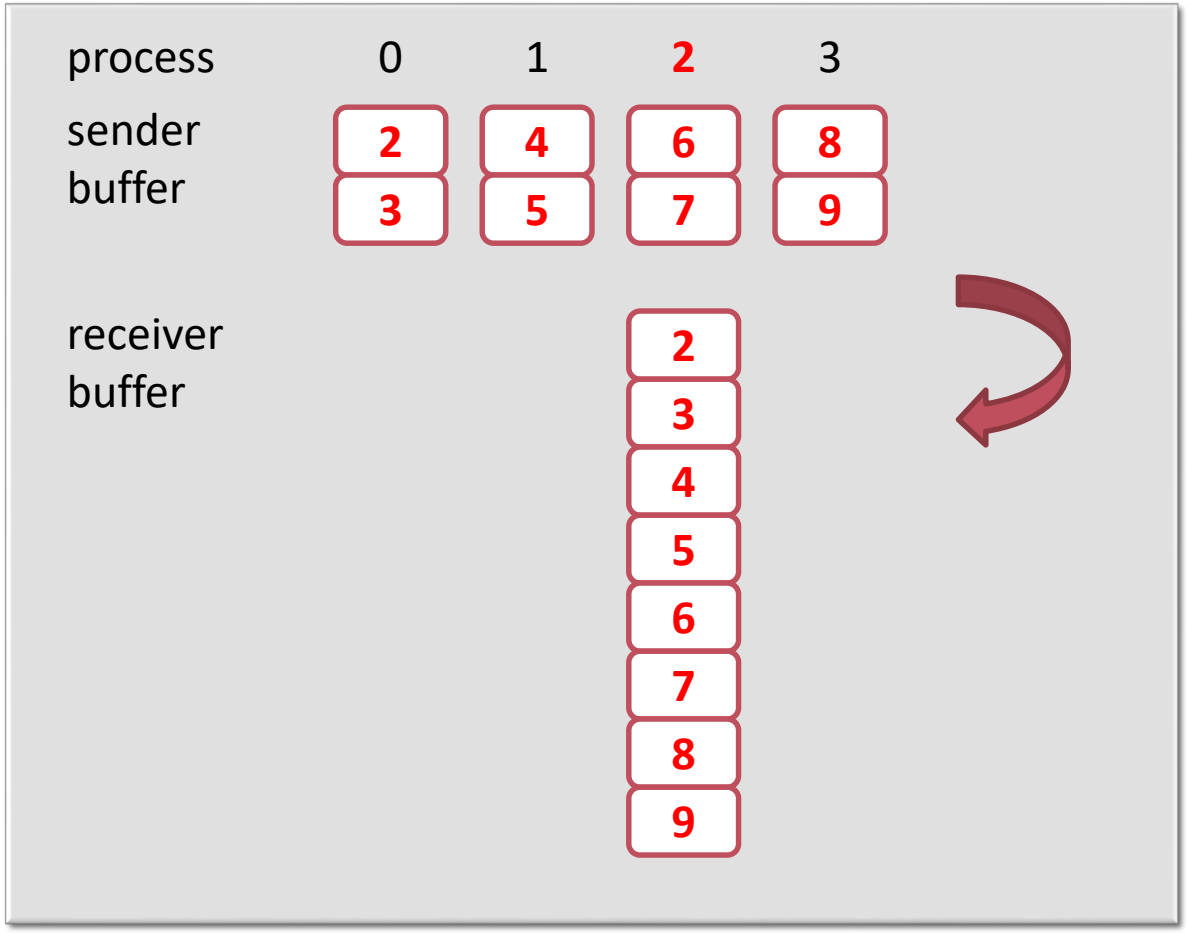


$$y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$

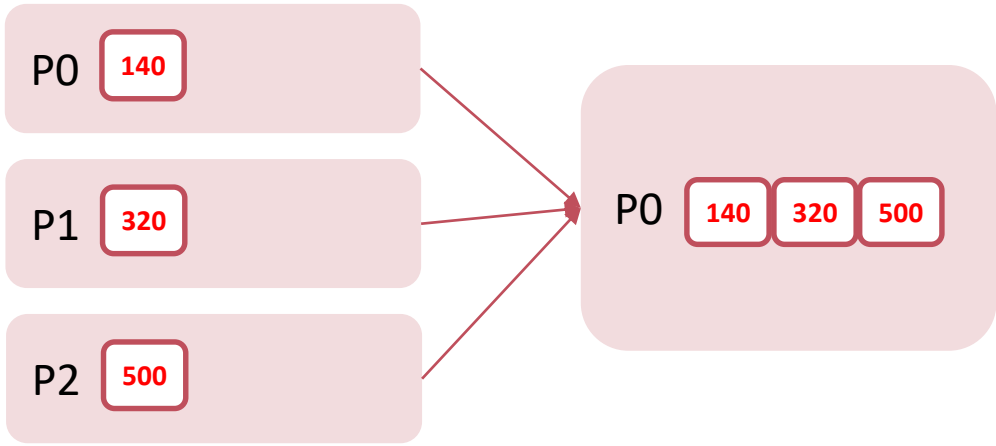


Gather

Collect all of the components of the vector onto destination process, then destination process can process all of the components.



$$y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$



TRM PROCESSOR DETAILS

TRM Architectural State

- PC
- 8 registers
- flag registers
- Memory (configurable)
  - nK * 36 bits instruction memory (1k = 1024)
  - nk * 32 bits data memory

PL vs. HDL

Programming Language

- Sequential execution
- No notion of time

```
var a,b,c: integer;

a := 1;
b := 2;
c := a + b;
```

unknown mapping to machine cycles

Hardware Description Language

- Continuous execution (combinational logic)

```
wire [31:0] a,b,c;

assign a=1;
assign b=2;
assign c=a+b;
```

