Robotics and Embedded Systems

Technische Universität München

Embedded Hardware (1)

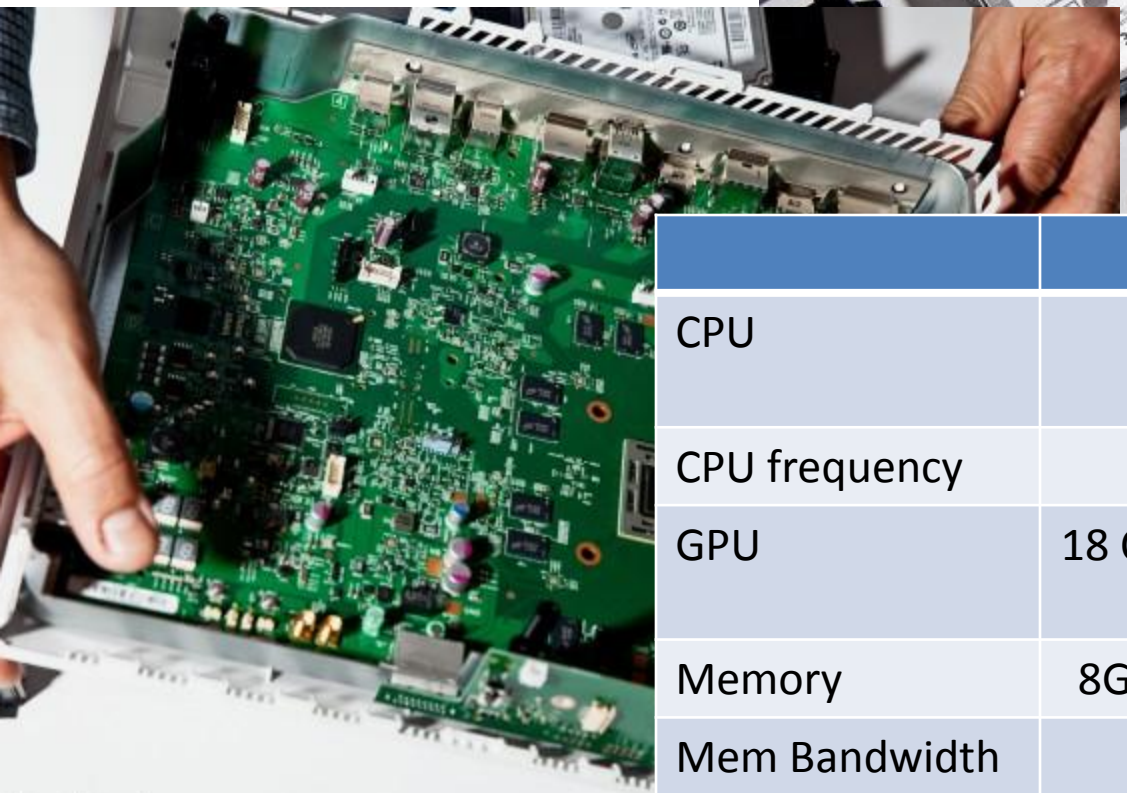
Kai Huang

News: PS4 and Xbox One are Coming

The Hardware

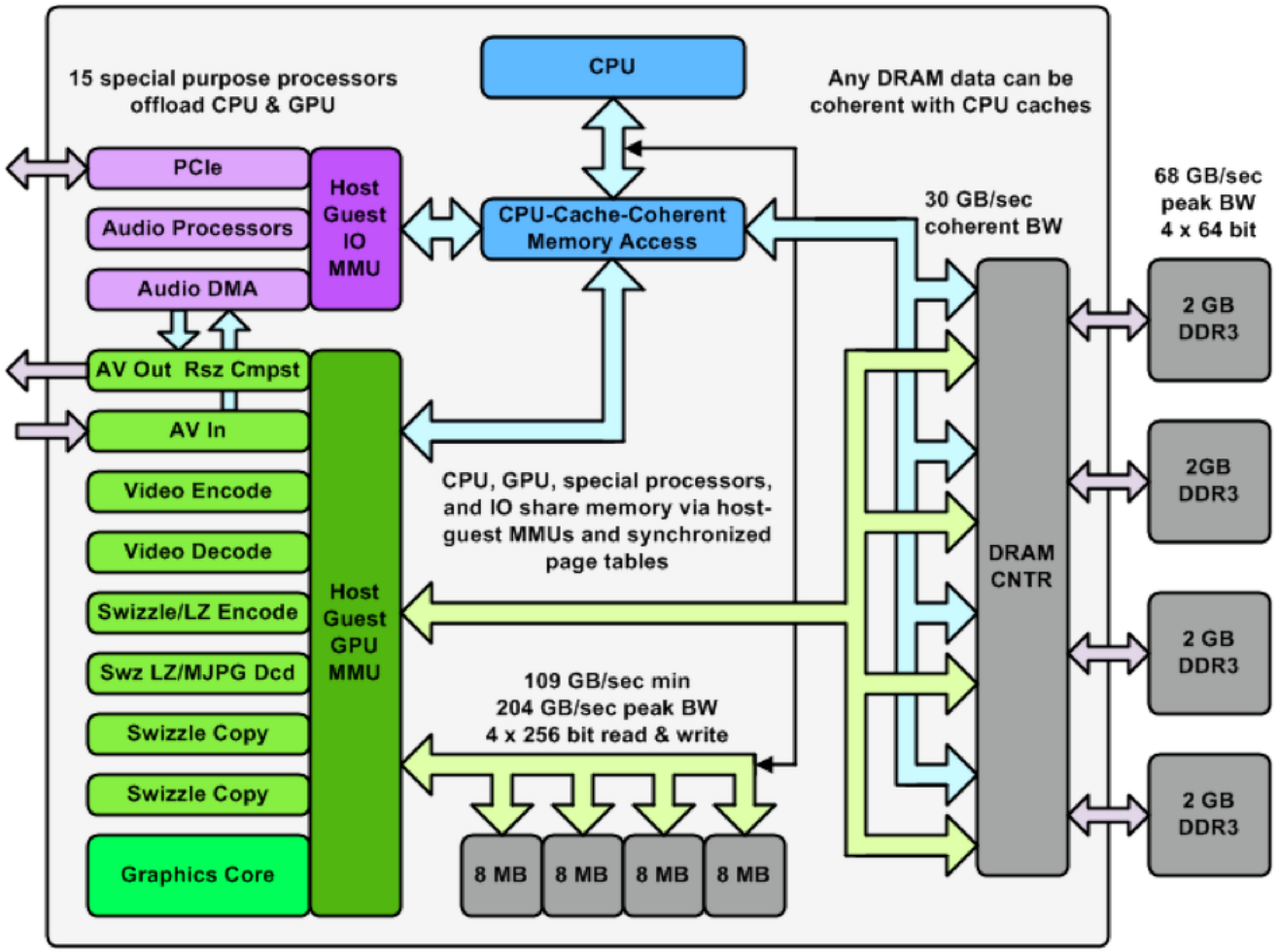
PS4→

Xbox One ↓



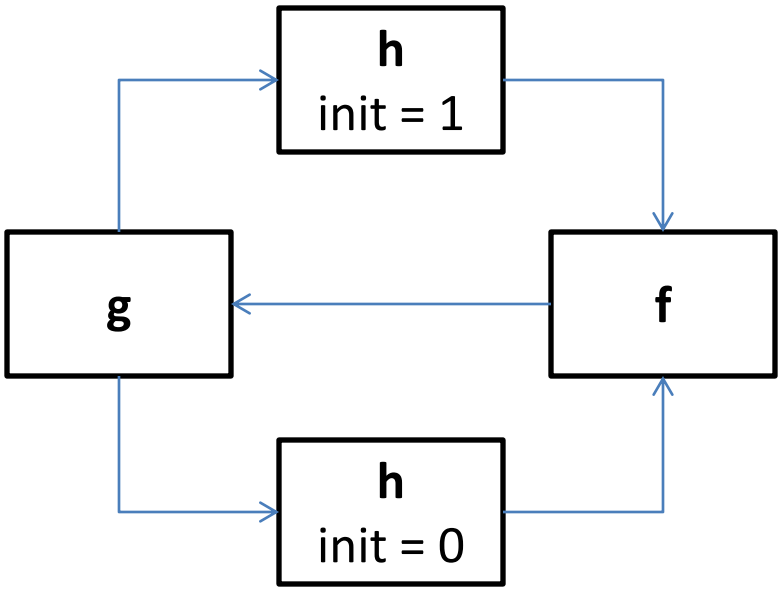
	PS4	Xbox One
CPU	semi-custom x86 AMD APU 28nm 8-core Jaguar CPU	
CPU frequency	1.6 GHz	1.75 GHz
GPU	18 CUs:1152 shaders (800MHz)	12 CUs:768 shader (853 MHz)
Memory	8G 5500MHz DDR5	8G 2133MHz DDR3
Mem Bandwidth	176GB/sec	68.3GB/sec
Embedded SRAM	N/A	32MB (204GB/sec)

