# Reconfigurable Computing

## Parallelism

*Indeed. RISC architecture is gonna change everything*

\- Angelina Jolie as Kate in Hackers

Philip Leong (philip.leong@sydney.edu.au)
School of Electrical and Information Engineering

http://phwl.org/talks

Permission to use figures have been gained where possible. Please contact me if you believe anything within infringes on copyright.

THE UNIVERSITY OF SYDNEY

# Introduction

- Parallelism is the key to achieving high performance
- We will cover two key techniques
  - Spatial parallelism (in 1 slide)
  - Pipelining
- Case study
  - RISC processor

# Parallelism

# Sequential Laundry

INPUT:
dirty laundry

OUTPUT:
6 more weeks

Device: Washer

Function: Fill, Agitate, Spin

$Washer_{PD}$ = 30 mins
$= T_W$

Device: Dryer

Function: Heat, Spin

$Dryer_{PD}$ = 60 mins
$= T_D$

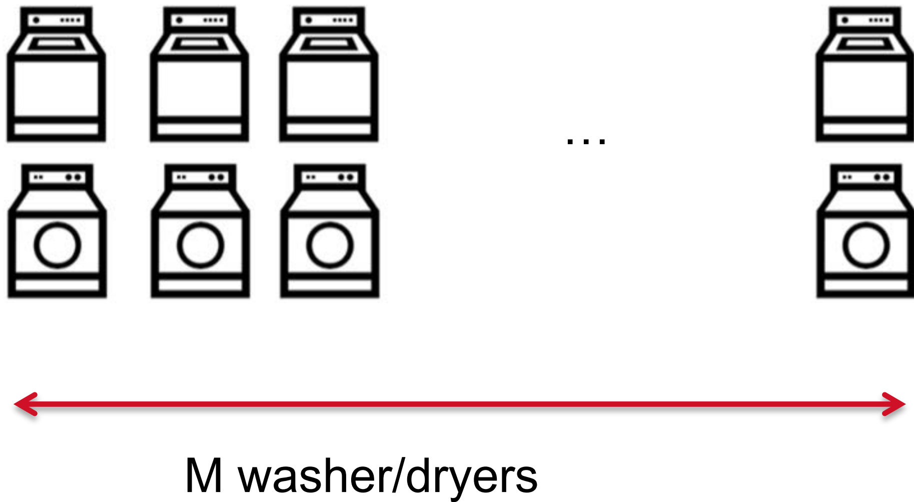
Figure by MIT OpenCourseWare.

2009 3/3/09 L08 - Pi

- Suppose we have N loads to process
- $Time = N(T_W + T_D)$

THE UNIVERSITY OF SYDNEY

# Spatially Parallel Laundry




- If we had M washer dryers (and no other overheads)
- Time=(N/M)(Washer+Dryer)
- But cost of equipment is also M times higher
- (no dependencies i.e. all the loads can be operated in parallel)

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Pipelined Laundry

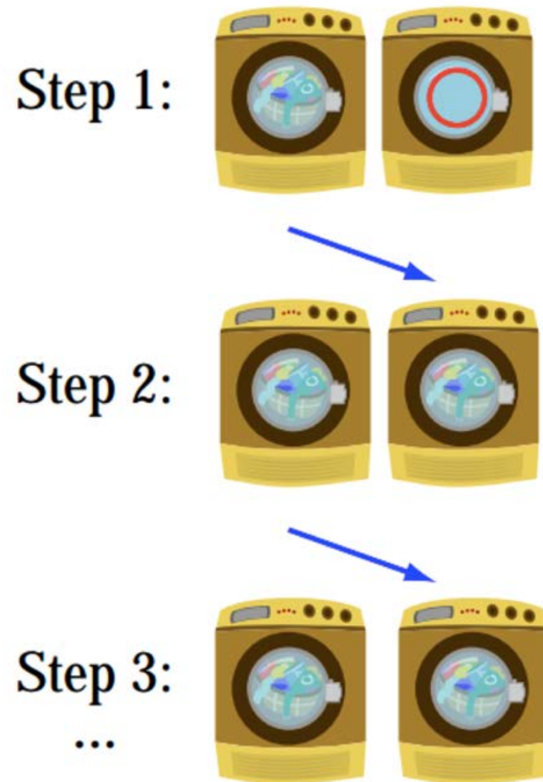


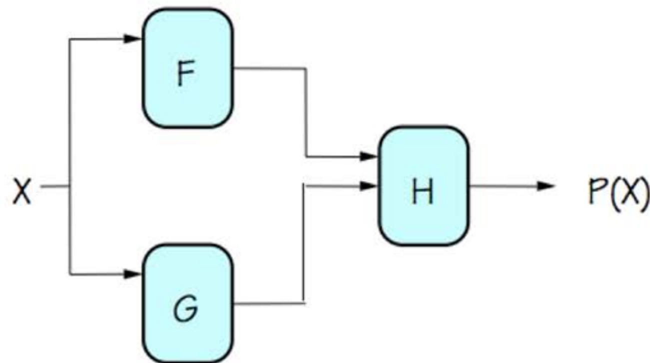
Figure by MIT OpenCourseware.

- We can dry the first batch while we wash the second
- Latency (time from when we supply the input to when the output is available)? $T_W+T_D$ (no change over seq)
- Throughput (the rate at which inputs or outputs are processed)? $max(T_W, T_D)$!
- Time ≈ N $max(T_W, T_D)$
- In general case for P stage pipeline to perform a task that takes time T
- Speedup=NT/((P+N-1)T/P)

  =NP/(P+N-1)
- As N becomes big, Speedup≈P (assuming that the cycle time is reduced from T to T/P)

# Pipelining

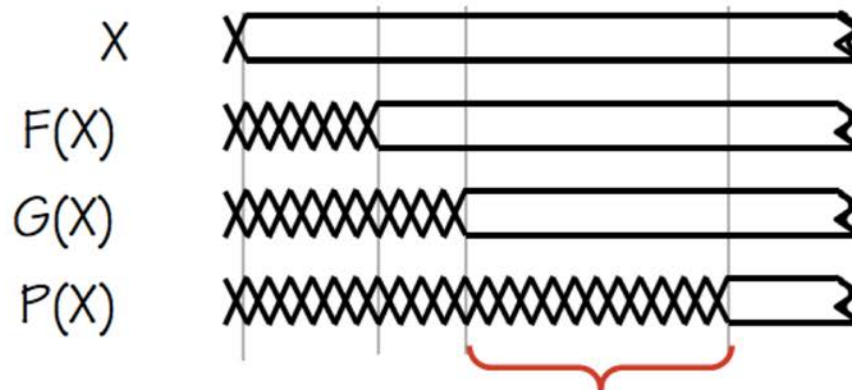
# Pipelining of Combinational Logic




For combinational logic:

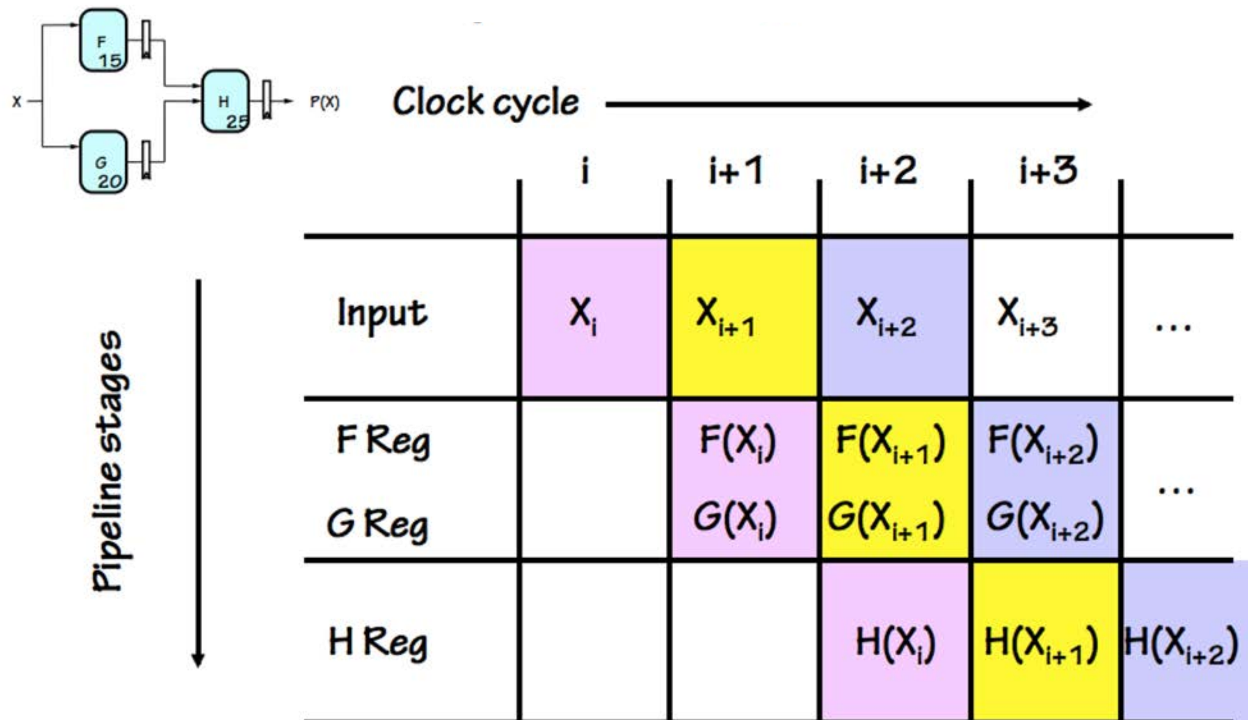
latency = $t_{PD}$,
throughput = $1/t_{PD}$.

We can't get the answer faster, but are we making effective use of our hardware at all times?



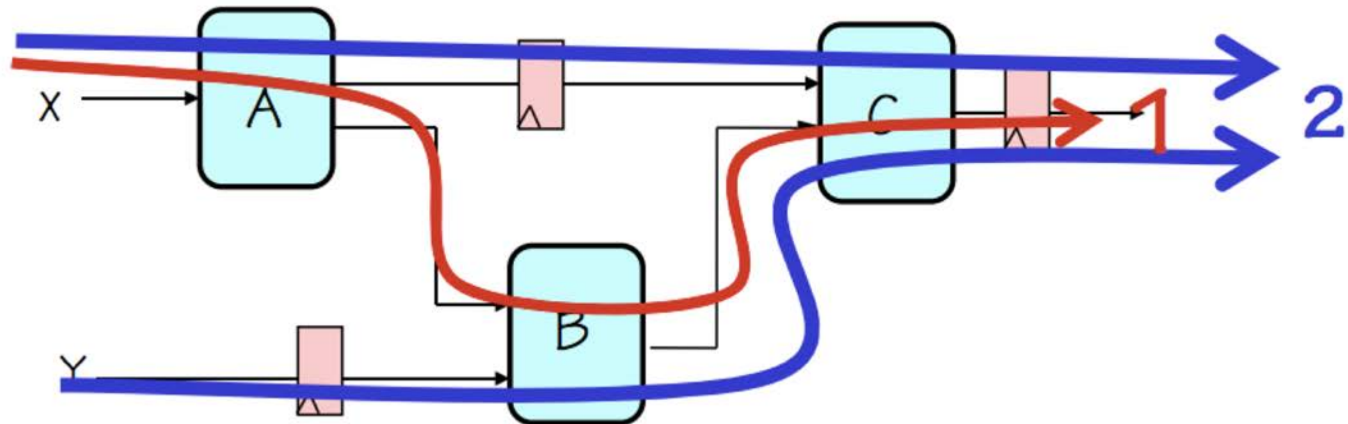

F & G are "idle", just holding their outputs stable while H performs its computation

# Pipeline Diagrams



# Ill-formed Pipelines

Consider a BAD job of pipelining:




For what value of K is the following circuit a K-Pipeline? ANS: none

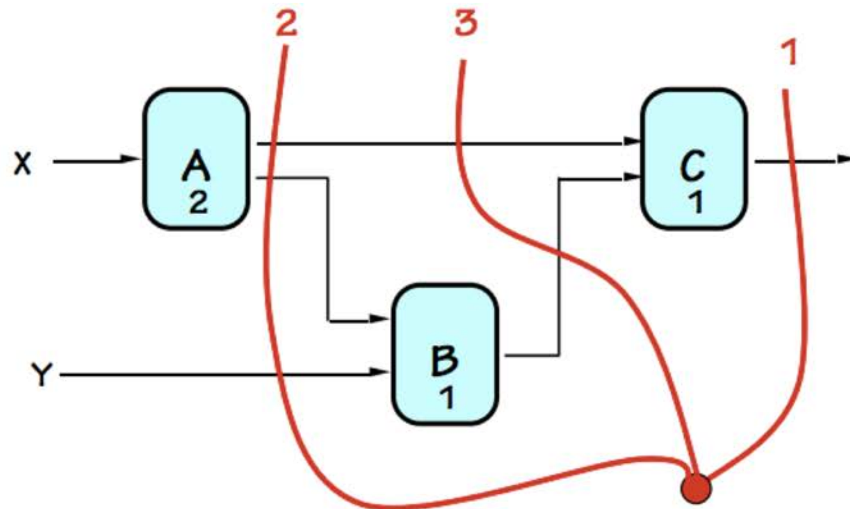
Problem:

***Successive inputs get mixed***: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

**This CAN'T HAPPEN on a well-formed K pipeline!**

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Pipeline Example



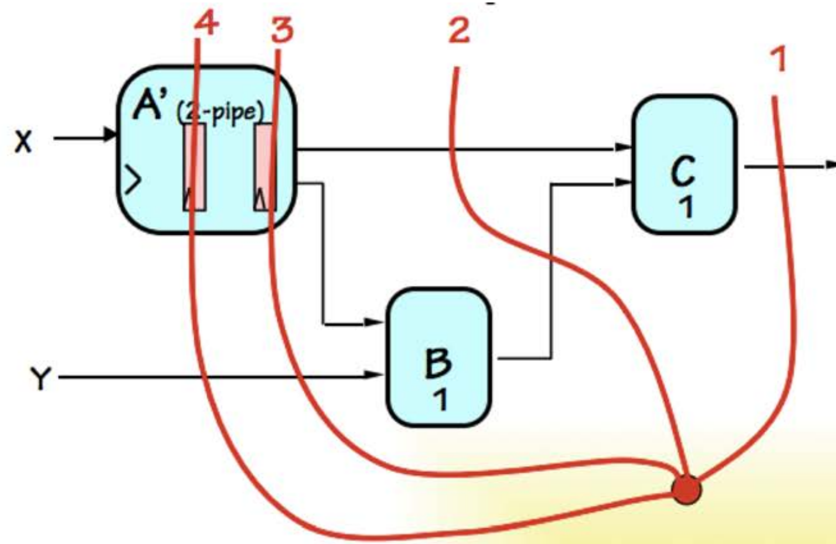

| | LATENCY | THROUGHPUT |
|---|---|---|
| 0-pipe: | 4 | 1/4 |
| 1-pipe: | 4 | 1/4 |
| 2-pipe: | 4 | 1/2 |
| 3-pipe: | 6 | 1/2 |

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Pipelined Components




4-stage pipeline, thruput=1

Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may increase clock frequency
- Must account for new pipeline stages in our plan

but... but...
How can I pipeline
*a clothes dryer???*

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

THE UNIVERSITY OF SYDNEY

# Back to our bottleneck

Recall our earlier example...