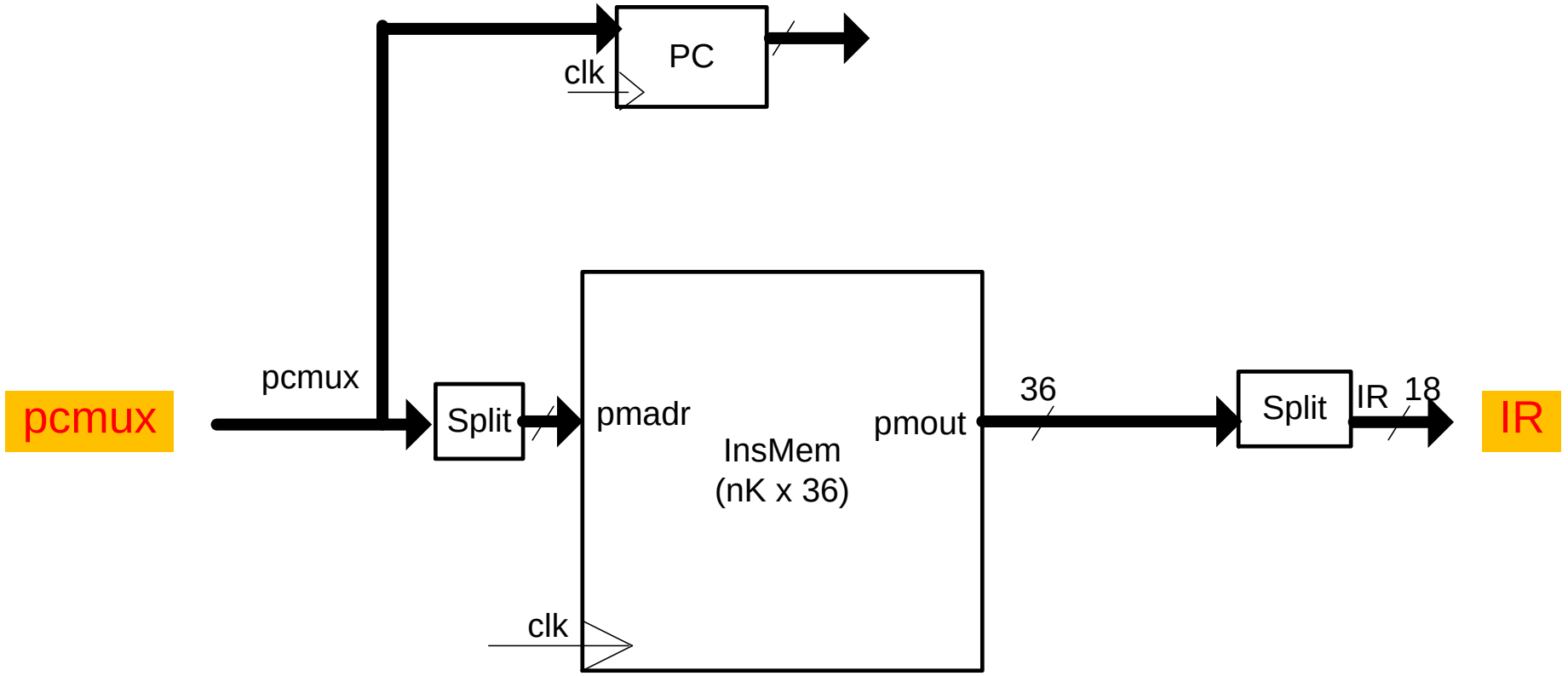
no memory associated

- Synchronous execution (register transfer)

```
reg [31:0] a,b,c;

always @ (posedge clk)
begin
  a <= 1;
  b <= 2;
  c <= a+b;
end;
```

synchronous at rising edge of the clock

Single-Cycle Datapath: arithmetical logical instruction fetch

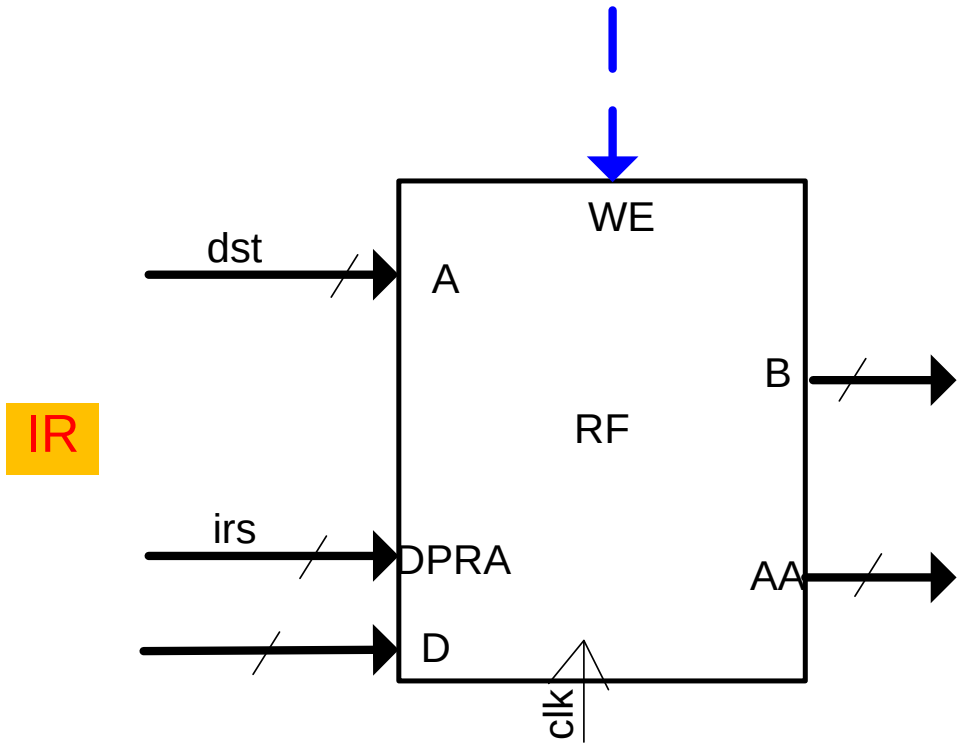


Single-Cycle Datapath: instruction fetch

```
wire [PAW-1:0] pcmux, nxpc;
wire [17:0] IR;
reg [PAW-1:0] PC;

IM #(.BN(IMB)) imx(.clk(clk), .pmadr({{{32-PAW}{1'b0}},pcmux[PAW-1:1]}),
   .pmout(pmout));

assign IR = (~rst)? NOP: (PC[0]) ? pmout[35:18] : pmout[17:0];

always @ (posedge clk) begin
  if (~rst) PC <= 0;
  else if (stall0)
      PC <= PC;
  else
      PC <= pcmux;
end
```