SoC Components

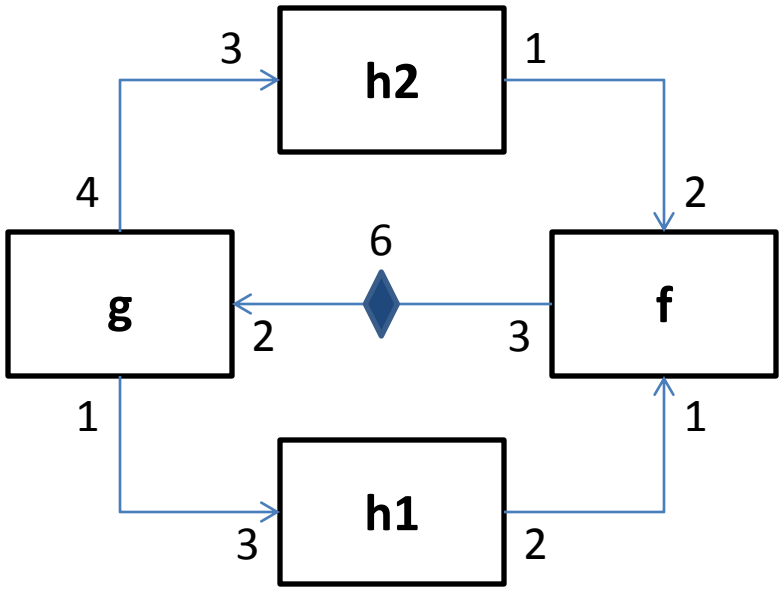


Dataflow MoC Recap

Kahn Process Network



Synchronous DataFlow



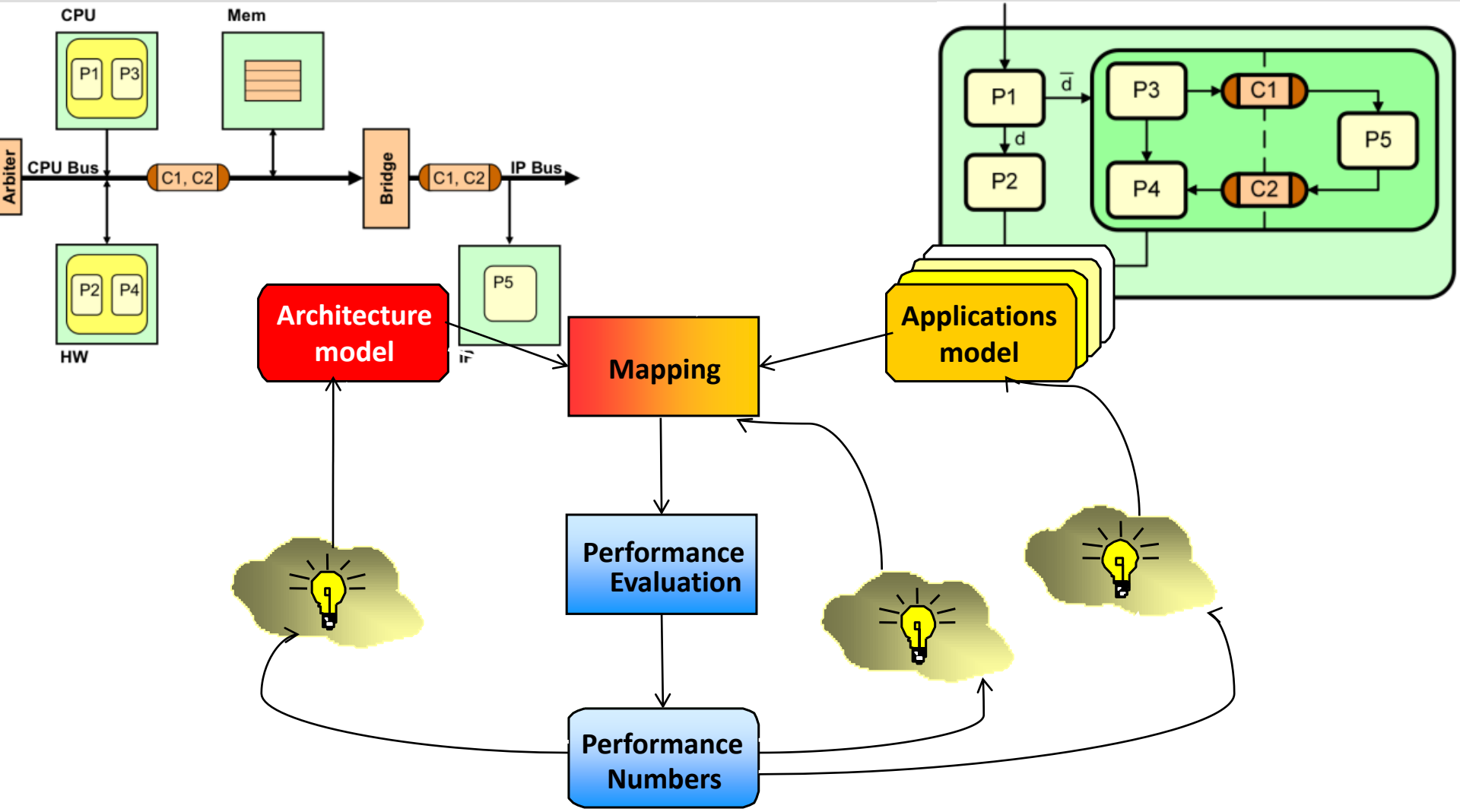
Outline

- Processor
- Memory
- I/O

Outline

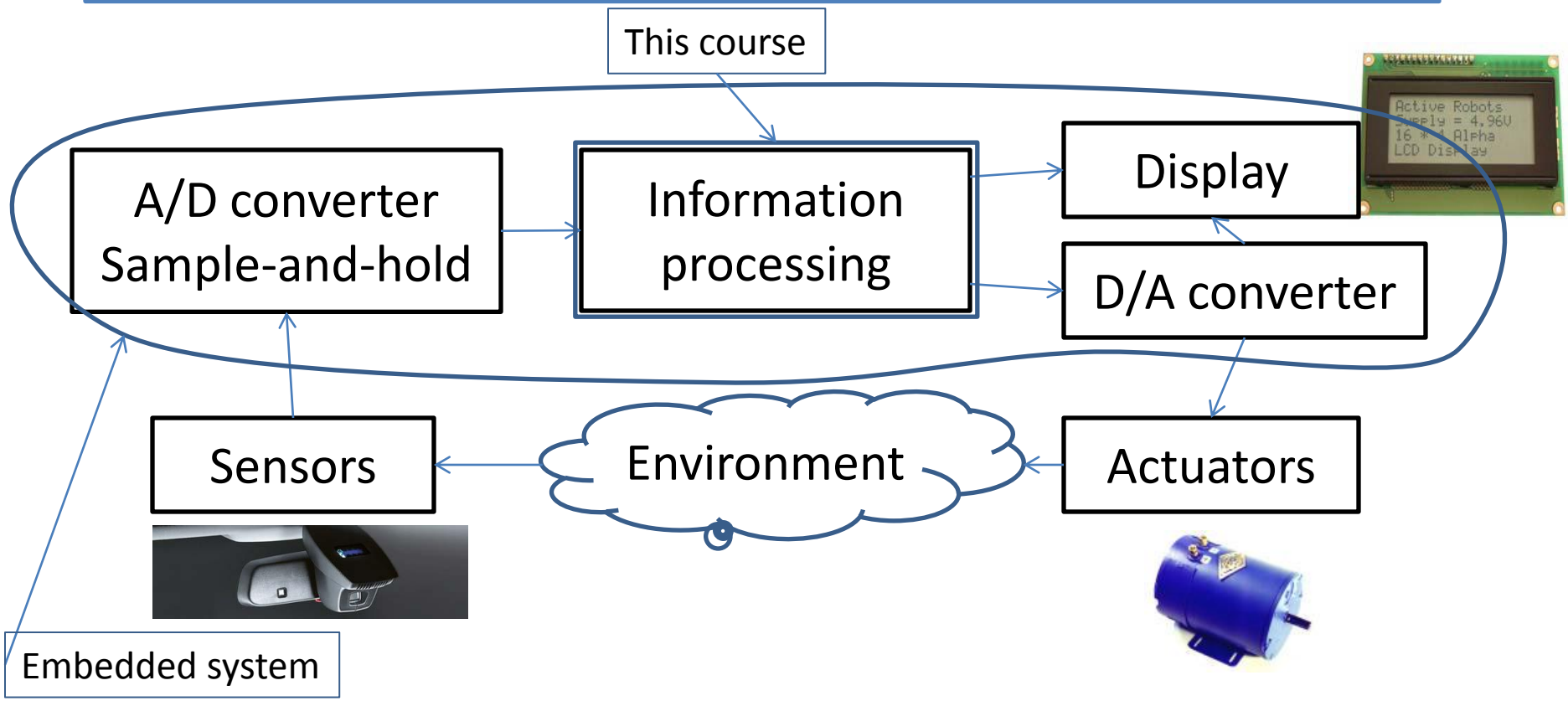
- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types
- Memory