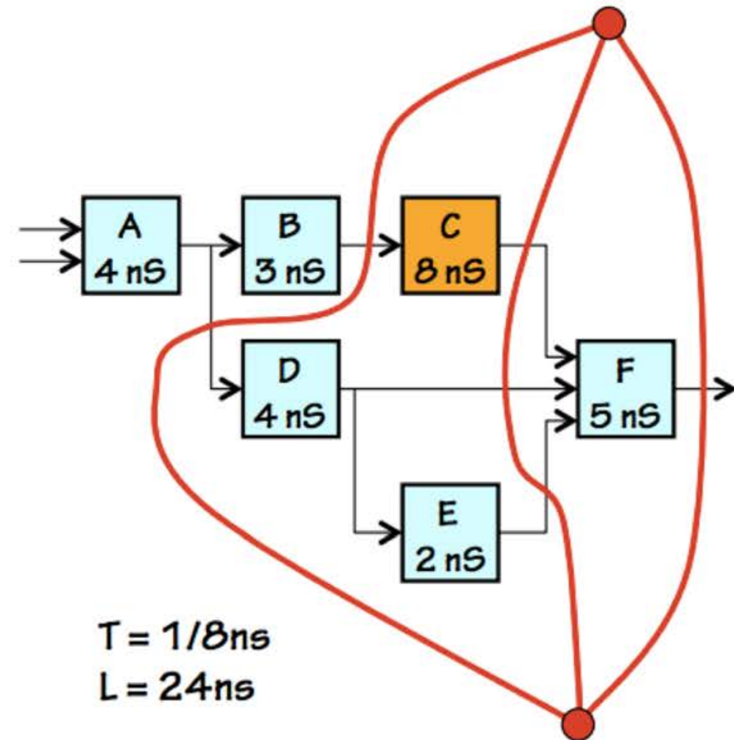
- C – the slowest component – limits clock period to 8 ns.
- HENCE throughput limited to 1/8ns.

We could improve throughput by

- Finding a pipelined version of C;

OR...

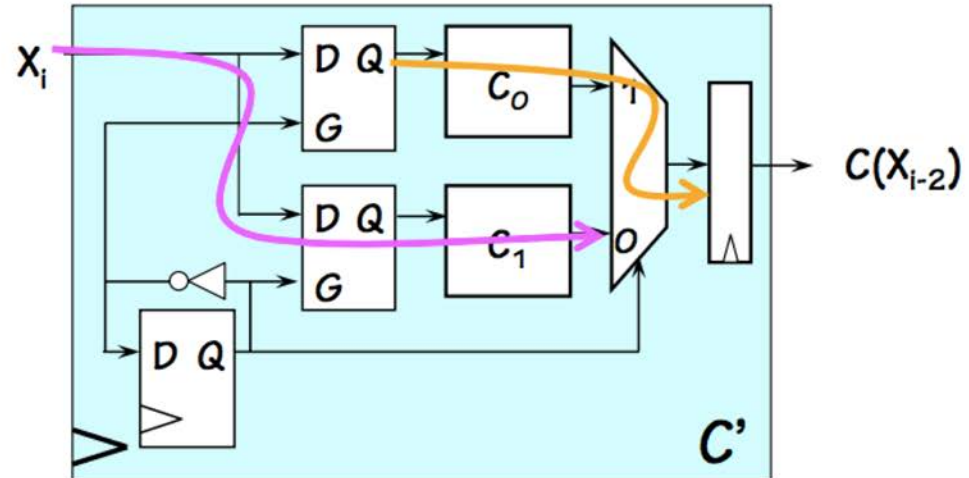
- *interleaving* multiple copies of C!



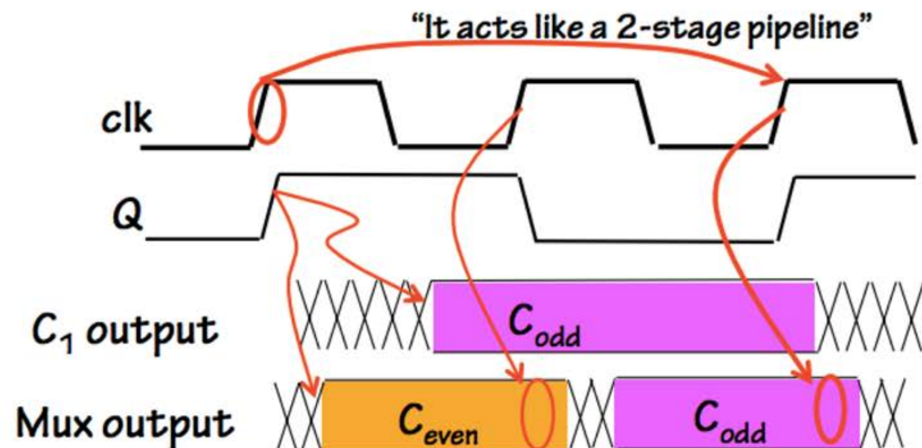

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Circuit Interleaving (Pipelining + Spatial Parallelism)

We can simulate a pipelined version of a slow component by replicating the critical element and alternate inputs between the various copies.




When Q is 1 the lower path is combinational (the latch is open), yet the output of the upper path will be enabled onto the input of the output register ready for the NEXT clock edge.



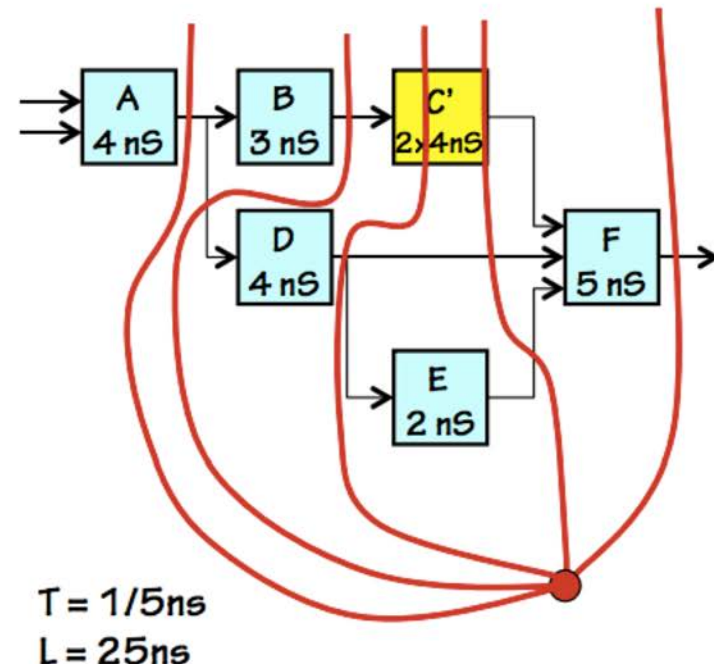

Meanwhile, the other latch maintains the input from the last clock.

# Combining Interleaving and Pipelining

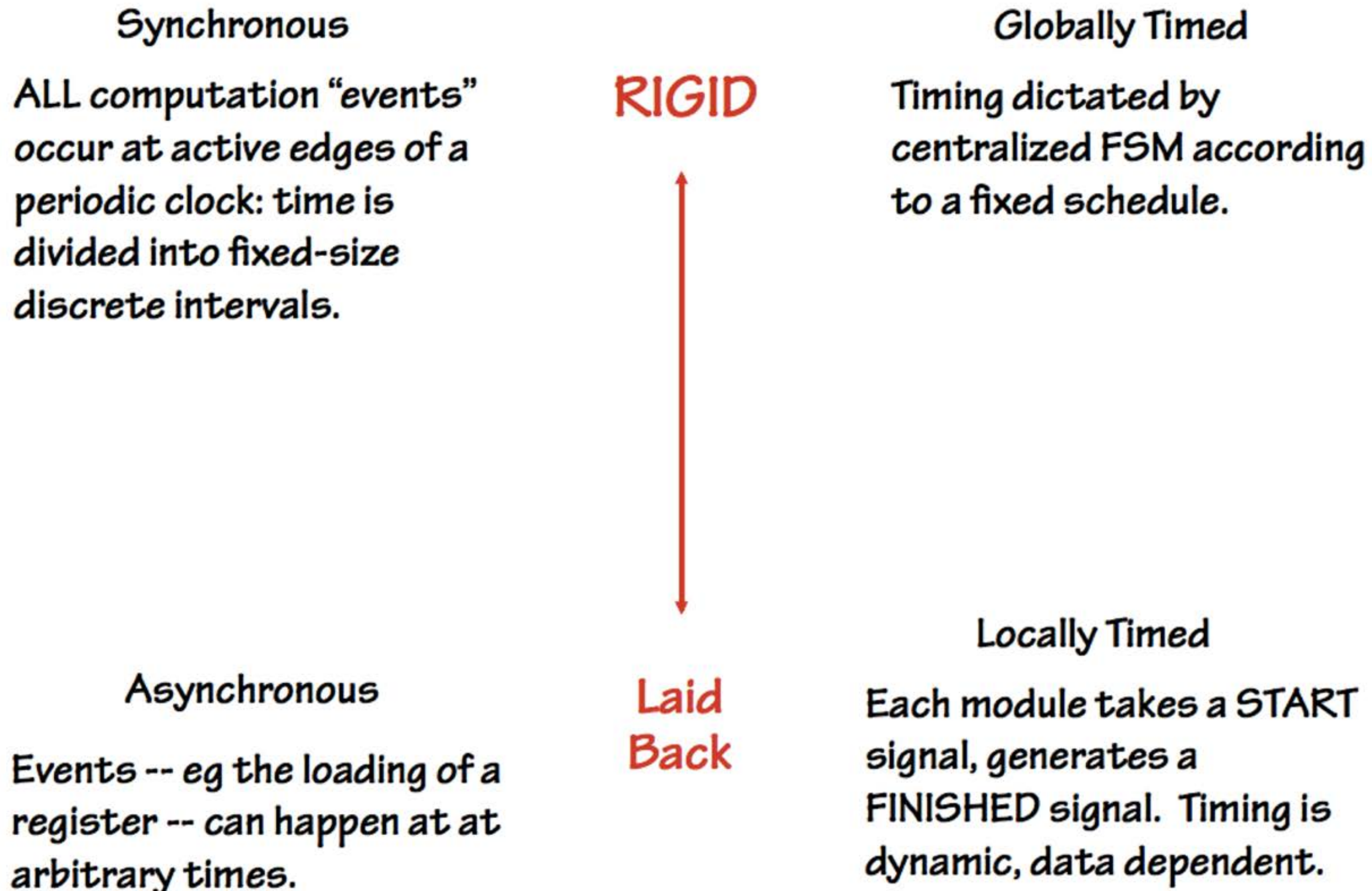
We can combine interleaving and pipelining. Here, C' interleaves two C elements with a propagation delay of 8 nS.

The resulting C' circuit has a throughput of 1/4 nS, and latency of 8 nS. This can be considered as an extra pipelining stage that passes through the middle of the C' module. One of our separation lines must pass through this pipeline stage.

By combining interleaving with pipelining we move the bottleneck from the C element to the F element.