Single-Cycle Datapath: register read

- **STEP 2:** Read source operands from register file



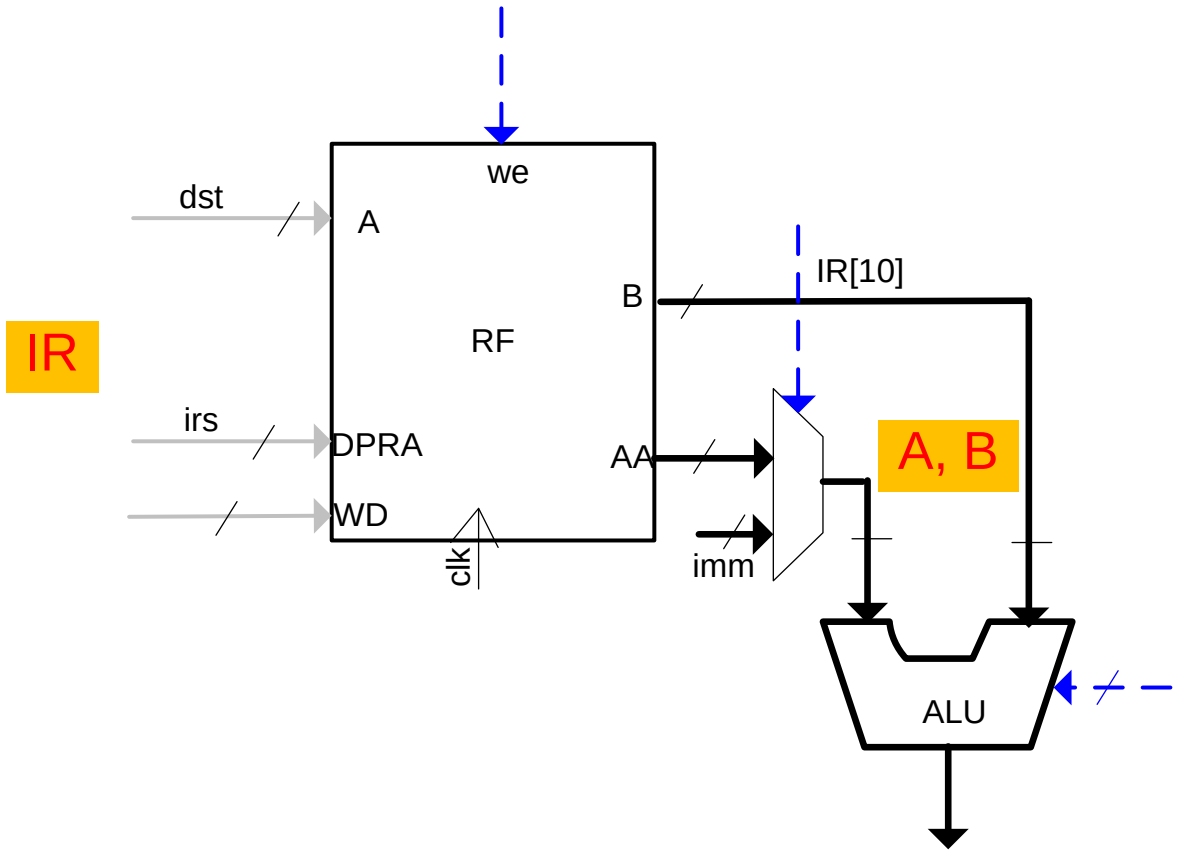

```
wire [2:0] rd, rs;
wire regWr;
wire [31:0] rdOut, rsOut;

//register file
…

assign irs = IR[2:0];
assign ird = IR[13:11];
assign dst = (BL)? 7: ird;
```

source register

destination register

Single-Cycle Datapath: ALU

- **STEP 3:** Compute the result via ALU




```
wire [31:0] AA, A, B, imm;
wire [32:0] aluRes;

assign A = (IR[10])? AA:
                {22'b0, imm};

assign minusA = {1'b0, ~A} + 33'd1;
assign aluRes =
    (MOV)? A:
    (ADD)? {1'b0, B} + {1'b0, A} :
    (SUB)? {1'b0, B} + minusA :
    (AND)? B & A :
    (BIC)? B & ~A :
    (OR)? B | A :
    (XOR)? B ^ A :
    ~A;
```

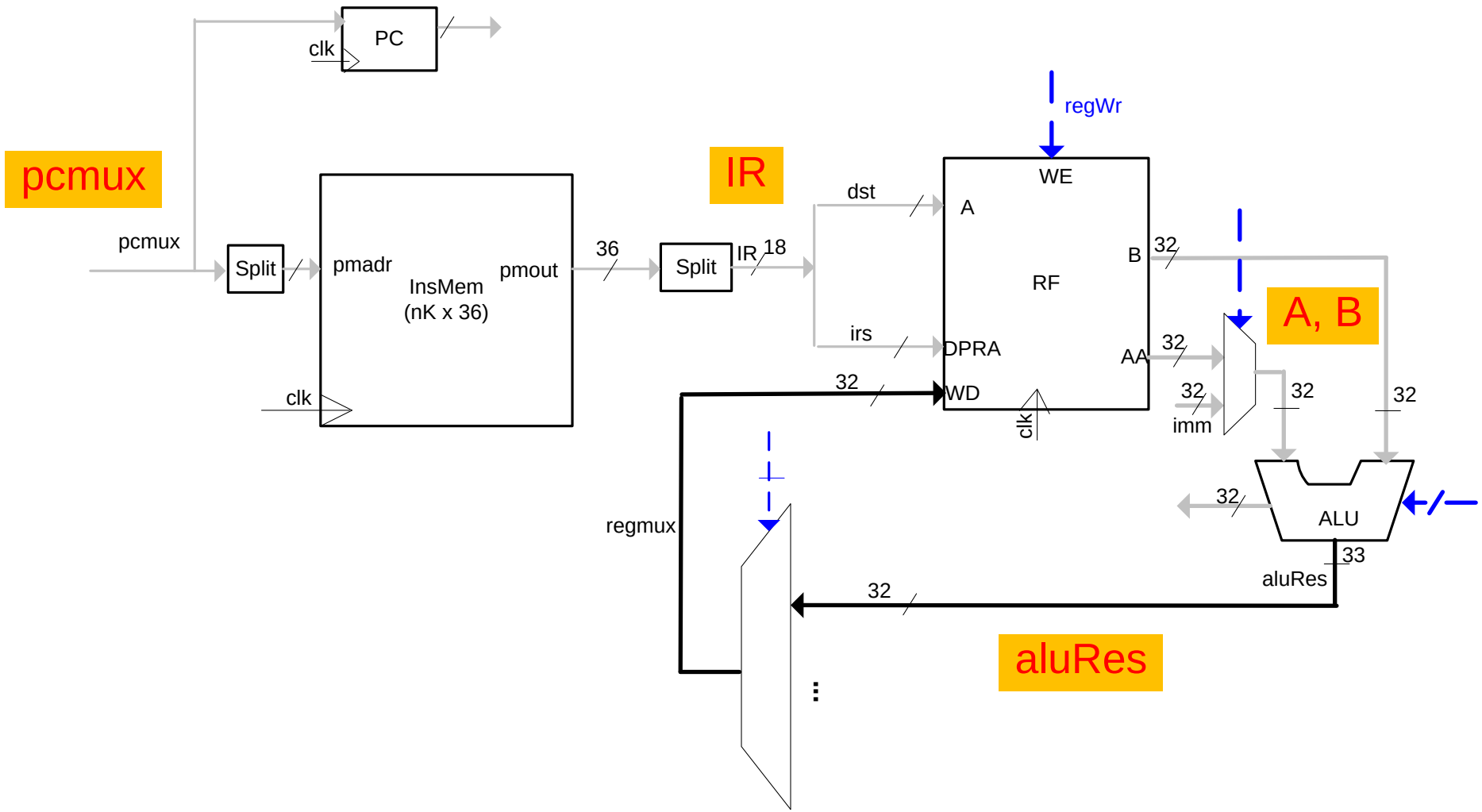
Control Path

```
assign vector = IR[10] & IR[9] & ~IR[8] & ~IR[7];
assign op = IR[17:14];

assign MOV = (op == 0);
assign NOT = (op == 1);
assign ADD = (op == 2);
assign SUB = (op == 3);
assign AND = (op == 4);
assign BIC = (op == 5);
assign OR  = (op == 6);
assign XOR = (op == 7);
assign MUL = (op == 8) & (~IR[10] | ~IR[9]);
assign ROR = (op == 10);
assign BR  = (op == 11) & IR[10] & ~IR[9];
assign LDR = (op == 12);
assign ST  = (op == 13);
assign Bc  = (op == 14);
assign BL  = (op == 15);
assign LDH = MOV & IR[10] & IR[3];
assign BLR = (op == 11) & IR[10] & IR[9];
```