- I/O

Y-Chart Methodology



Embedded System Hardware

Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):

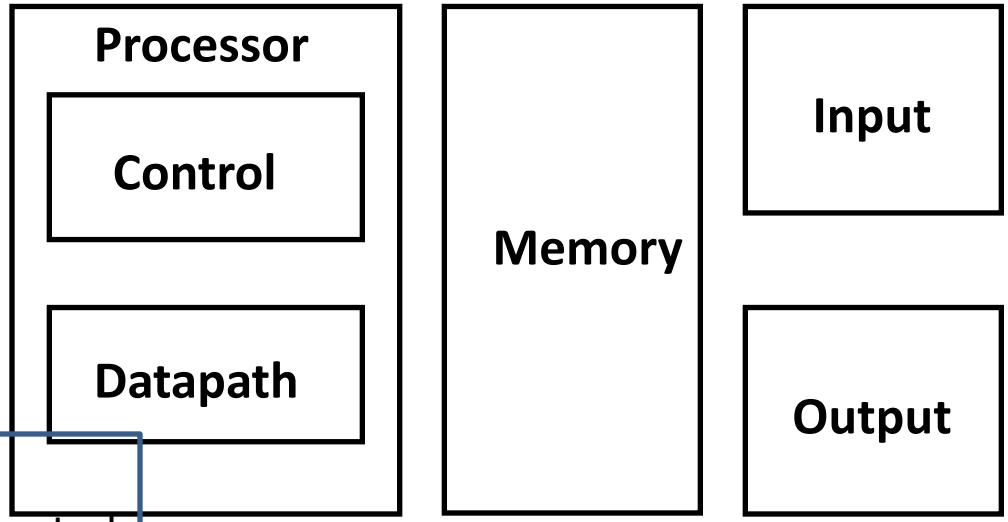


The Big Picture

- Since 1946 all computers have had 5 components

Input unit accepts information:
- Human operators,
- Electromechanical devices
- Other computers