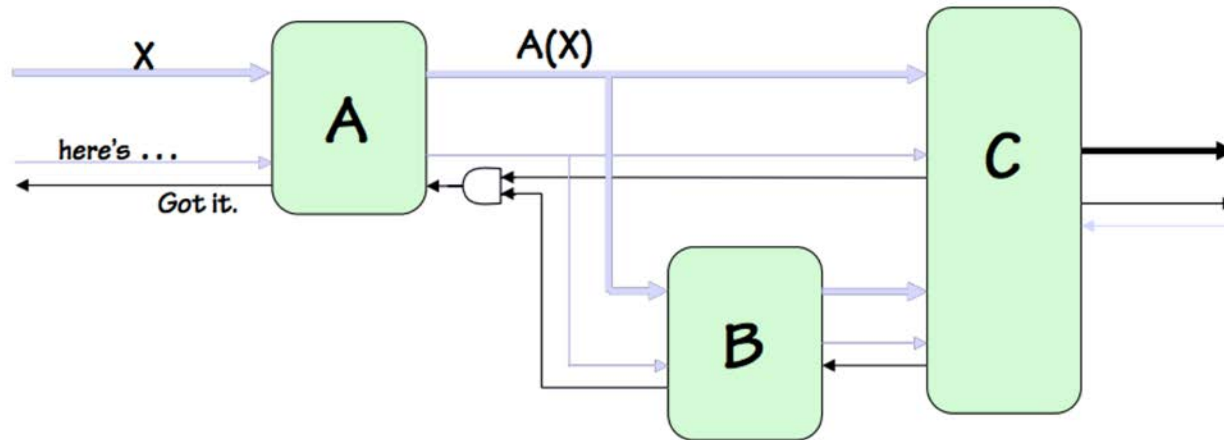


# Control Structures



# Self-timed Example

a glimpse of an asynchronous, locally-time discipline




Elegant, timing-independent design:

- Each component specifies its own time constraints
- Local adaptation to special cases (eg, multiplication by 0)
- Module performance improvements automatically exploited
- Can be made asynchronous (no clock at all!) or synchronous

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Summary

- Latency (L) = time it takes for given input to arrive at output
- Throughput (T) = rate at each new outputs appear
- For combinational circuits: $L = t_{PD}$ of circuit, $T = 1/L$
- For K-pipelines (K > 0):
  - always have register on output(s)
  - K registers on *every* path from input to output
  - Inputs available shortly after clock i, outputs available shortly after clock (i+K)
  - $T = 1/(t_{PD,REG} + t_{PD}$ of slowest pipeline stage $+ t_{SETUP})$
    - more throughput → split slowest pipeline stage(s)
    - use replication/interleaving if no further splits possible
  - $L = K / T$
    - pipelined latency ≥ combinational latency

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Other Techniques

- Techniques not discussed in this lecture but often used in reconfigurable computing
  - Vectorization
  - Systolic arrays
  - Task-level parallelism (particularly MPI)

# Single Cycle Processor - Datapath

# Introduction

- Example of a design based on datapath+control with synchronous control
- Discuss the design of a reduced instruction set (RISC) processor for the MIPS instruction set
- The design methodology for any field-programmable custom computing machine (FCCM) is similar
  - Those without programmability are a special case of processors
- The are several different design methodologies, we cover single cycle first and then pipelined

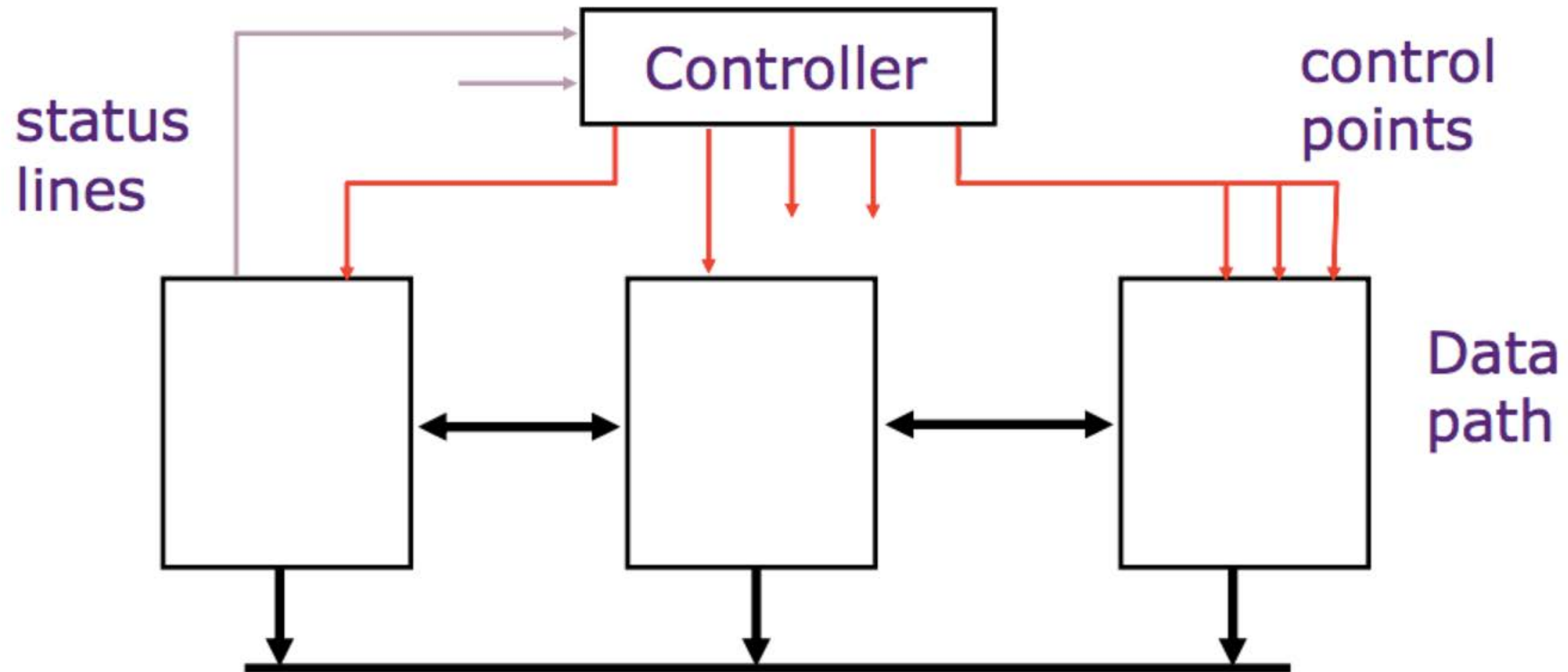
# Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions/Program function of source code, compiler and instruction set architecture (ISA)
- Cycles/Instruction (CPI) function of ISA and microarchitecture
- Time/Cycle depends on microarchitecture and the VLSI technology used

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Microarchitecture

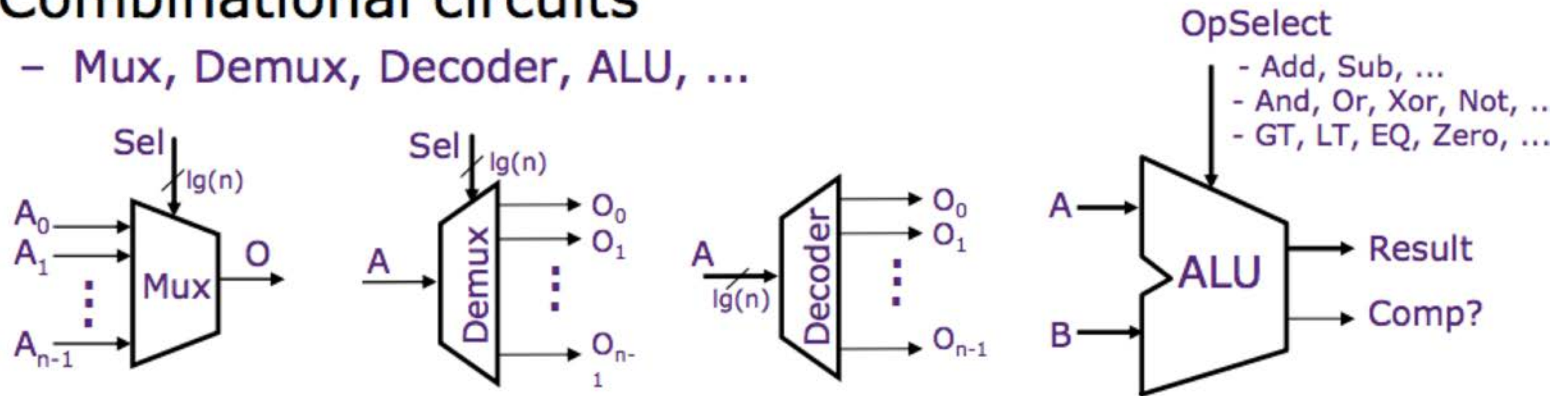



- Structure: How components are connected (static)
- Behaviour: How data moves between components (dynamic)

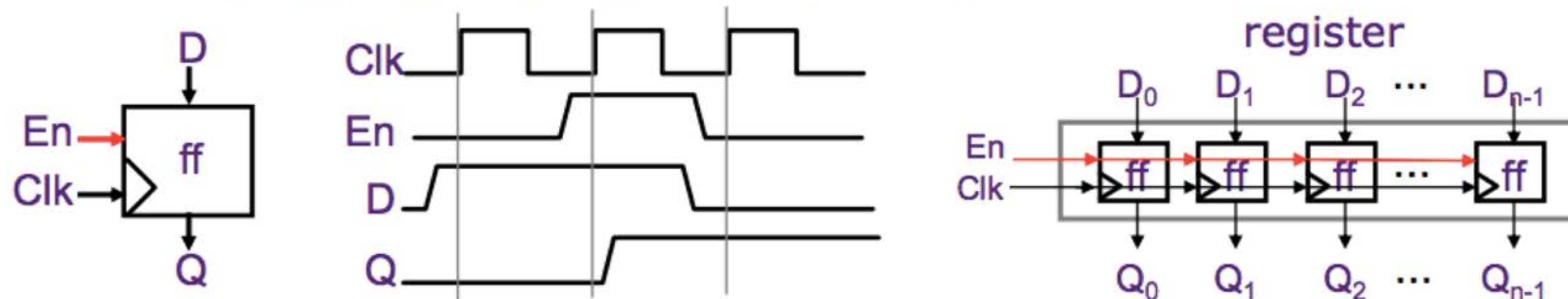
Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Building Blocks

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



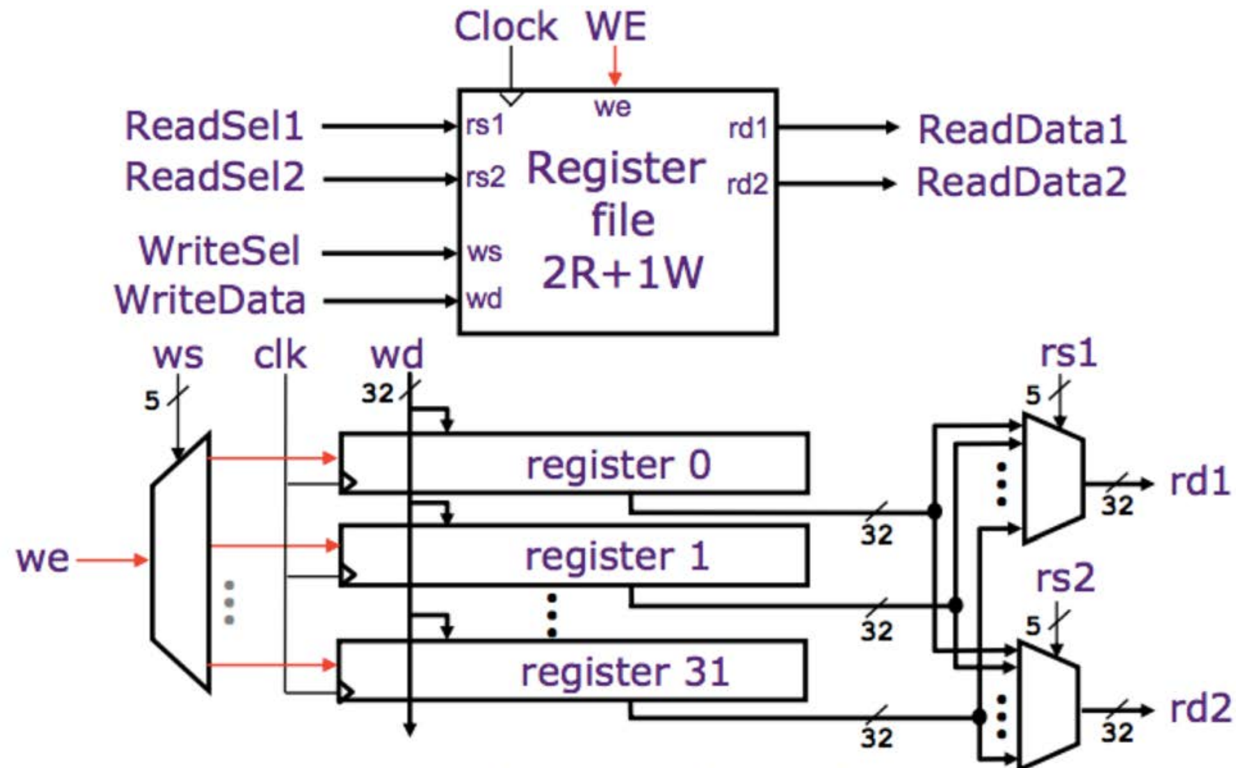

- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM




Edge-triggered: Data is sampled at the rising edge

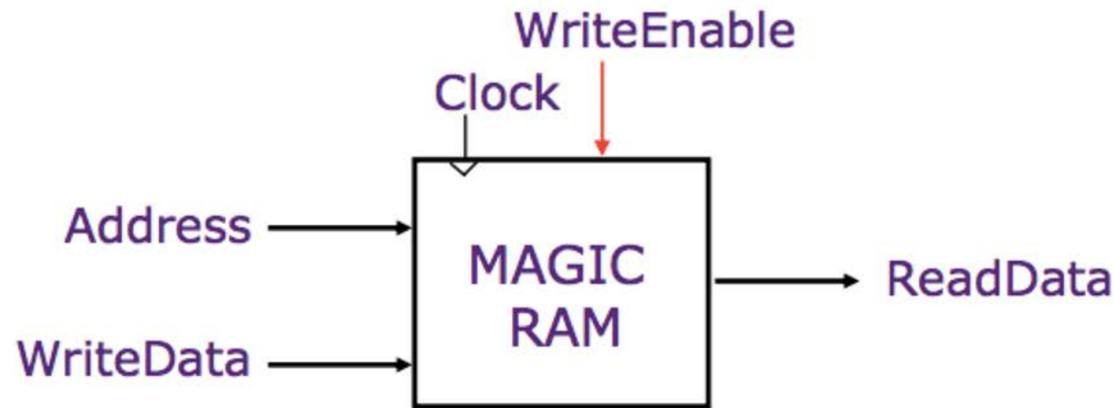
Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Register Files




- No timing issues in reading a selected register
- Register files with a large number of ports are difficult to design
  - Intel's Itanium, GPR File has 128 registers with 8 read ports and 4 write ports!!!