$IR_{17:14}$	Function
0000	B := A
0001	B := ~A
0010	B := B + A
0011	B := B – A
0100	B := B & A
0101	B := B & ~A
0110	B := B | A
0111	B := B ^ A

Single-Cycle Datapath: write back to Rd

- **STEP 4:** Write result back to Rd

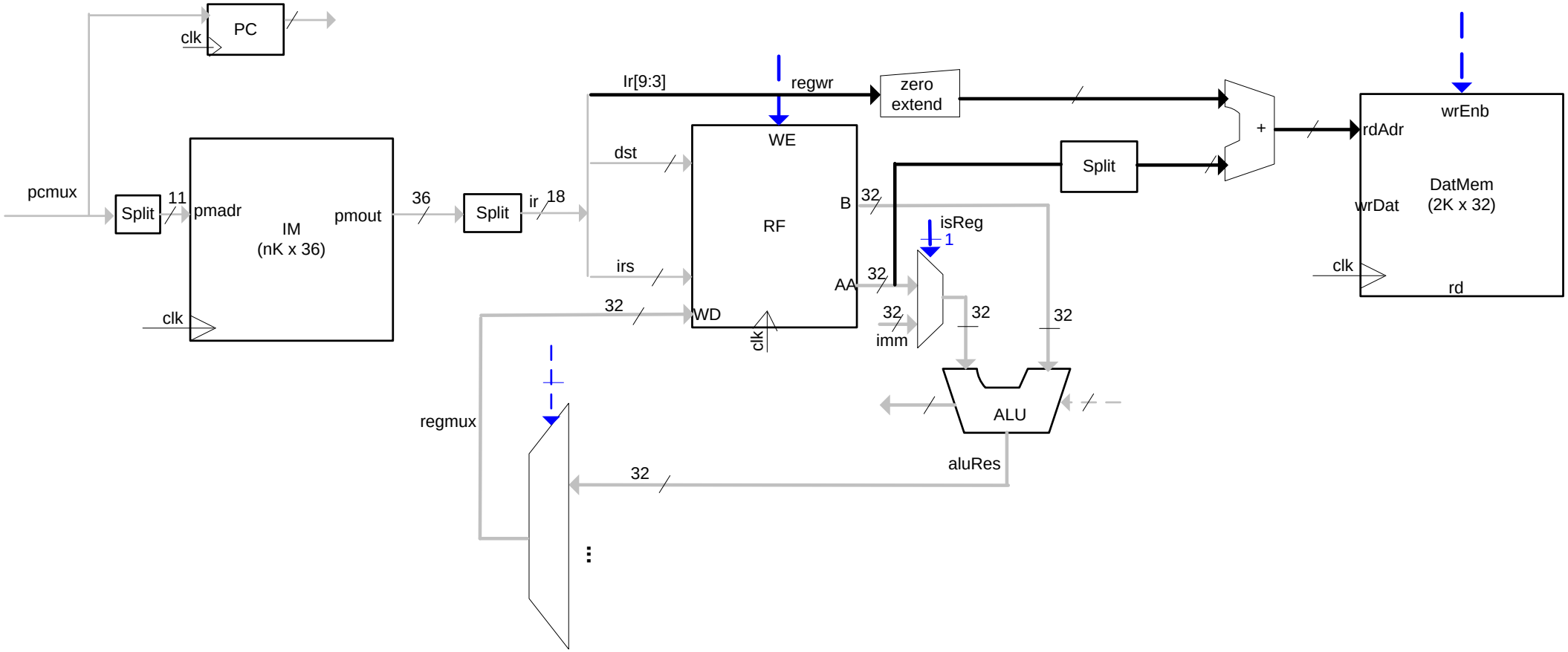


Single-Cycle Datapath: write back to Rd

```
wire [31:0] regmux;
wire regwr;

...

assign regwr = (~LD | stall1) & ~ST & ~Bc & ~BR;

assign regmux =
    LDR ?
    ioenbReg ? InbusReg : dmout
    : ROR ? s3
    : (BL | BR) ? {{DW-IAW{1'b0},nxpc}
    : aluRes
```