Control unit coordinates various actions:
- Input,
- Output
- Processing



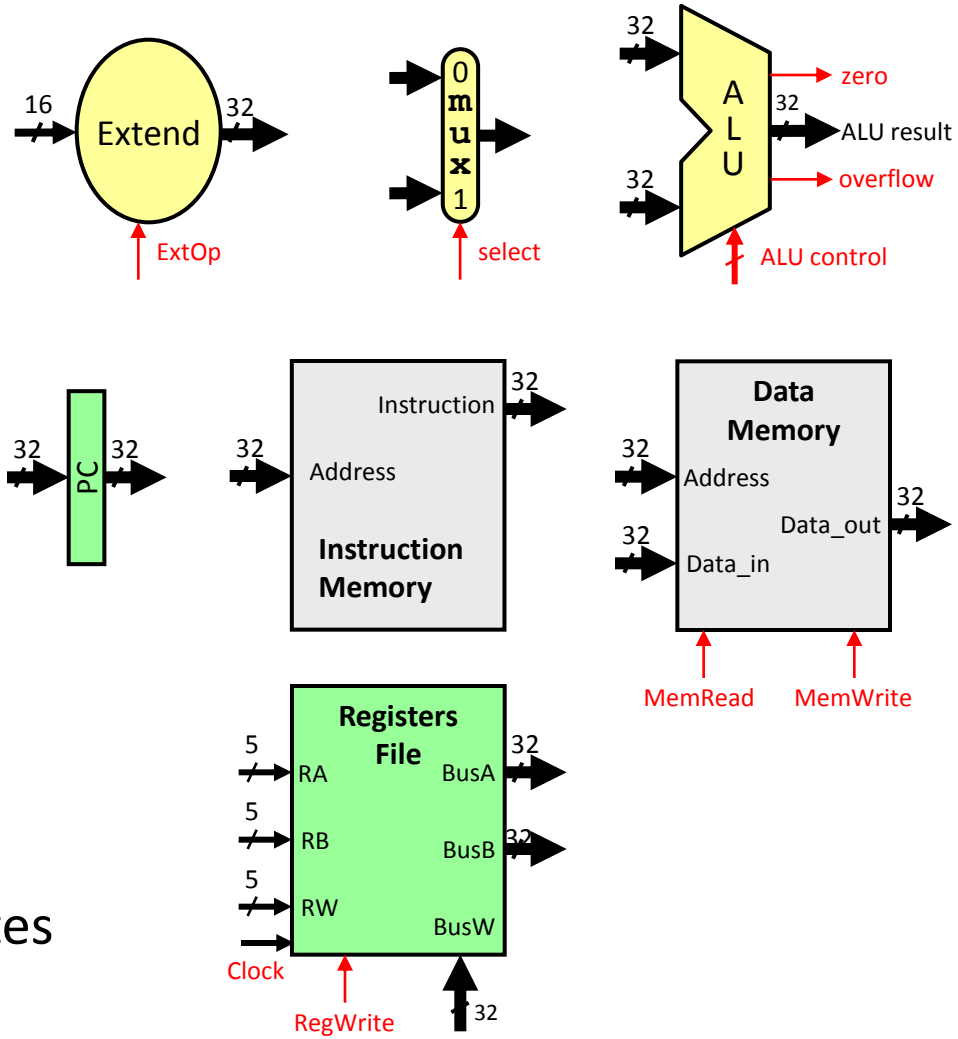
Datapath:
- the part of the central processing unit (CPU) that does the actual computations

Stores information:
- Instructions,
- Data

Output unit sends results of processing:
- To a monitor display,
- To a printer

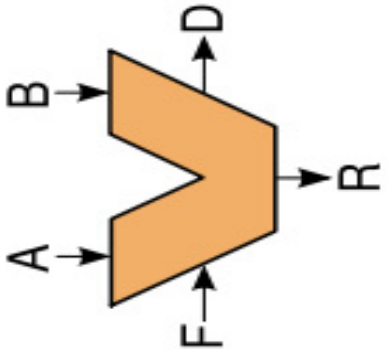
Datapath Components

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file
- Clocking methodology
  - Timing of reads and writes