# Simplified Memory




Reads and writes are always completed in one cycle

- a Read can be done any time (i.e. combinational)
- a Write is performed at the rising clock edge if it is enabled

$\Rightarrow$ the write address and data must be stable at the clock edge

# Single Cycle RISC Processor

# MIPS ISA

- Processor State
  - 32 32-bit GPRs, R0 always contains a 0
  - 32 single precision FPRs, may also be viewed as 16 double precision FPRs, FP status register, used for FP compares & exceptions
  - PC, the program counter
  - some other special registers
- Data types
  - 8-bit byte, 16-bit half word
  - 32-bit word for integers
  - 32-bit word for single precision floating point, 64-bit word for double precision floating point
- Load/Store style instruction set
  - data addressing modes- immediate & indexed
  - branch addressing modes- PC relative & register indirect
  - Byte addressable memory- big endian mode
- All instructions are 32 bits

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# MIPS Instruction Set

| Format | | | | | | |
|---|---|---|---|---|---|---|
| R-type: | op | rs | rt | rd | | func |
| I-type: | op | rs | rt | immediate16 | | |
| J-type: | op | immediate26 | | | | |

| | | source(s) | destination |
|---|---|---|---|
| ALU | rd ← (rs) func (rt) | rs, rt | rd |
| ALUi | rt ← (rs) op imm | rs | rt |
| LW | rt ← M [(rs) + imm] | rs | rt |
| SW | M [(rs) + imm] ← (rt) | rs, rt | |
| BZ | cond (rs) | | |
| | true: PC ← (PC) + imm | rs | |
| | false: PC ← (PC) + 4 | rs | |
| J | PC ← (PC) + imm | | |
| JAL | r31 ← (PC), PC ← (PC) + imm | | 31 |
| JR | PC ← (rs) | rs | |
| JALR | r31 ← (PC), PC ← (rs) | rs | 31 |

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Instruction Execution

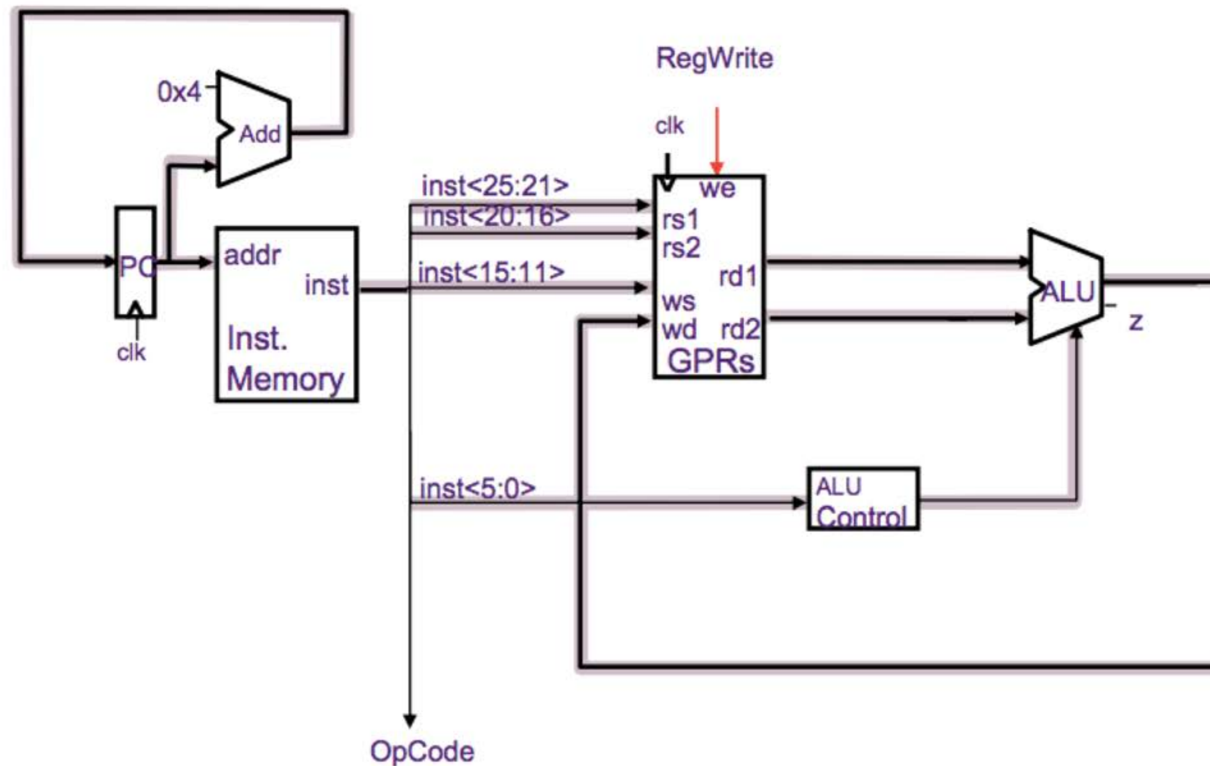
› Execution of an instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back

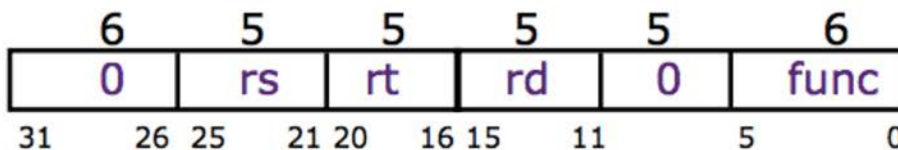
and the computation of the address of the next instruction

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Datapath: Reg-Reg Instructions



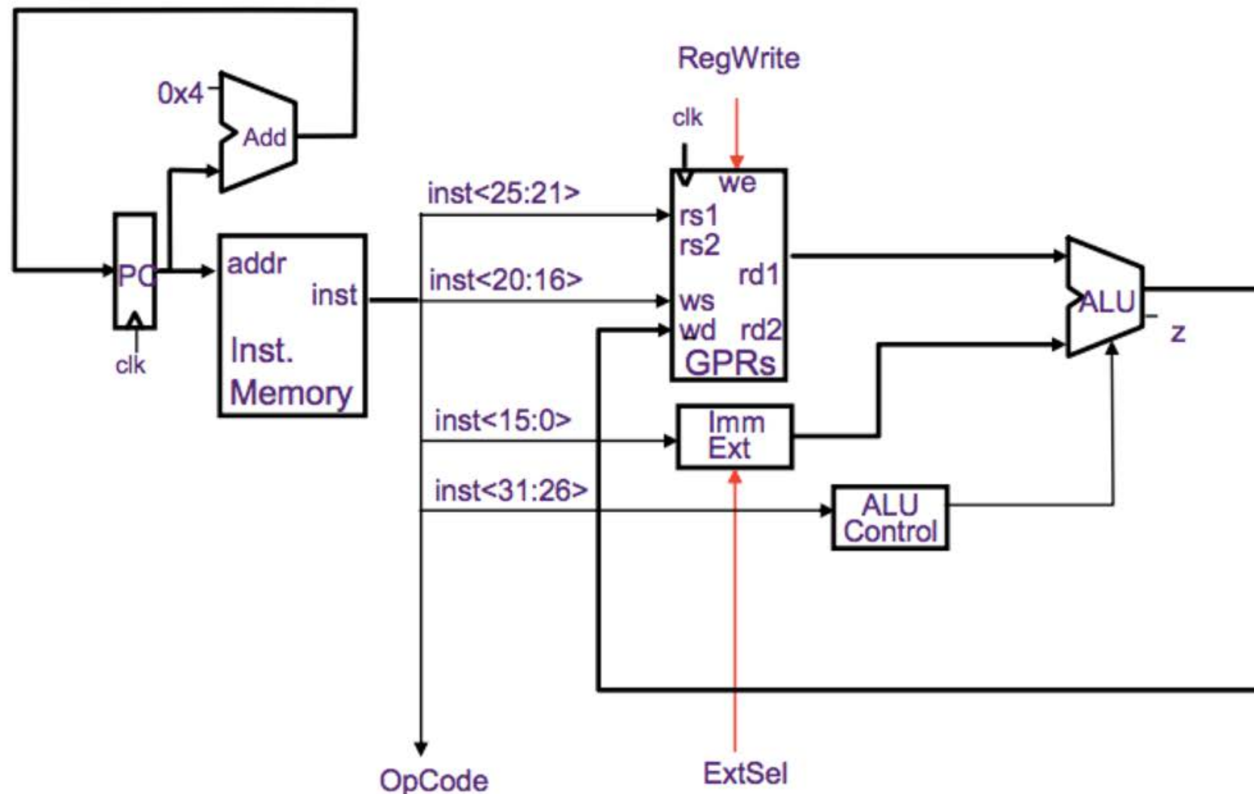

*RegWrite Timing?*

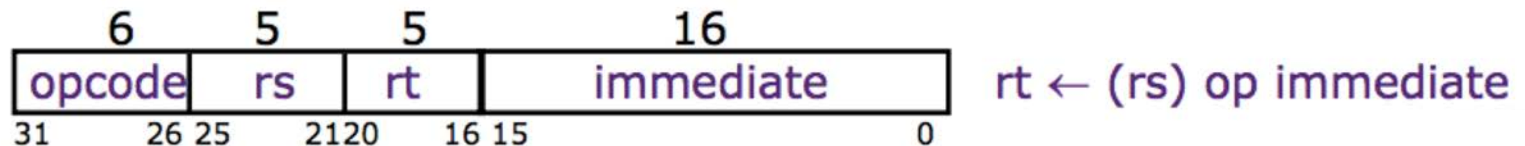



rd ← (rs) func (rt)