Single-Cycle Datapath: LD

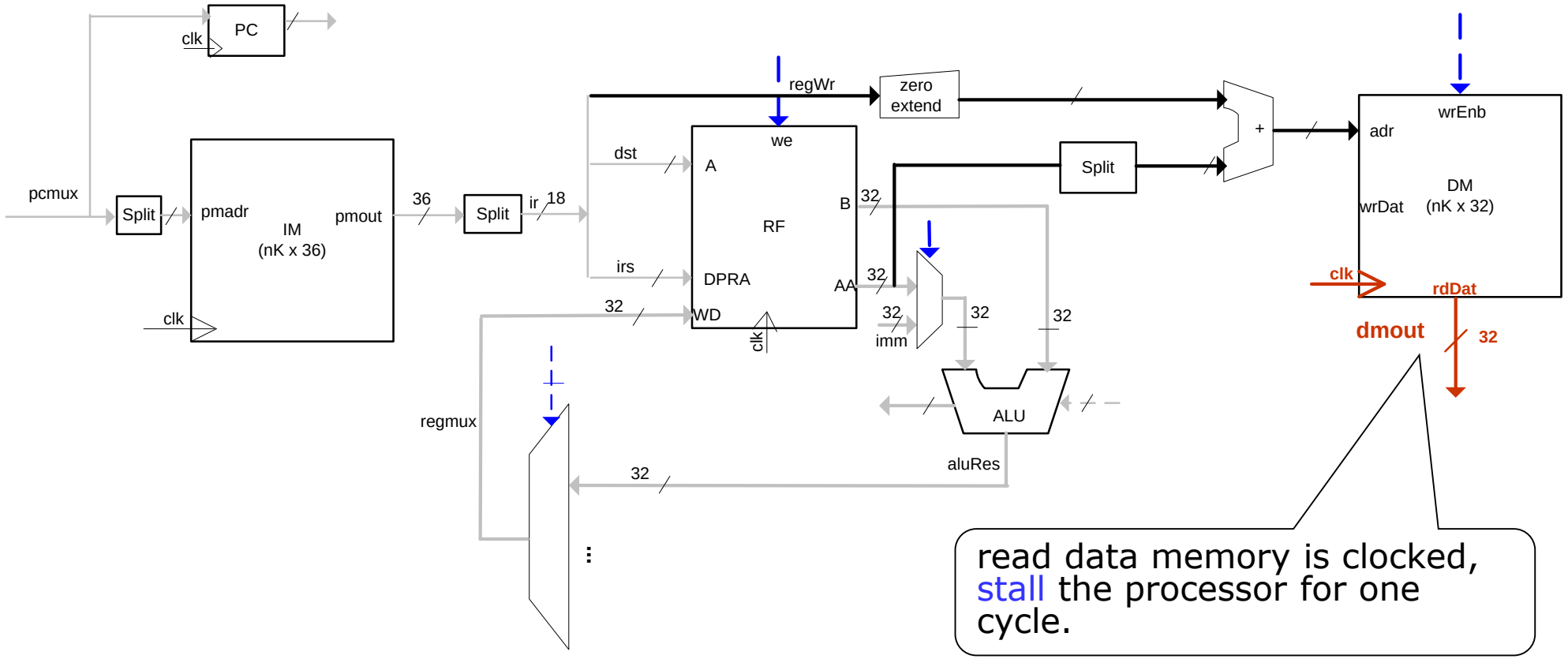
- **STEP 1:** Fetch instruction
- **STEP 2:** Read source operand from the register file
- **STEP 3:** Compute the memory address



from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

Single-Cycle Datapath: LD

- **STEP 3:** Compute the memory address
- **STEP 4:** Read data from data memory

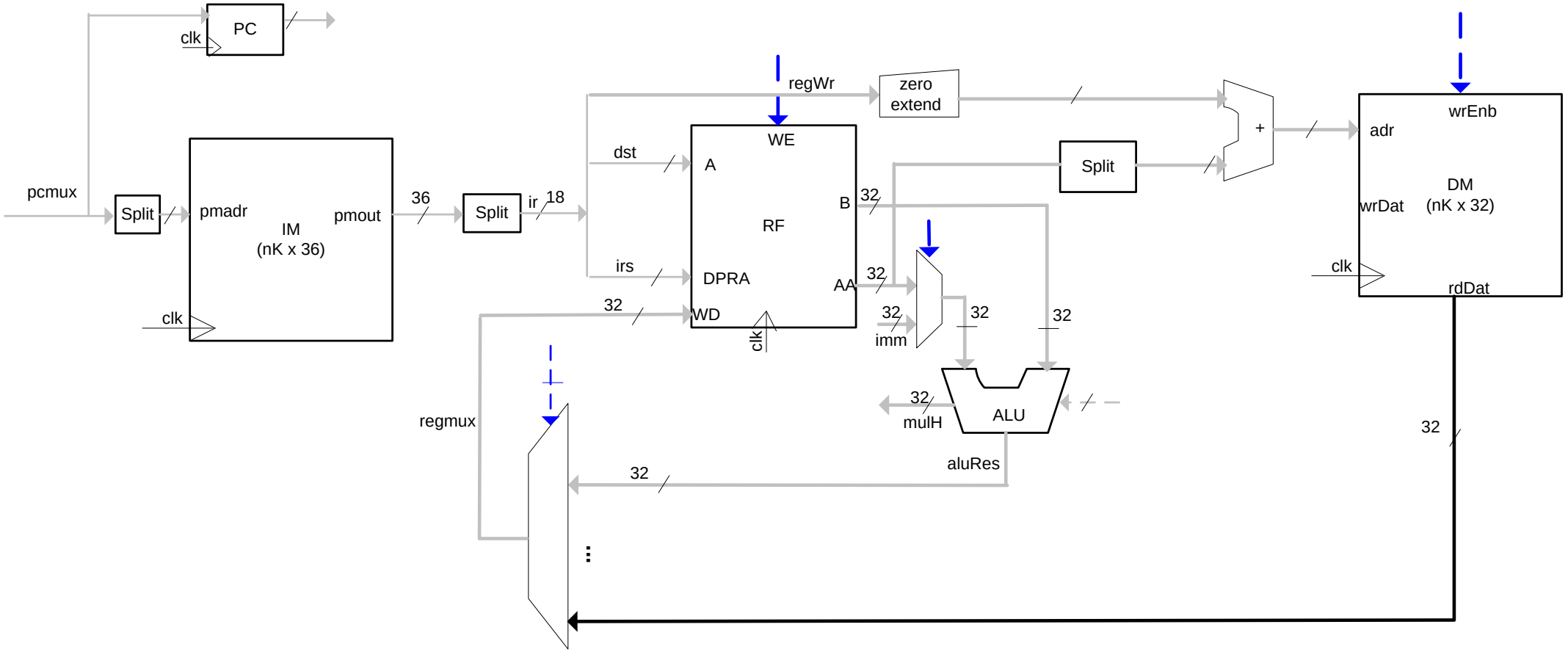


TRM Stalling

- stop fetching next instruction, pcmux keeps the current value
- disable register file write enable and memory write enable signals to avoid changing the state of the processor.
  - only LD and MUL instructions stall the processor.
  - dmwe signal is not affected.
  - regwr signal is affected.