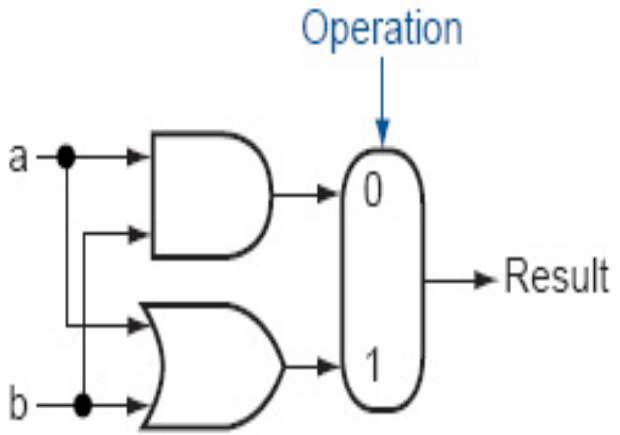


ALU: Arithmetic Logic Unit

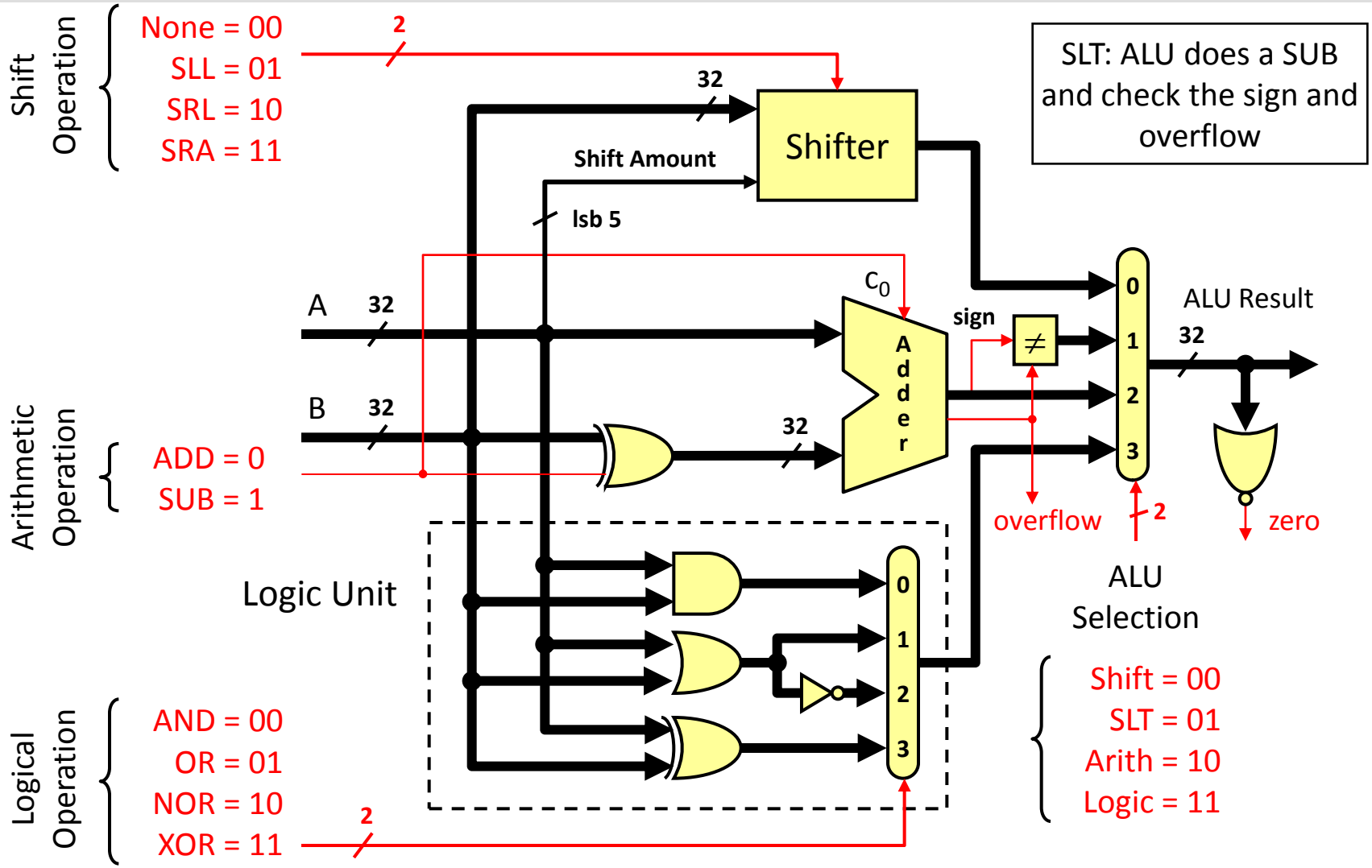
1. ALU is a digital circuit that performs Arithmetic (Add, Sub, . . .) and Logical (AND, OR, NOT) operations.
2. John Von Neumann proposed the ALU in 1945 when he was working on EDVAC.



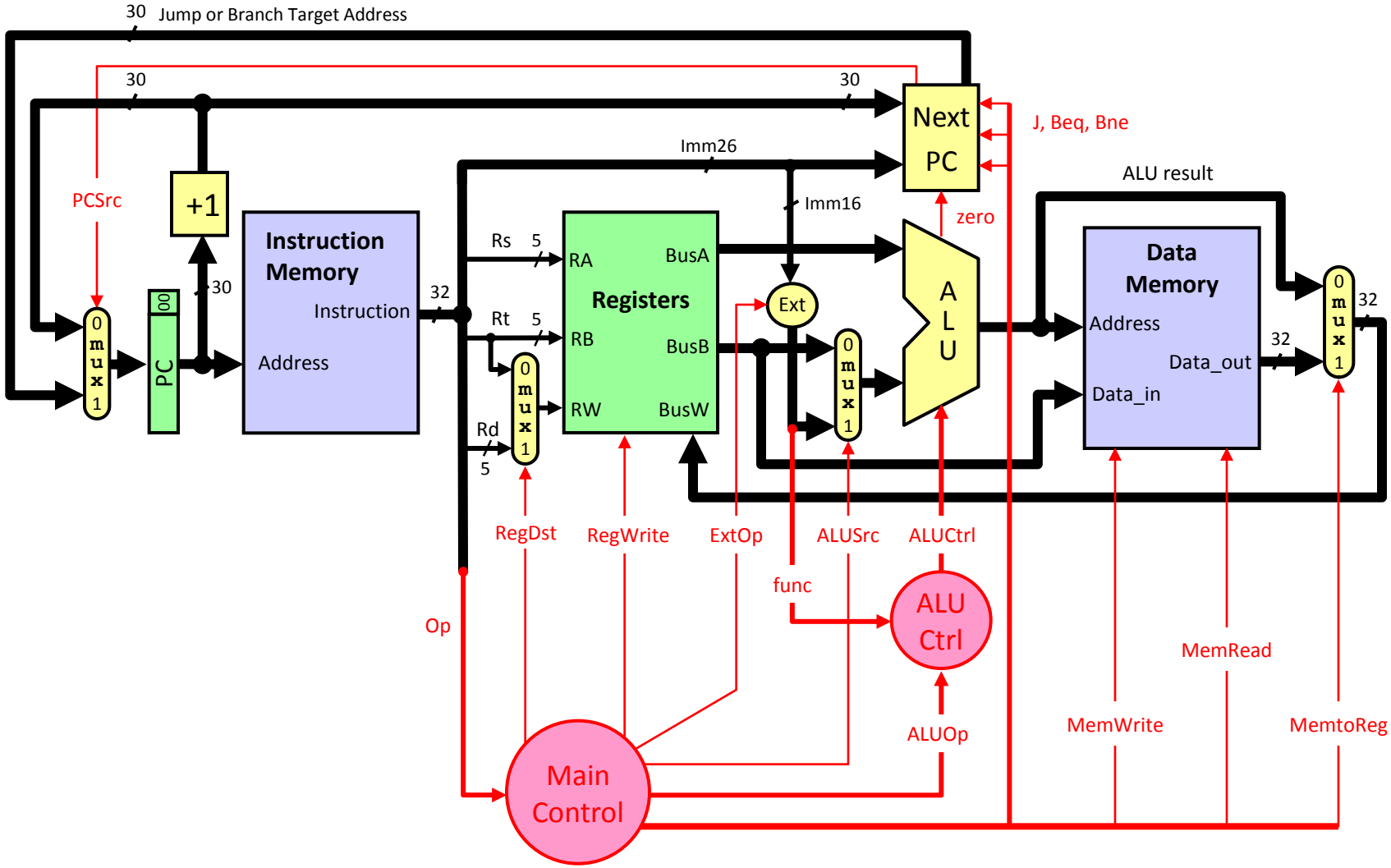
1-Bit ALU



Multifunction ALU



Single-Cycle Datapath (with Control Signal)



Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4
BEQ	if (Reg(Rs) == Reg(Rt))	
	PC ← PC + 4 + 4 × sign_extend(Im16)	
	else PC ← PC + 4	

Instructions are Executed in Steps

- R-type
  - Fetch instruction: Instruction ← MEM[PC]
  - Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)
  - Execute operation: ALU_result ← func(data1, data2)
  - Write ALU result: Reg(Rd) ← ALU_result
  - Next PC address: PC ← PC + 4
- I-type
  - Fetch instruction: Instruction ← MEM[PC]
  - Fetch operands: data1 ← Reg(Rs), data2 ← Extend(imm16)
  - Execute operation: ALU_result ← op(data1, data2)
  - Write ALU result: Reg(Rt) ← ALU_result
  - Next PC address: PC ← PC + 4
- BEQ
  - Fetch instruction: Instruction ← MEM[PC]
  - Fetch operands: data1 ← Reg(Rs), data2 ← Reg(Rt)
  - Equality: zero ← subtract(data1, data2)
  - Branch: if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
    else PC ← PC + 4

Instruction Execution Examples

- **LW**

lw Rt,C(Rs)

Fetch instruction:	Instruction ← MEM[PC]
Fetch base register:	base ← Reg(Rs)
Calculate address:	address ← base + sign_extend(imm16)
Read memory:	data ← MEM[address]
Write register Rt:	Reg(Rt) ← data
Next PC address:	PC ← PC + 4

- **SW**

sw Rt,C(Rs)

Fetch instruction:	Instruction ← MEM[PC]
Fetch registers:	base ← Reg(Rs), data ← Reg(Rt)
Calculate address:	address ← base + sign_extend(imm16)
Write memory:	MEM[address] ← data
Next PC address:	PC ← PC + 4

- **Jump**

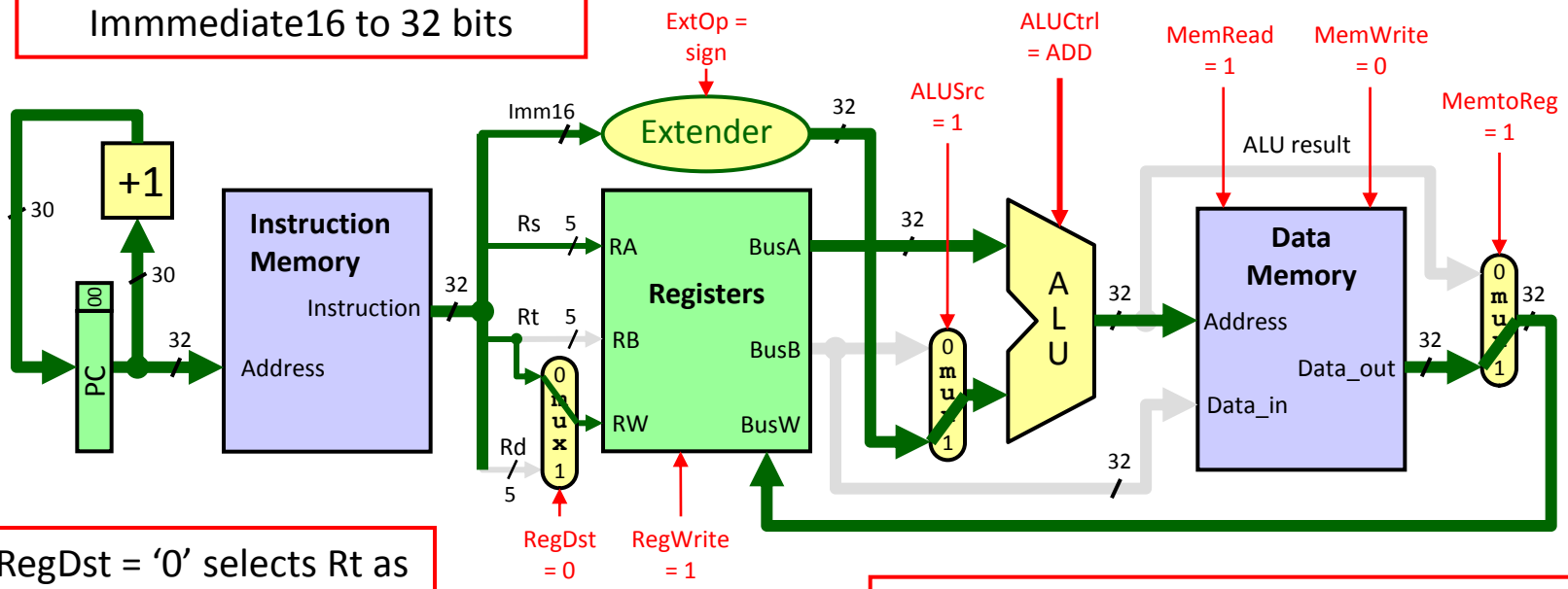
j C

Fetch instruction:	Instruction ← MEM[PC]
Target PC address:	target ← PC[31:28] , Imm26 , '00'
Jump:	PC ← target

concatenation

Execution of Load Instruction: lw Rt,C(Rs)