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

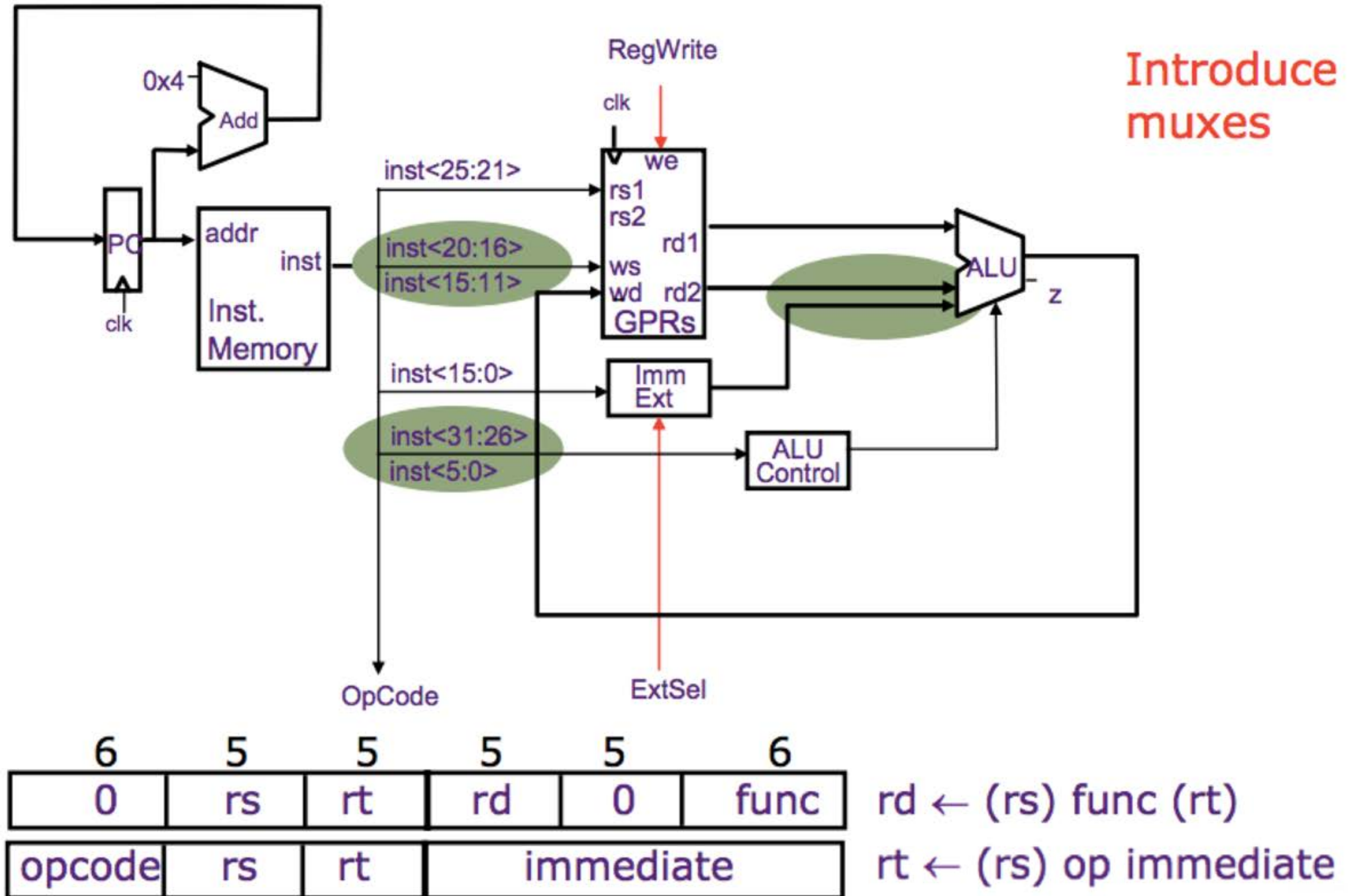
# Datapath: Reg-Imm Instructions



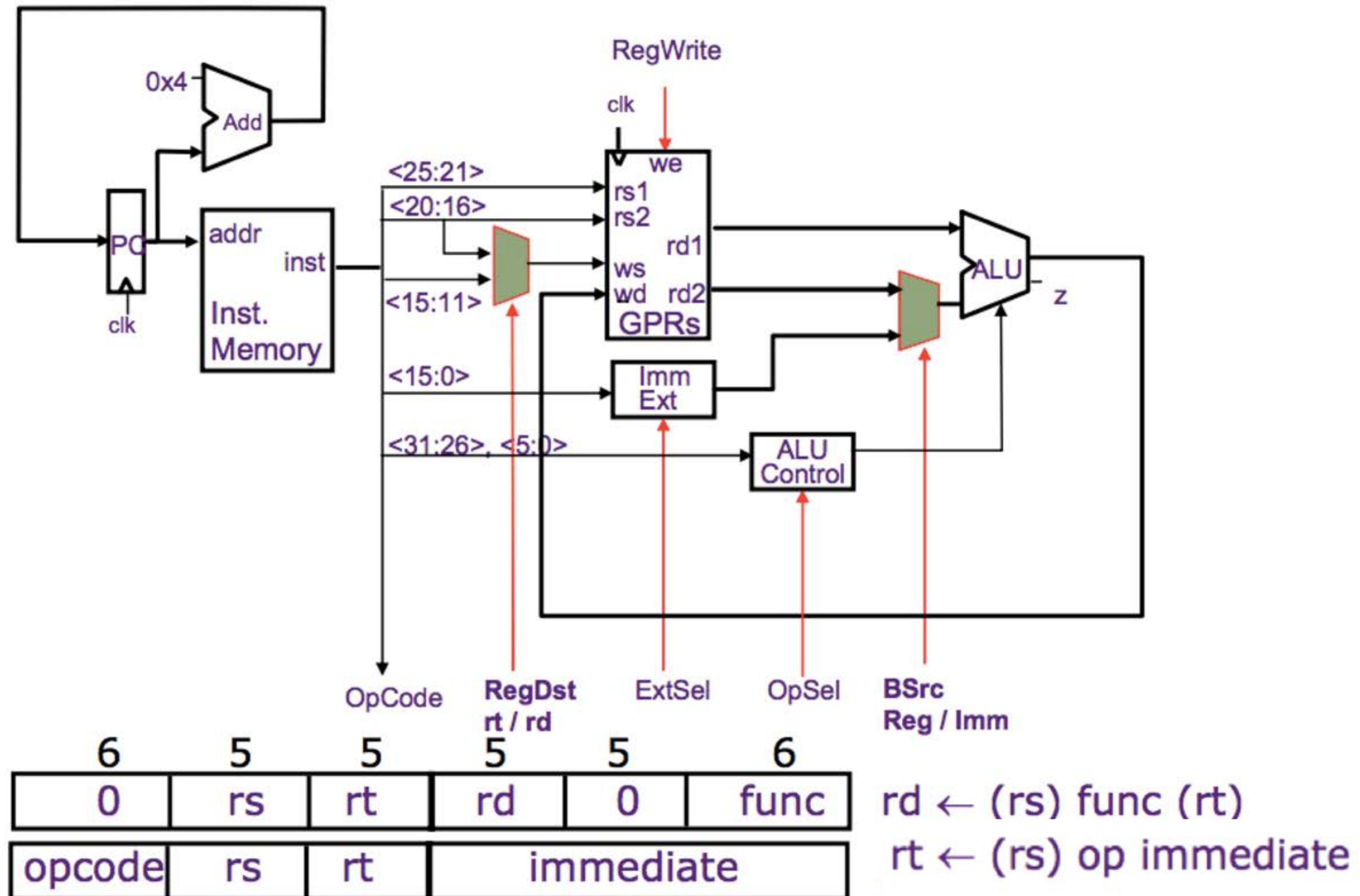

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Conflicts in Merging Datapath




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Datapath for ALU Instructions




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Datapath for Memory Instructions

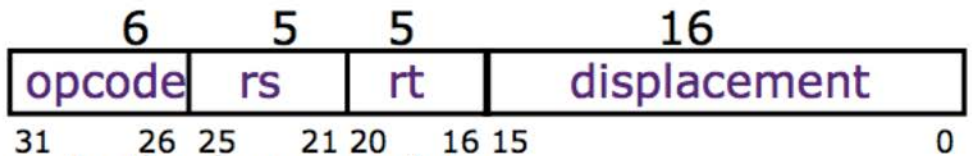
- Should program and data memory be separate?
- Harvard style: separate (Aiken and Mark 1 influence)
  - read-only program memory
  - read/write data memory
  - at some level the two memories have to be the same
- Princeton style: the same (von Neumann's influence)
  - A Load or Store instruction requires accessing the memory more than once during its execution

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

THE UNIVERSITY OF SYDNEY

# Load/Store Instructions: Harvard Architecture

0x4
Add
PC
clk
addr
inst
Inst.
Memory
"base"
disp
RegWrite
clk
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
Imm
Ext
ALU
Control
ALU
z
MemWrite
clk
we
addr
rdata
Data
Memory
wdata
WBSrc
ALU / Mem
OpCode
RegDst
ExtSel
OpSel
BSrc



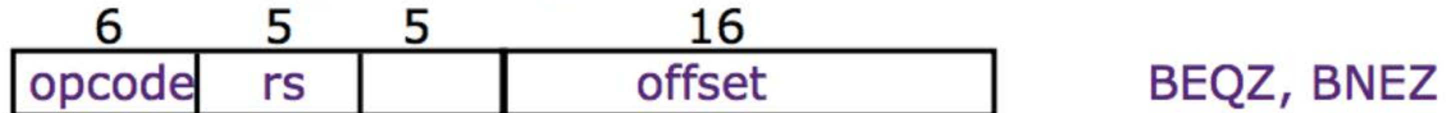

addressing mode
(rs) + displacement

rs is the base register
rt is the destination of a Load or the source for a Store

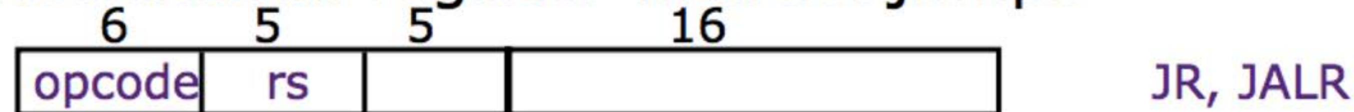
Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# MIPS Control Instructions

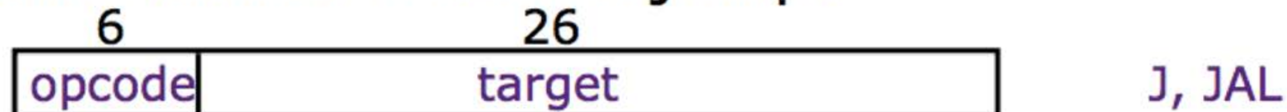
Conditional (on GPR) PC-relative branch




Unconditional register-indirect jumps



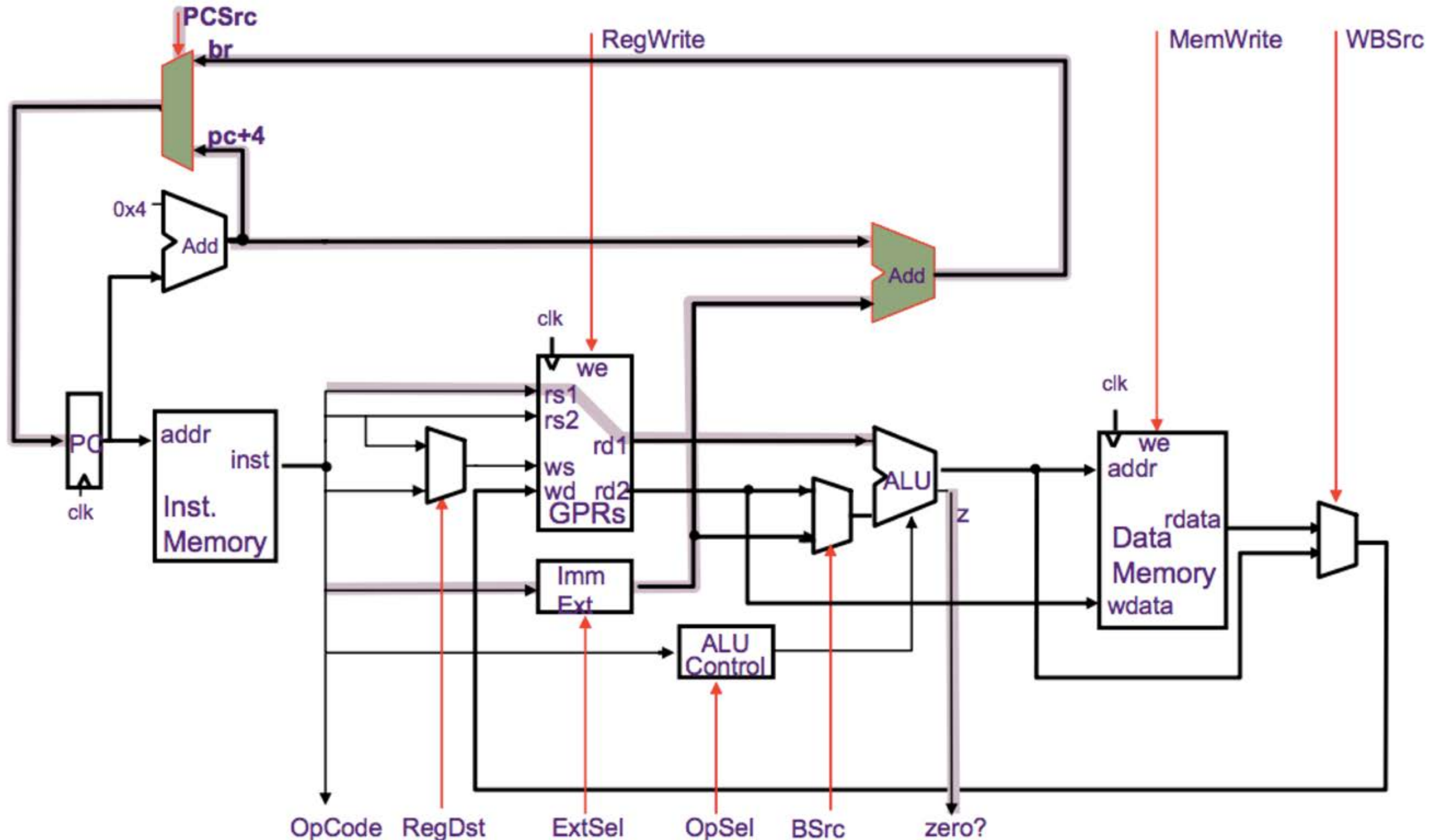

Unconditional absolute jumps




- PC-relative branches add offset×4 to PC+4 to calculate the target address (offset is in words): ±128 KB range
- Absolute jumps append target×4 to PC<31:28> to calculate the target address: 256 MB range
- jump-&-link stores PC+4 into the link register (R31)
- All Control Transfers are delayed by 1 instruction
  we will worry about the branch delay slot later