Single-Cycle Datapath: LD

- **STEP 4:** Read data from data memory
- **STEP 5:** Write data back into the register file



from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

TRM: LD

LD rd, [rs+offset]

Verilog code in TRM module

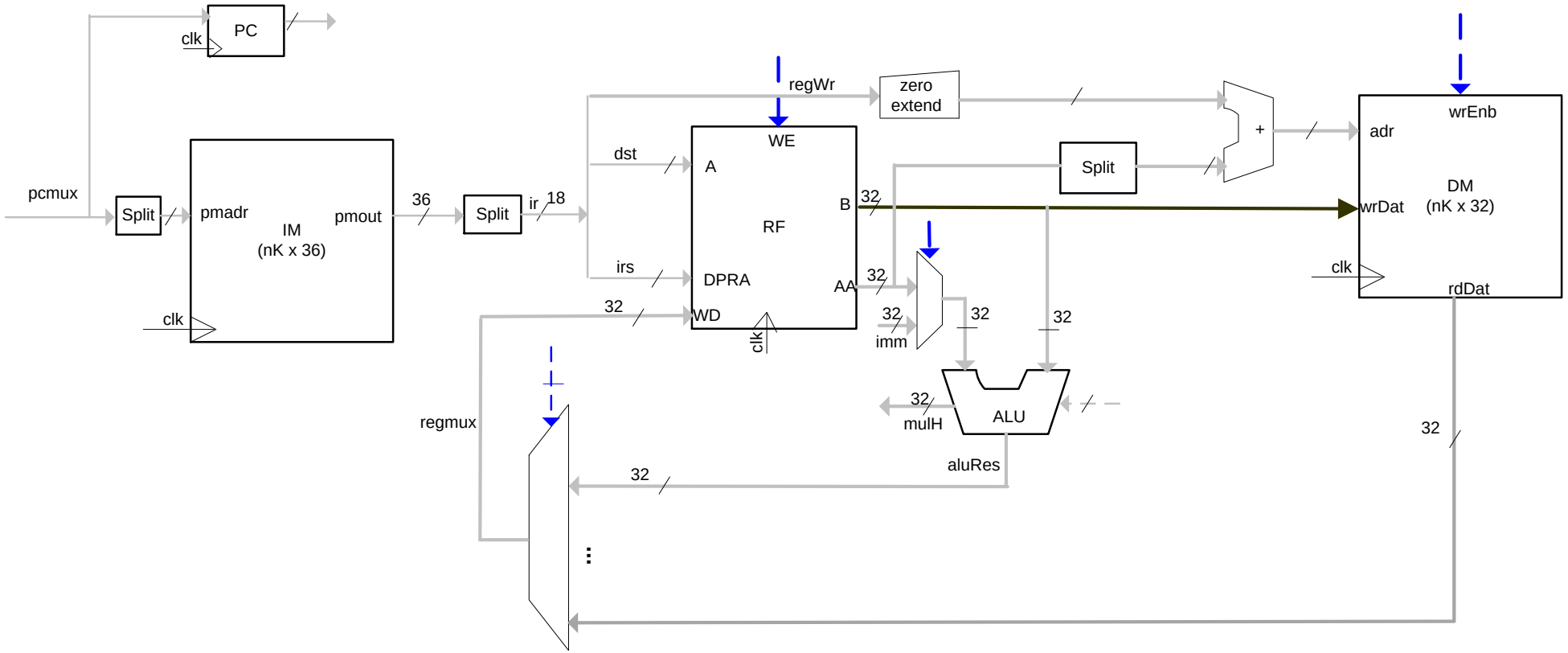
```
wire [31:0] dmout;
wire [DAW:0] dmadr;
wire [6:0] offset;
reg IoenbReg;

//register file
…
Assign dmadr = (irs == 7) ? {{{DAW-6}{1'b0}}, offset} :(AA[DAW:0] + {{{DAW-6}{1'b0}}, offset});
assign ioenb = &(dmadr[DAW:6]);
assign rfWd = ...
              (LDR & ~IoenbReg)? dmout:
              (LDR & IoenbReg)? InbusReg: //from IO
              ...;
always @(posedge clk)
    IoenbReg <= ioenb;
```

Src register = Ir ignored (Harvard architecture!)

I/O space: Uppermost 2^6 bytes in data memory

Single-Cycle Datapath: ST

- **STEP 1:** Fetch instruction
- **STEP 2:** Read source operand from the register file
- **STEP 3:** Compute the memory address
- **STEP 4:** Write data into the data memory



from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

Single-Cycle Datapath: ST

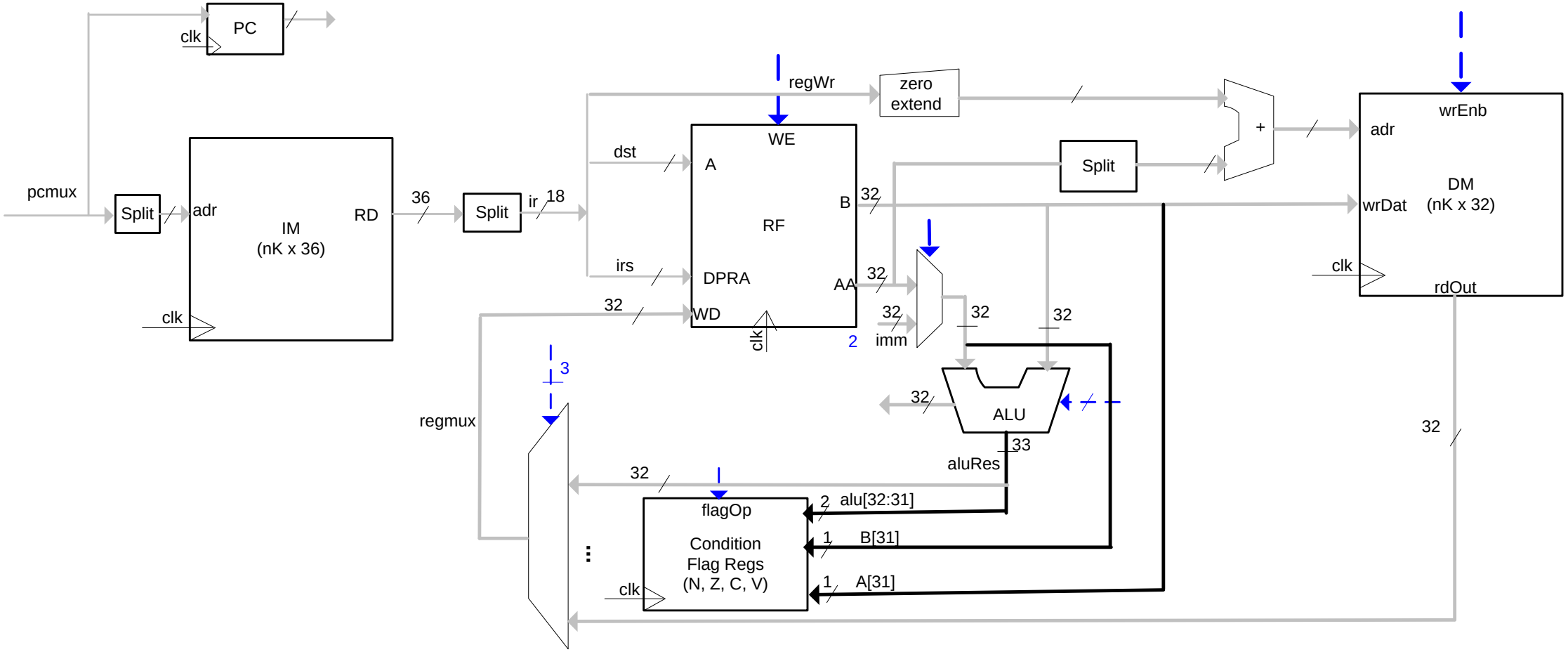
- **STEP 3:** Compute the memory address
- **STEP 4:** Write data into the data memory

```
wire [31:0] dmin;
wire dmwr;

//register file
…
DM #(.BN(DMB)) dmx (.clk(clk),
   .wrDat(dmin),
    .wrAdr({{{31-DAW}{1'b0}},dmadr}),
    .rdAdr({{{31-DAW}{1'b0}},dmadr}),
    .wrEnb(dmwe),
    .rdDat(dmout));

Assign dmwe = ST & ~IR[10] & ~ioenb;
assign dmin = B;
```