ExtOp = 'sign' to sign-extend Immmediate16 to 32 bits



RegDst = '0' selects Rt as destination register

MemRead = '1' to read data memory

ALUSrc = '1' selects extended immediate as second ALU input

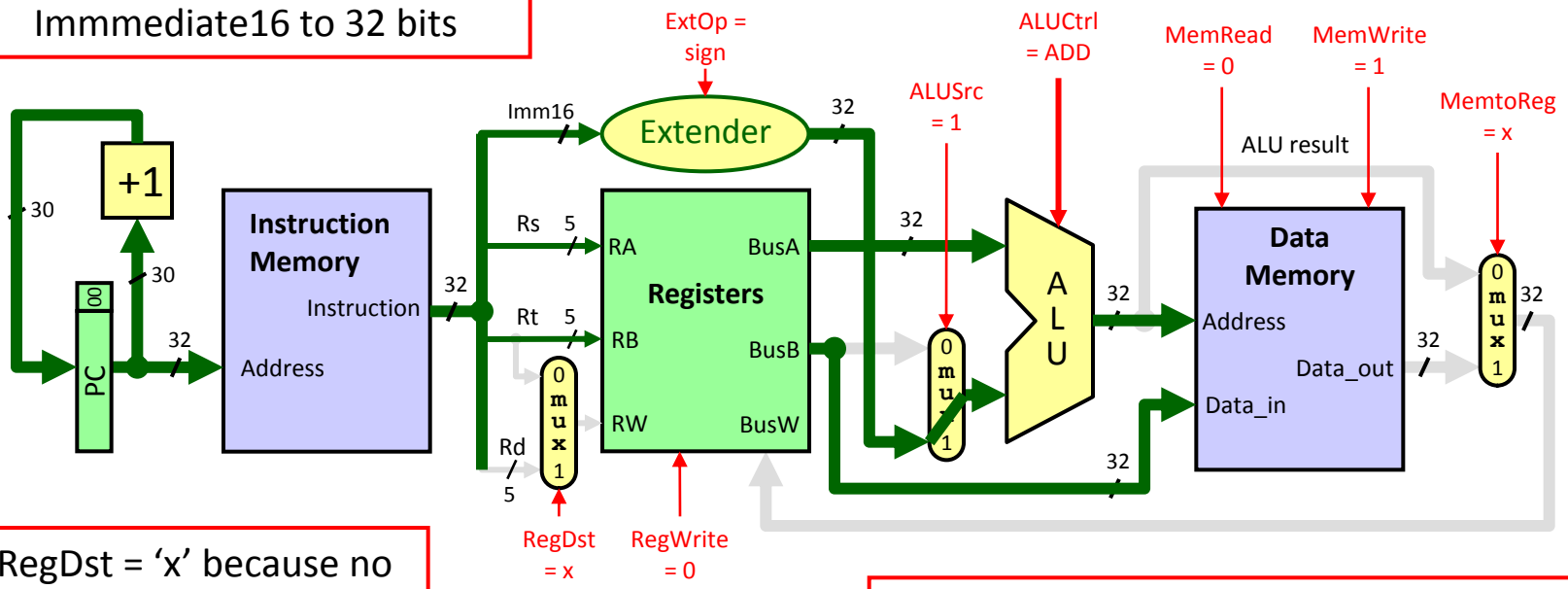
MemtoReg = '1' places the data read from memory on BusW

ALUCtrl = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

RegWrite = '1' to write the memory data on BusW to register Rt

Execution of Store Instruction: sw Rt,C(Rs)

ExtOp = 'sign' to sign-extend Immmediate16 to 32 bits



RegDst = 'x' because no destination register

MemWrite = '1' to write data memory

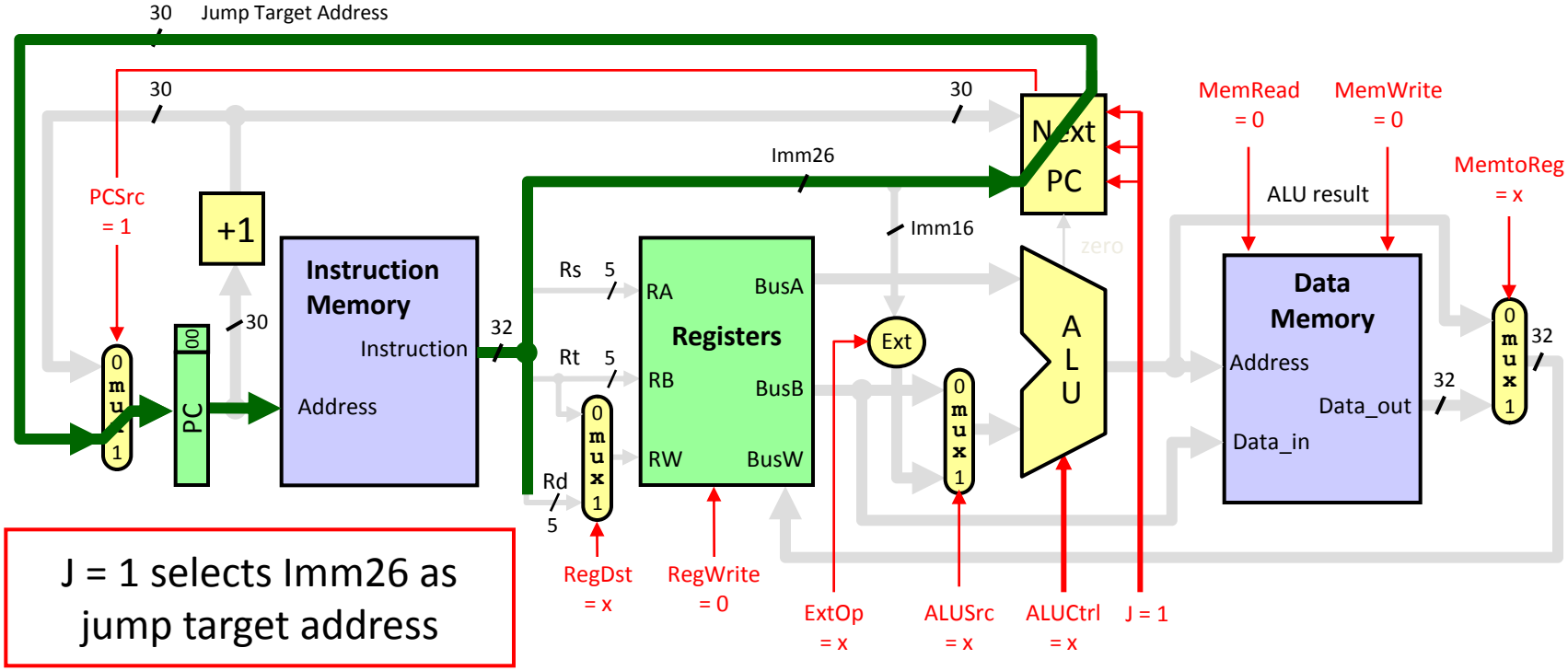
ALUSrc = '1' to select the extended immediate as second ALU input