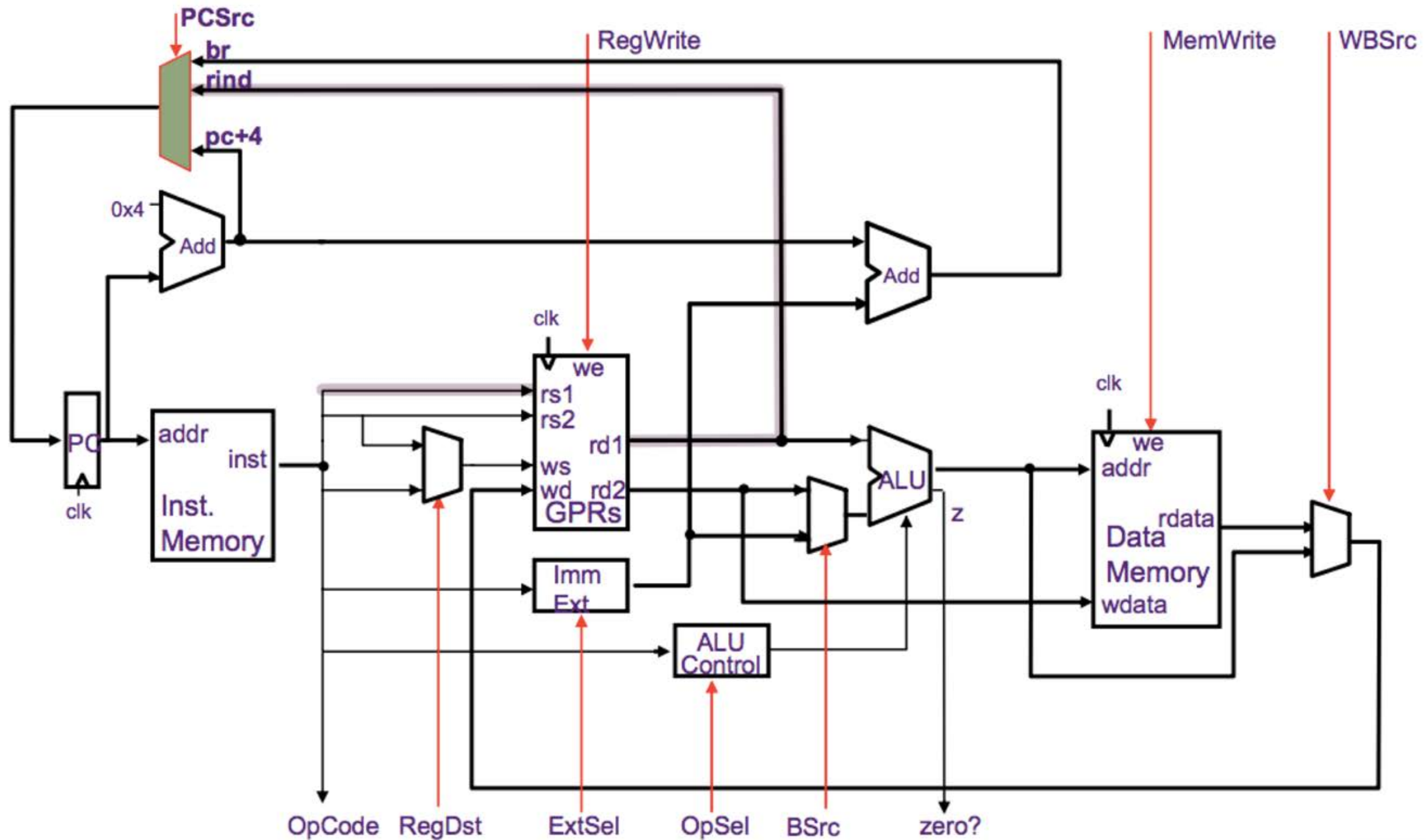
Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Conditional Branches (BEQZ, BNEZ)



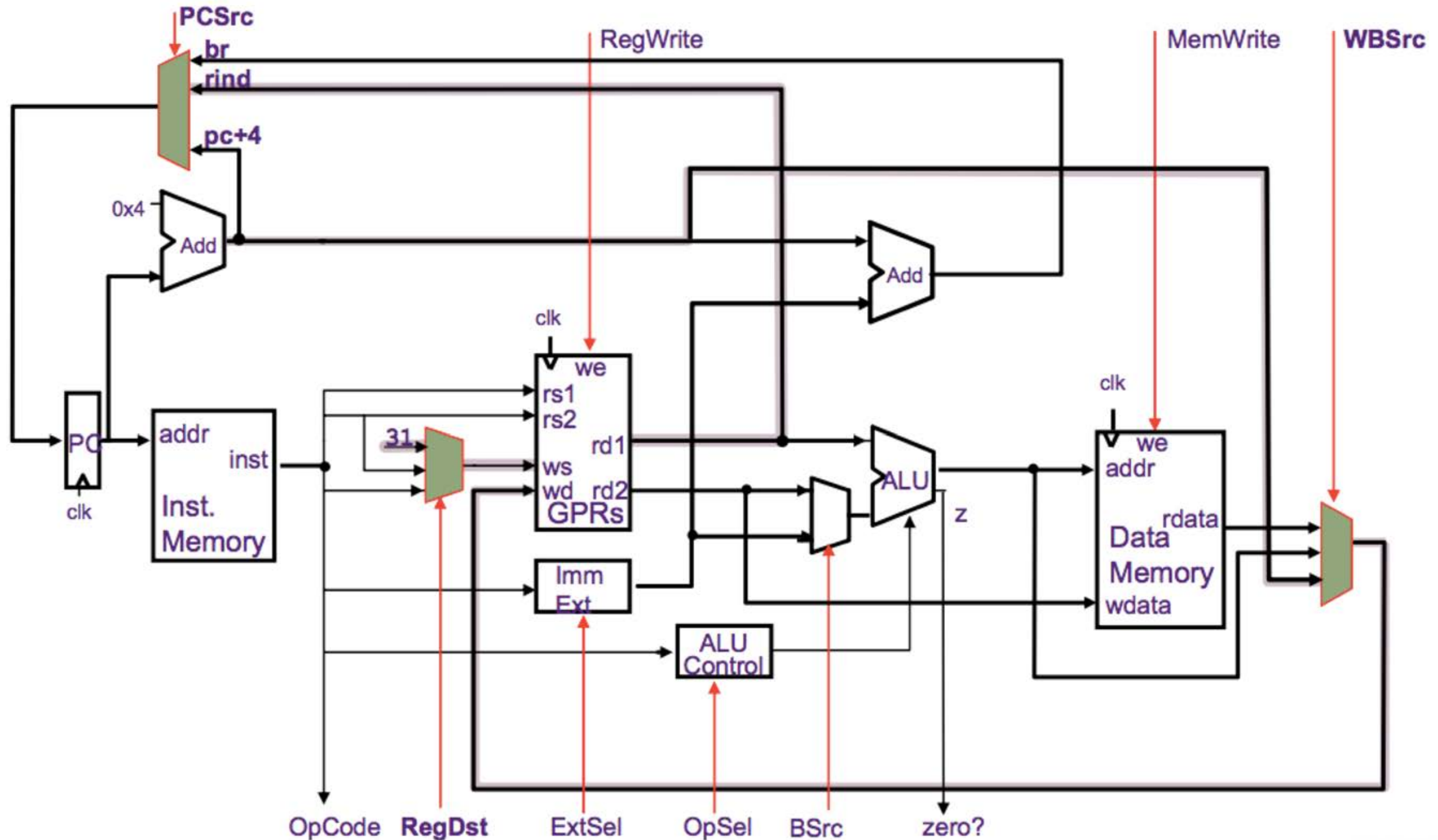

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Register-Indirect Jumps



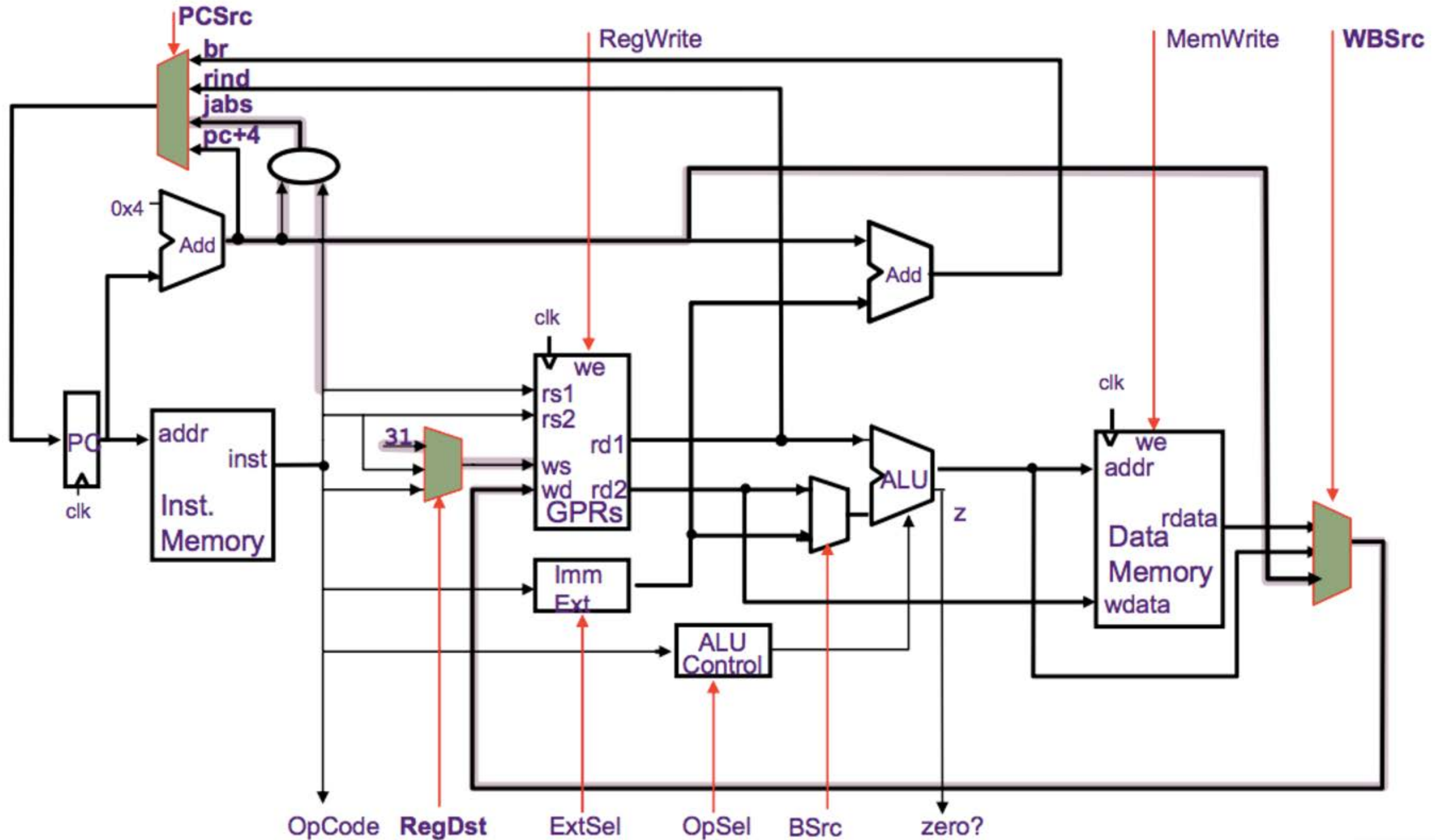

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Register-Indirect Jump-&-Link (JALR)



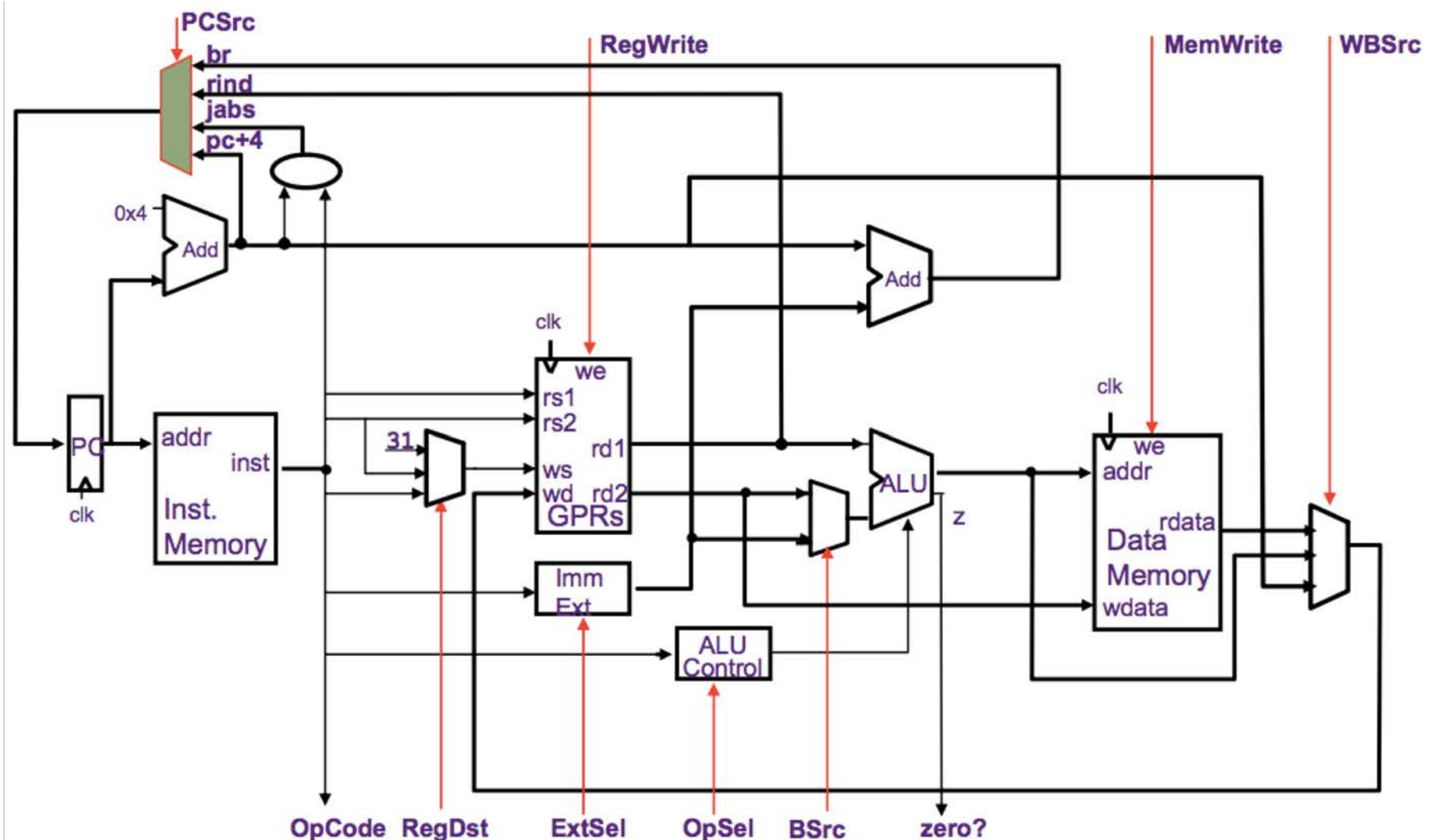

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Absolute Jumps (J, JAL)