Single-Cycle Datapath: set flag registers

```
always @ (posedge clk, negedge rst) begin  // flags
    handling
  if (~rst) begin N <= 0; Z <= 0; C <= 0; V <= 0; end
  else begin
    if (regwr) begin
        N <= aluRes[31];
        Z <= (aluRes[31:0] == 0);
        C <= (ROR & s3[0]) | (~ROR & aluRes[32]);
        V <= ADD & ((~A[31] & ~B[31] & aluRes[31])
                | (A[31] & B[31] & ~aluRes[31]))
                | SUB & ((~B[31] & A[31] & aluRes[31])
                | (B[31] & ~A[31] & ~aluRes[31]));
    end
  end
end
```

Single-Cycle Datapath: set flag registers

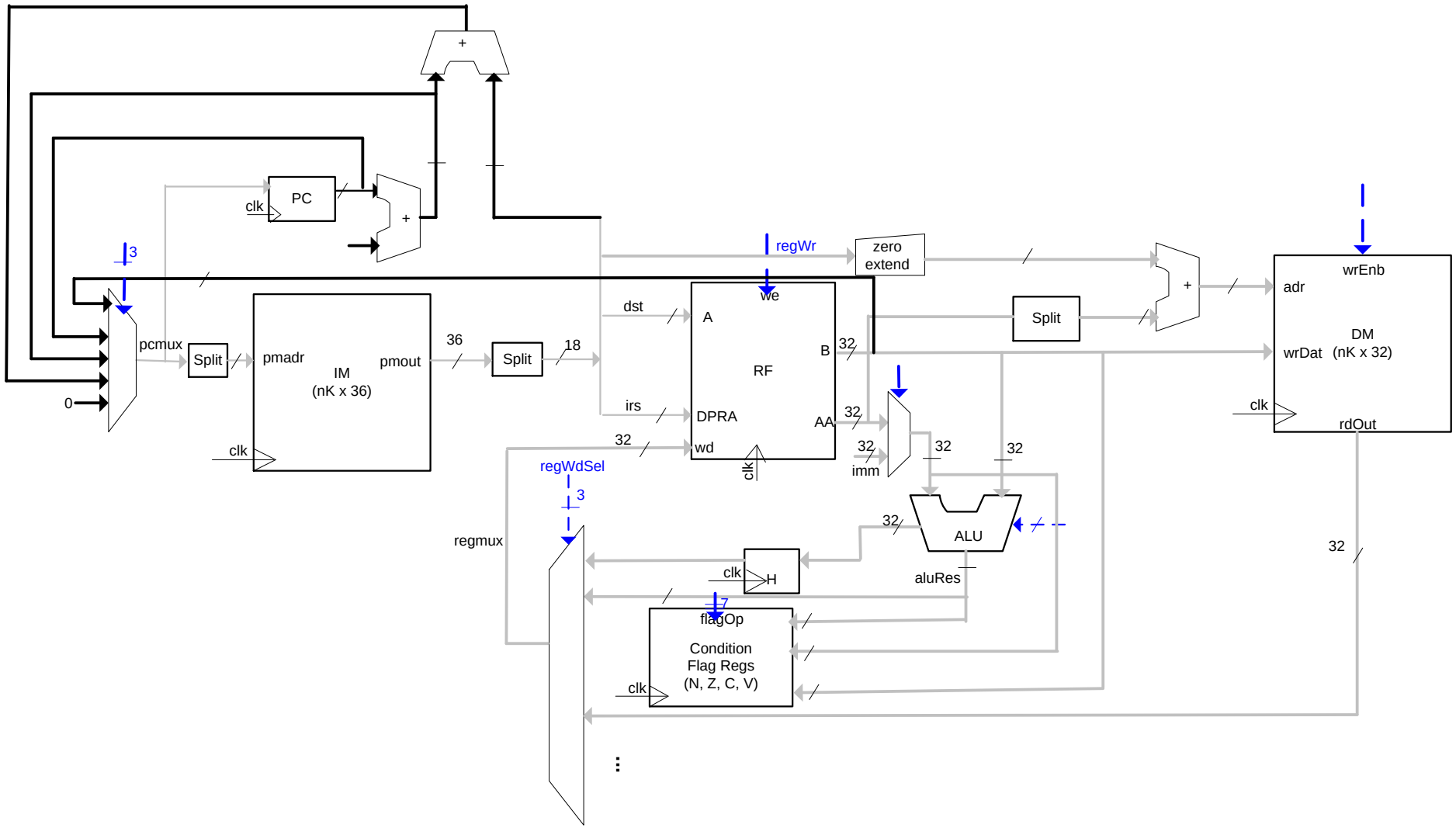


from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

Single-Cycle Datapath: Branch instructions

- Type c instructions, BR instruction, BL instruction
  - PC <= PC + 1 + off
  - PC <= Rs
  - **PC <= PC + 1 (by default)**
  - **PC <= PC (if stall)**
  - **PC <= 0 (reset)**

Single-Cycle Datapath: Branch instructions

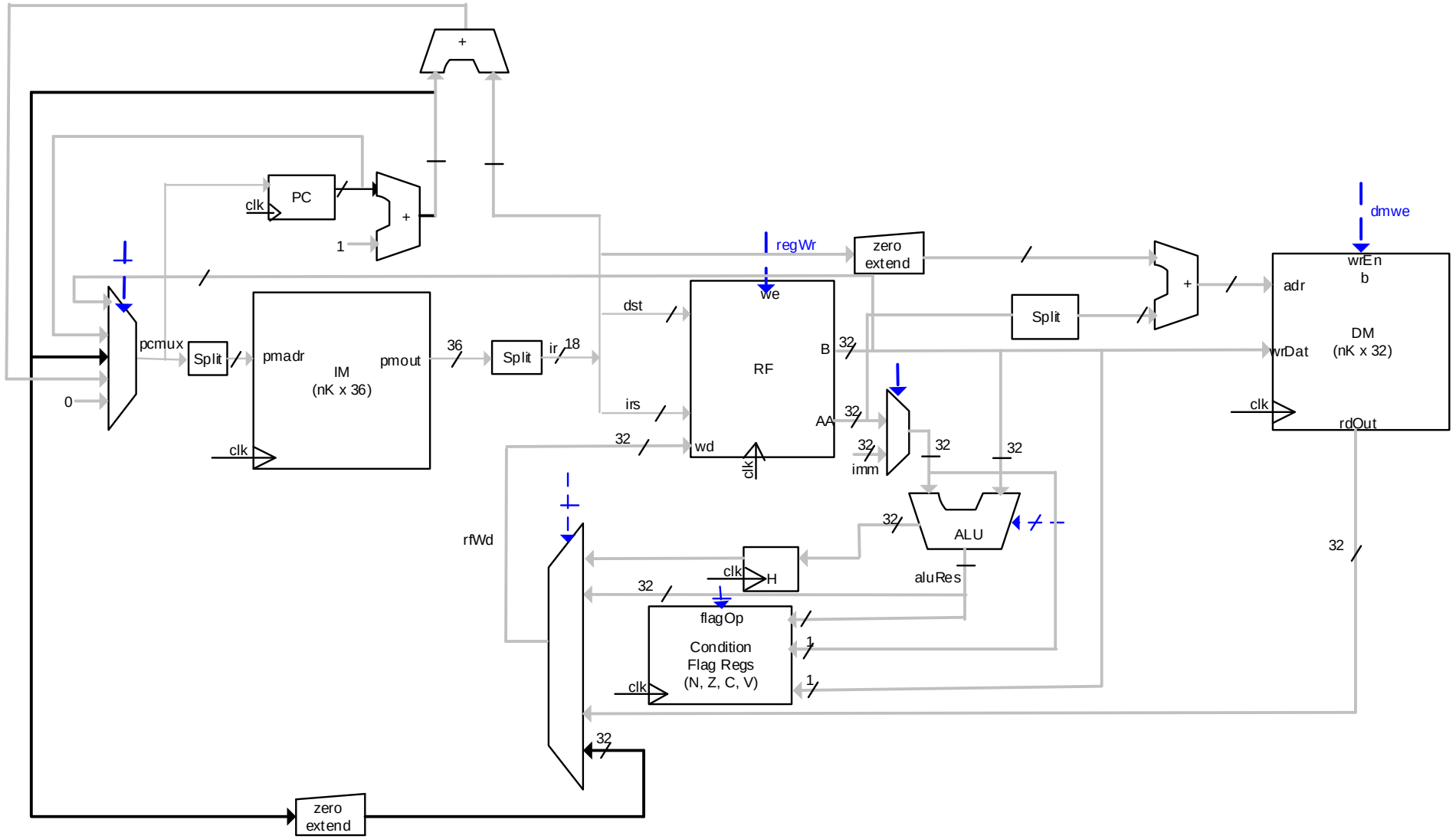


from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

Single-Cycle Datapath: Branch instructions

```
//pcmux logic
assign pcmux =
  (~rst) ? 0 :
  (stall0) ? PC:
  (BL)? {{10{IR[BLS-1]}},IR[BLS-1: 0]}+ nxpc :
  (Bc & cond) ? {{{PAW-10}{IR[9]}}, IR[9:0]} + nxpc :
  (BLR | BR ) ? A[PAW-1:0] :
  nxpc;
```

Complete single-cycle datapath



from Lectures on Reconfigurable Computing, *Dr. Ling Liu*, ETH Zürich

EXPECTATIONS FOR THE EXAM

Background

ARM

- Characterize ARM Instruction Set
- Processor Modes, Register Shadowing, Interrupts / IRQ Table

Language Support

- Loading and Linking
- The Oberon execution model: Commands and Module Loading, Module Unloading
- Object Files and consistency
- Runtime support: type inheritance and inference

Minos Case Study

- Memory Management: Classical Memory Layout
- MMU setup for Minos
- Stack- and Heap Management
- Preemptive, Rate Monotonic Scheduling. Single core scheduling with one stack
- Process Context
- (Prevention of) data races
- SPI: characterization, communication model

A2 case study

- Multi-Core boot
- APIC: idea, most important tasks, Interprocessor Interrupts
- Race conditions and their prevention (multicore): Spinlocks and beyond
- Active Oberon Compute Model: Semantics of EXCLUSIVE / AWAIT (Egg-Shell Modell)
- Stack Management
- Activity Management (A2): process states, data structures
- Context-Switches (Synchronous / Asynchronous) [no IRQ-trick details]

Lock-Free Kernel

- Spinlocks, CAS, Contention and Backoff
- Lock-free algorithms (counter, stack)
- ABA Problem
- Hazard-Pointers
- Implicit cooperative Multitasking → Processor local storage
- Task Switch Finalizers

Case Study 3: RISC / Oberon

- Pros & Cons of building from scratch
- Processor construction: instruction set (and stalls)
- Measuring hardware speed
- Characterize the Oberon system (User Interface / Core Structure / Programming Model)
- Memory mapped registers

Case Study 4: Active Cells

- Active Cells Programming Model: Idea, Semantics
- Hybrid compilation, Implementation (ideas)
- TRM
- Fast Path
- Axi-4 Stream interconnects
- Extensible Hardware, Engines (very qualitative)

Typical Exam Questions

Entry Questions

- How does a multiprocessor system boot?
- Typical memory layout of a one-/multi-processor system (no heavyweight processes). Unmapped pages...
- Specialities of ARM instruction set / the ARM architecture
- What is GPIO?
- What happens when a command is activated in Oberon
- What does module loading and module unloading mean and imply?

Progressing towards...

- How do you implement a low-level lock? On a single-core system, on a multicore-system?
- How to implement a lock-free stack? What is the ABA problem? How to solve it?
- Difference between static- and dynamic loading
- Remember why the first page of the system was unmapped? Pitfalls?
- Describe a simple scheduler with periodic and aperiodic tasks (etc.)
- Sketch the life cycles of processes in A2
- Stack allocation in A2. How could a dynamic stack be implemented?
- Advantages and disadvantages of 18-bit instructions of TRM
- Advantages and disadvantages of building hardware from scratch
- Hybrid compilation: what? How?
- Why patch code / data files to bit stream? Alternatives?

A selection of possible exam questions will be available through the repository

THE END

I will be available for further questions via email

felix.friedrich@inf.ethz.ch

Do not hesitate to ask!