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Harvard-Style Datapath for MIPS




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Control for Single Cycle RISC Processor

THE UNIVERSITY OF SYDNEY

# Single-Cycle Hardwired Control: Harvard architecture

› We will assume

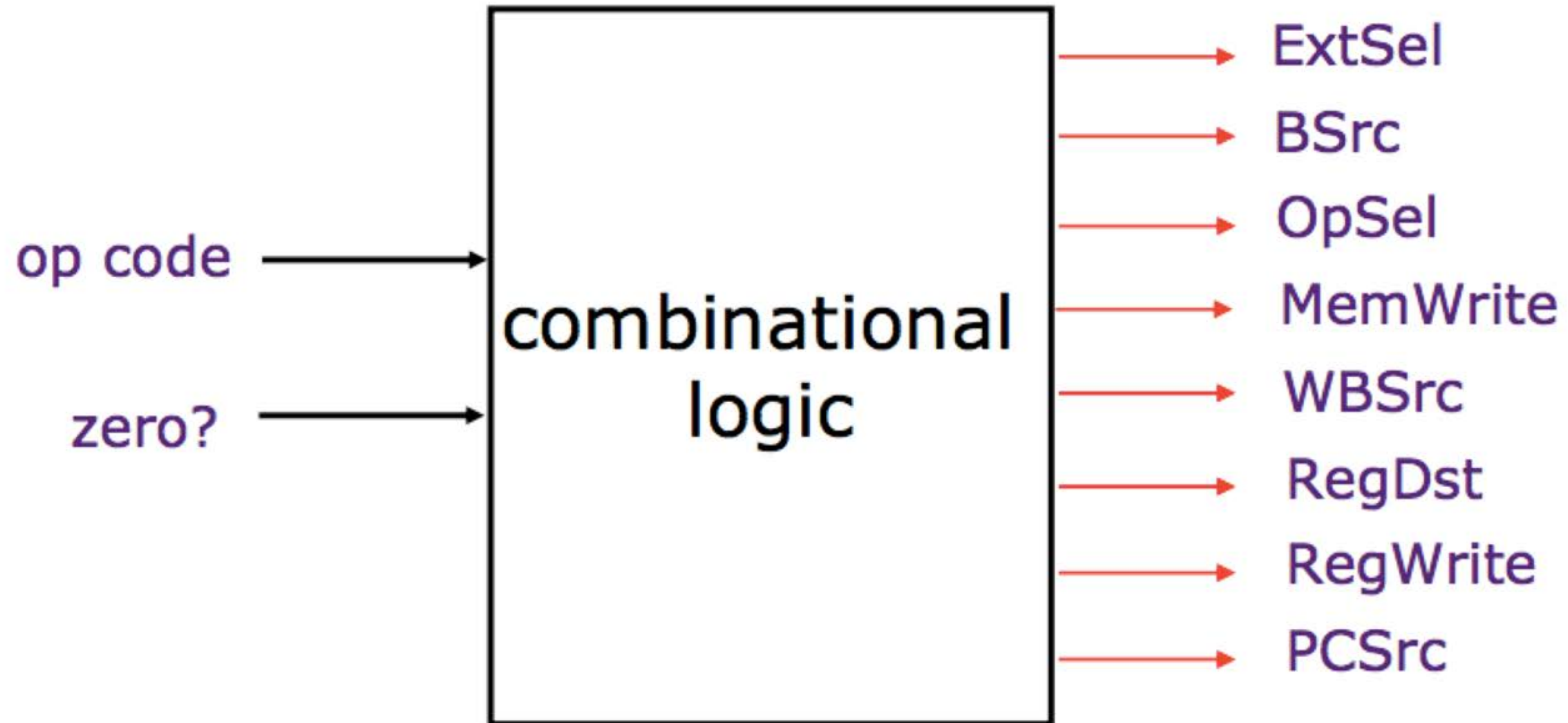
- clock period is sufficiently long for all of the following steps to be “completed”:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

$$\Rightarrow \quad t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

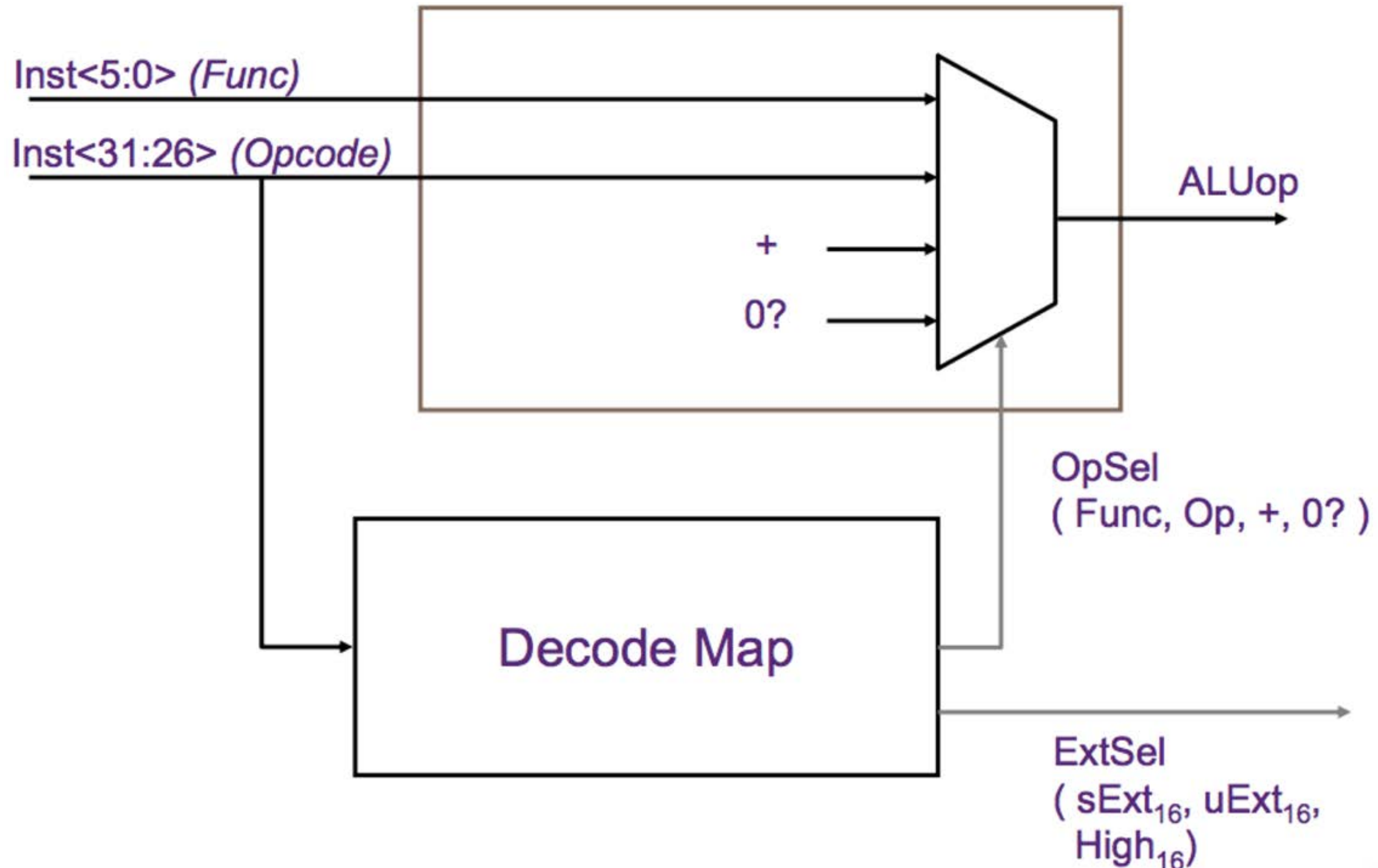
- At the rising edge of the following clock, the PC, the register file and the memory are updated

# Hardwired Control is pure Combinational Logic




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# ALU Control & Immediate Extension




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Hardwired Control Table

| Opcode | ExtSel | BSrc | OpSel | MemW | RegW | WBSrc | RegDst | PCSrc |
|---|---|---|---|---|---|---|---|---|
| ALU | * | Reg | Func | no | yes | ALU | rd | pc+4 |
| ALUi | $sExt_{16}$ | Imm | Op | no | yes | ALU | rt | pc+4 |
| ALUiu | $uExt_{16}$ | Imm | Op | no | yes | ALU | rt | pc+4 |
| LW | $sExt_{16}$ | Imm | + | no | yes | Mem | rt | pc+4 |
| SW | $sExt_{16}$ | Imm | + | yes | no | * | * | pc+4 |
| $BEQZ_{z=0}$ | $sExt_{16}$ | * | 0? | no | no | * | * | br |
| $BEQZ_{z=1}$ | $sExt_{16}$ | * | 0? | no | no | * | * | pc+4 |
| J | * | * | * | no | no | * | * | jabs |
| JAL | * | * | * | no | yes | PC | R31 | jabs |
| JR | * | * | * | no | no | * | * | rind |
| JALR | * | * | * | no | yes | PC | R31 | rind |

BSrc = Reg / Imm
RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC
PCSrc = pc+4 / br / rind / jabs

# RISC Processor in Chisel

THE UNIVERSITY OF SYDNEY

# Introduction

- Go over the design of a simple RISC processor in Chisel
- Code if from the chisel-tutorial Risc.scala example

# Ports

```
package TutorialExamples

import Chisel._

class Risc extends Module {
  val io = new Bundle {
    val isWr   = Bool(INPUT)
    val wrAddr = UInt(INPUT, 8)
    val wrData = Bits(INPUT, 32)
    val boot   = Bool(INPUT)
    val valid  = Bool(OUTPUT)
    val out    = Bits(OUTPUT, 32)
  }
```

```
val file = Mem(Bits(width = 32), 256)
val code = Mem(Bits(width = 32), 256)
val pc   = Reg(init=UInt(0, 8))

val add_op :: imm_op :: Nil = Enum(Bits(), 2)

val inst = code(pc)
val op   = inst(31,24)
val rci  = inst(23,16)
val rai  = inst(15, 8)
val rbi  = inst( 7, 0)

val ra = Mux(rai === Bits(0), Bits(0), file(rai))
val rb = Mux(rbi === Bits(0), Bits(0), file(rbi))
val rc = Bits(width = 32)
```

# Main Code

```
io.valid := Bool(false)
io.out   := Bits(0)
rc       := Bits(0)

when (io.isWr) {
  code(io.wrAddr) := io.wrData
} .elsewhen (io.boot) {
  pc := UInt(0)
} .otherwise {
    switch(op) {
      is(add_op) { rc := ra + rb }
      is(imm_op) { rc := (rai << UInt(8)) | rbi }
    }
    io.out := rc
    when (rci === UInt(255)) {
      io.valid := Bool(true)
    } .otherwise {
      file(rci) := rc
    }
    pc := pc + UInt(1)
  }
}
```

# Test Bench

```
class RiscTests(c: Risc) extends Tester(c) {
 def wr(addr: UInt, data: UInt)  = {
  poke(c.io.isWr,   1)
  poke(c.io.wrAddr, addr.litValue())
  poke(c.io.wrData, data.litValue())
  step(1)
 }
 def boot()  = {
  poke(c.io.isWr, 0)
  poke(c.io.boot, 1)
  step(1)
 }
 def tick()  = {
  poke(c.io.isWr, 0)
  poke(c.io.boot, 0)
  step(1)
 }
def I (op: UInt, rc: Int, ra: Int, rb: Int) =
  Cat(op, UInt(rc, 8), UInt(ra, 8), UInt(rb, 8))
 val app  = Array(I(c.imm_op,   1, 0, 1), // r1 <- 1
                  I(c.add_op,   1, 1, 1), // r1 <- r1 + r1
                  I(c.add_op,   1, 1, 1), // r1 <- r1 + r1
                  I(c.add_op, 255, 1, 0)) // rh <- r1
 wr(UInt(0), Bits(0)) // skip reset
 for (addr <- 0 until app.length)
  wr(UInt(addr), app(addr))
 boot()
 var k = 0
 do {
  tick(); k += 1
 } while (peek(c.io.valid) == 0 && k < 10)
 expect(k < 10, "TIME LIMIT")
 expect(c.io.out, 4)
}
```

# Datapath

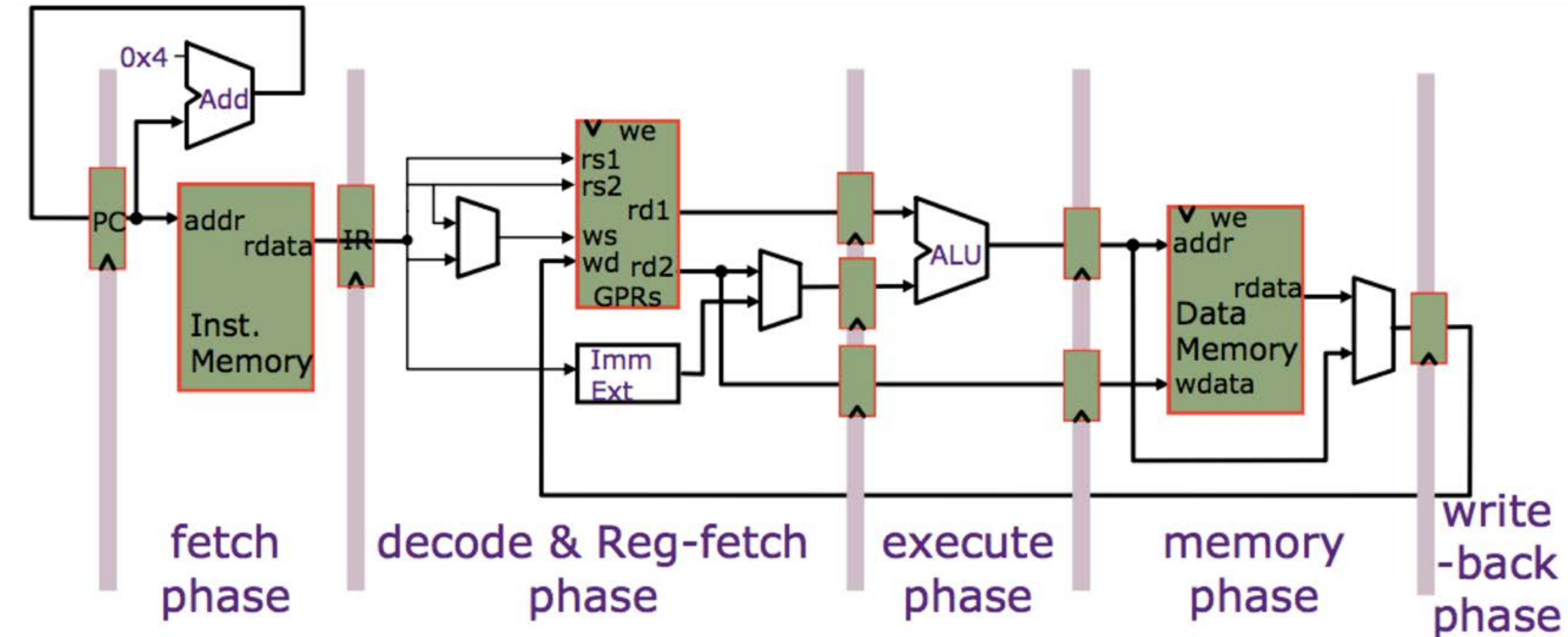
# Pipelined RISC Processor

# Pipelined MIPS

- To pipeline MIPS:
  - First build MIPS without pipelining with CPI=1
  - Next, add pipeline registers to reduce cycle time while maintaining CPI=1

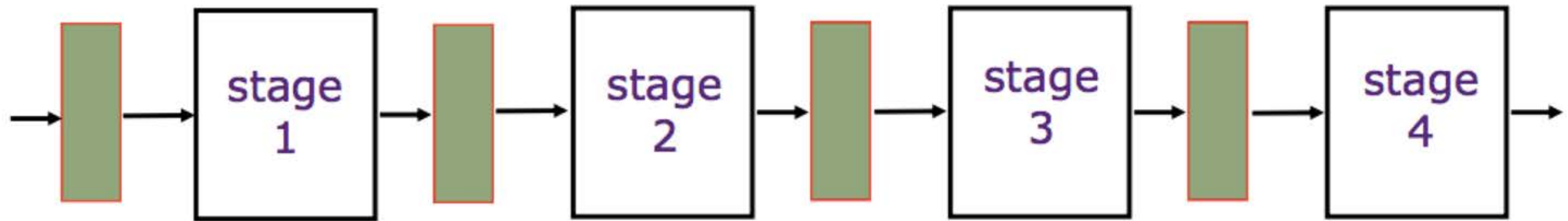
Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Pipelined Datapath




- Clock period can be reduced by dividing the execution of an instruction into multiple cycles
- $t_C > \max\{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\}$ ( $= t_{DM}$ probably)
- However, CPI will increase unless instructions are pipelined

# An Ideal Pipeline




- All objects go through the same stages
  - No sharing of resources between any two stages
  - Propagation delay through all pipeline stages is equal
  - The scheduling of an object entering the pipeline is not affected by the objects in other stages
- These conditions generally hold for industrial assembly lines.
- But can an instruction pipeline satisfy the last condition?

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# How to divide the datapath into stages

- Suppose memory is significantly slower than other stages. In particular, suppose

  t = 10 units IM

  t = 10 units DM

  t = 5 units ALU

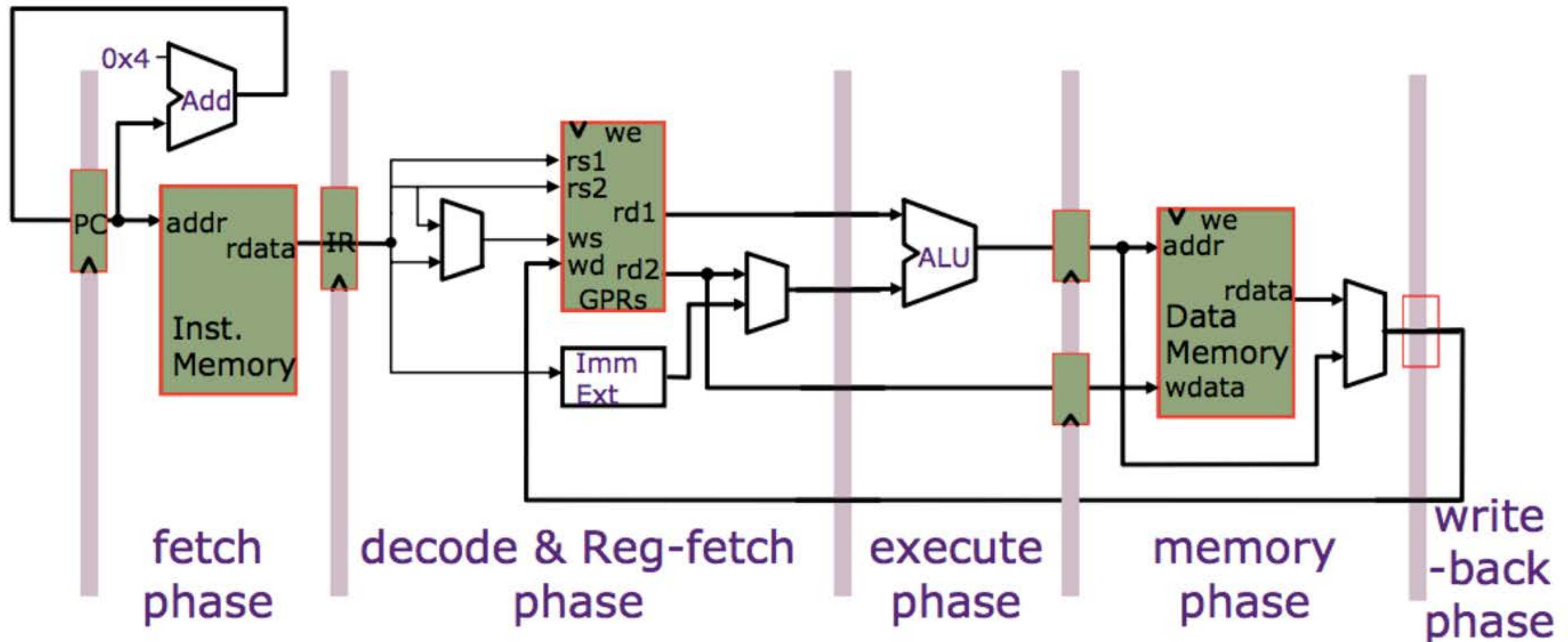
  t = 1 unit RF

  t = 1 unit RW

- Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Alternative Pipelining




$t_C > \max \{t_{IM}, t_{RF}+t_{ALU}, t_{DM}+t_{RW}\} = t_{DM}+t_{RW}$

$\Rightarrow$ increase the critical path by 10%

Write-back stage takes much less time than other stages. Suppose we combined it with the memory phase

THE UNIVERSITY OF SYDNEY

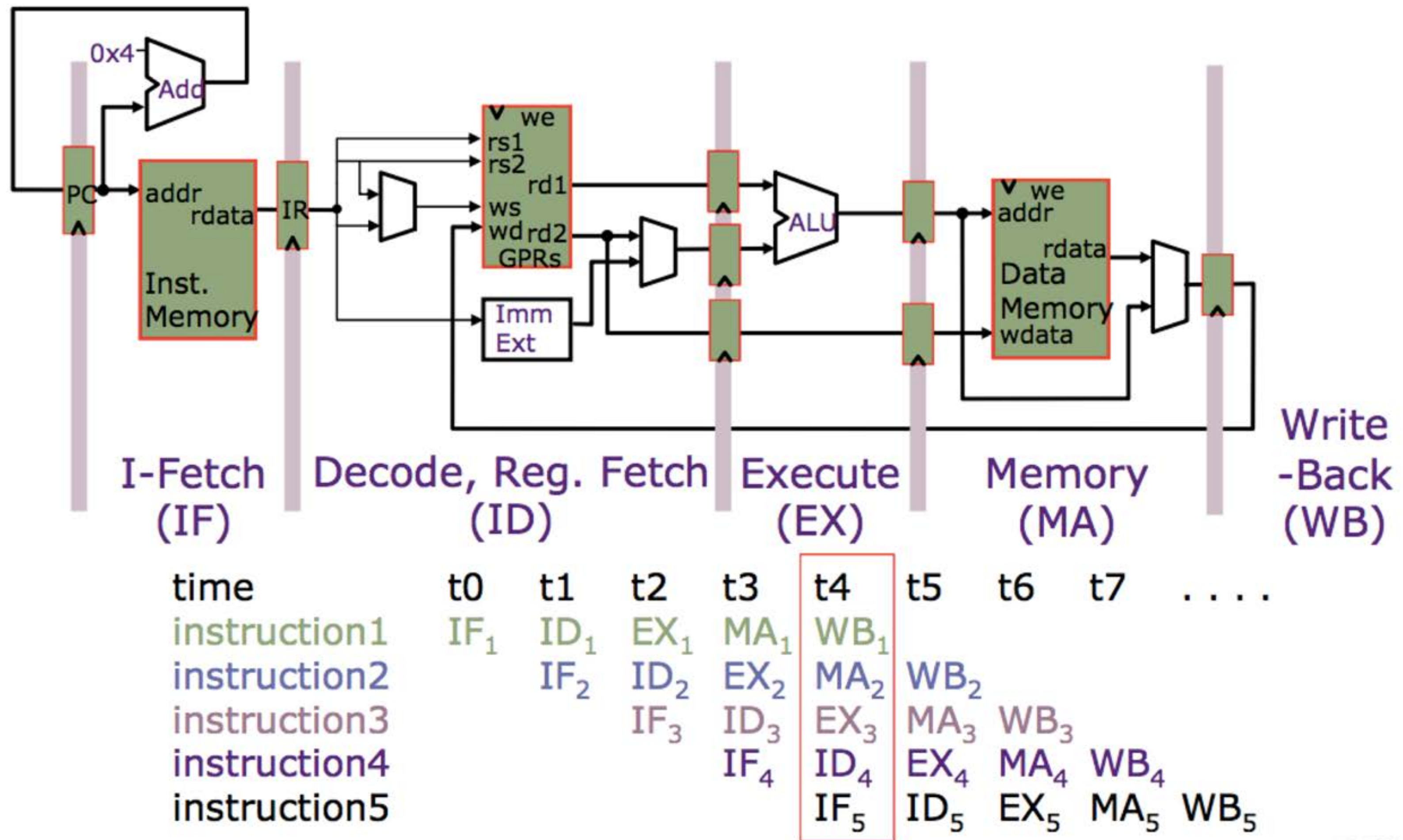
# Maximum Speedup from Pipelining

| Assumptions | Unpipelined $t_C$ | Pipelined $t_C$ | Speedup |
|---|---|---|---|
| 1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$ 4-stage pipeline | 27 | 10 | 2.7 |
| 2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 4-stage pipeline | 25 | 10 | 2.5 |
| 3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$ 5-stage pipeline | 25 | 5 | 5.0 |

It is possible to achieve higher speedup with more stages in the pipeline.

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# 5 stage Pipeline




Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Summary

- Design of a simple RISC processor
  - First you design datapath and then add control
  - All designs (serial, parallel) or otherwise follow a similar methodology
- Not covered
  - What happens if there are pipeline dependencies e.g. add r1, r2, r3; add r2, r1, r1
  - What happens if we branch e.g. add r1, r2, r3; beqz r1 loop
  - These are called hazards and can be resolved by stalling (low performance) and to some degree by forwarding (add internal paths to make data available earlier)
  - Refer to [1] and [2] for details

Source: Arvind (MIT Opencourseware licence http://ocw.mit.edu/terms/index.htm#cc)

# Exercise

- Draw the datapath and control for the Chisel RISC processor
- Modify the Chisel RISC processor to have a 2 stage pipeline (instruction fetch and execution)

THE UNIVERSITY OF SYDNEY

# References

[1] Patterson, D. A., and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufman.

[2] Hennessy, J. L., and D. A. Patterson. Computer Architecture: A Quantitative Approach, Morgan Kaufman.