MemtoReg = 'x' because we don't care what data is placed on BusW

ALUCtrl = 'ADD' to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

RegWrite = '0' because no register is written by the store instruction

Execution of Jump Instruction: j C



J = 1 selects Imm26 as jump target address

Upper 4 bits are from the incremented PC

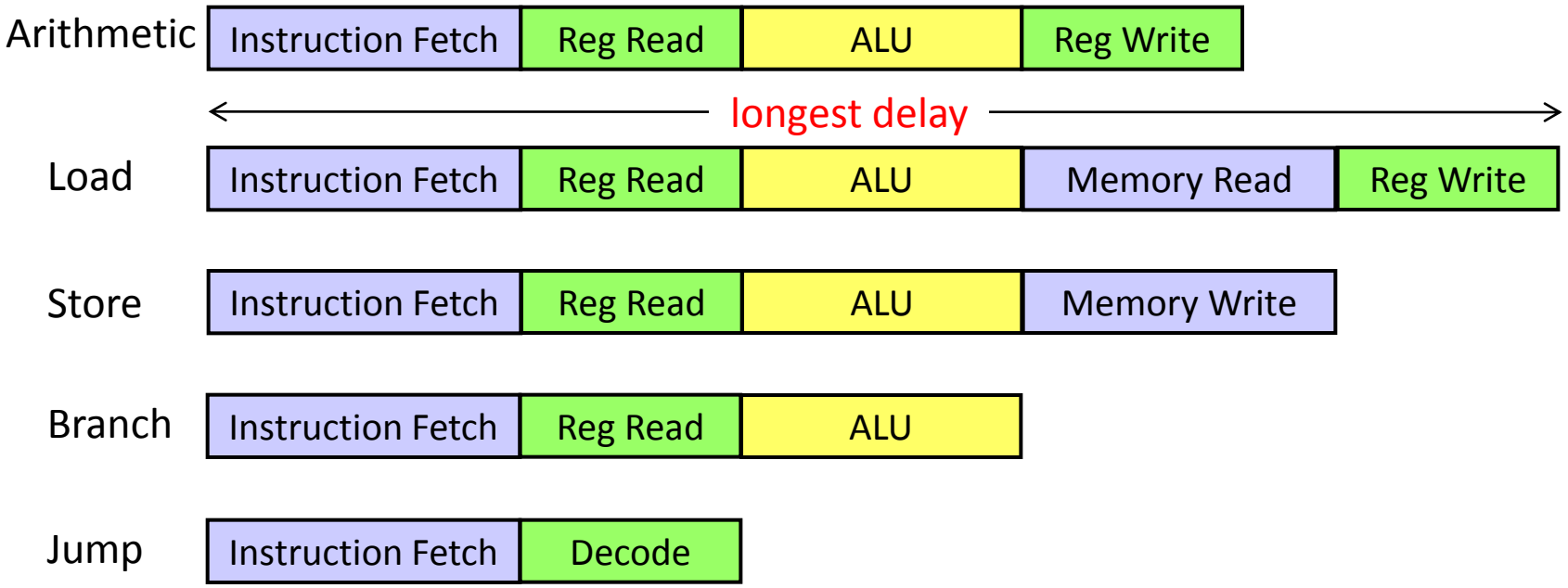
PCSrc = 1 to select jump target address

MemRead, MemWrite & RegWrite are 0

We don't care about RegDst, ExtOp, ALUSrc, ALUCtrl, and MemtoReg

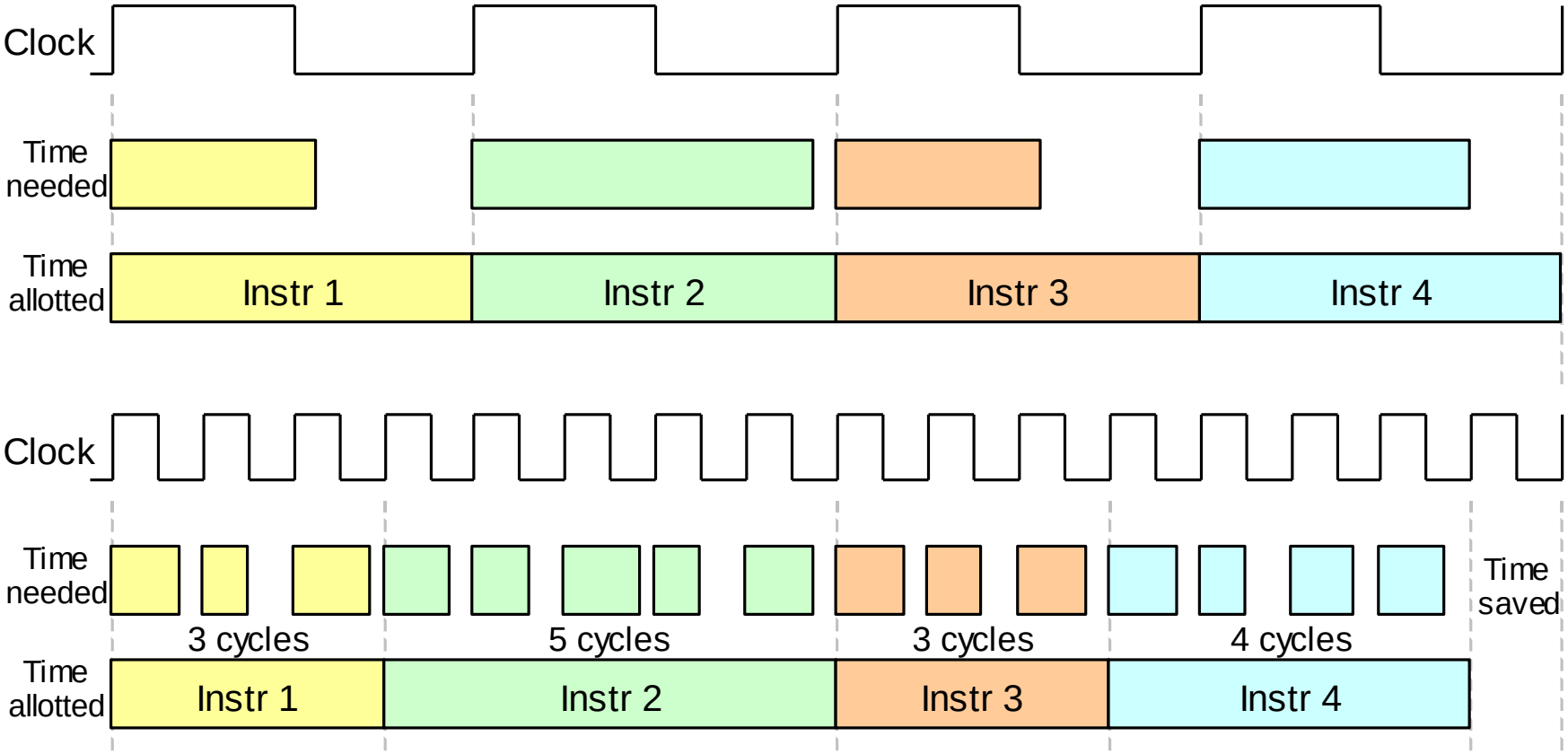
Drawbacks of Single Cycle Processor

- Long cycle time
  - All instructions take as much time as the slowest



- Alternative Solution: Multicycle implementation
  - Break down instruction execution into multiple cycles

Single-Cycle vs. Multicycle



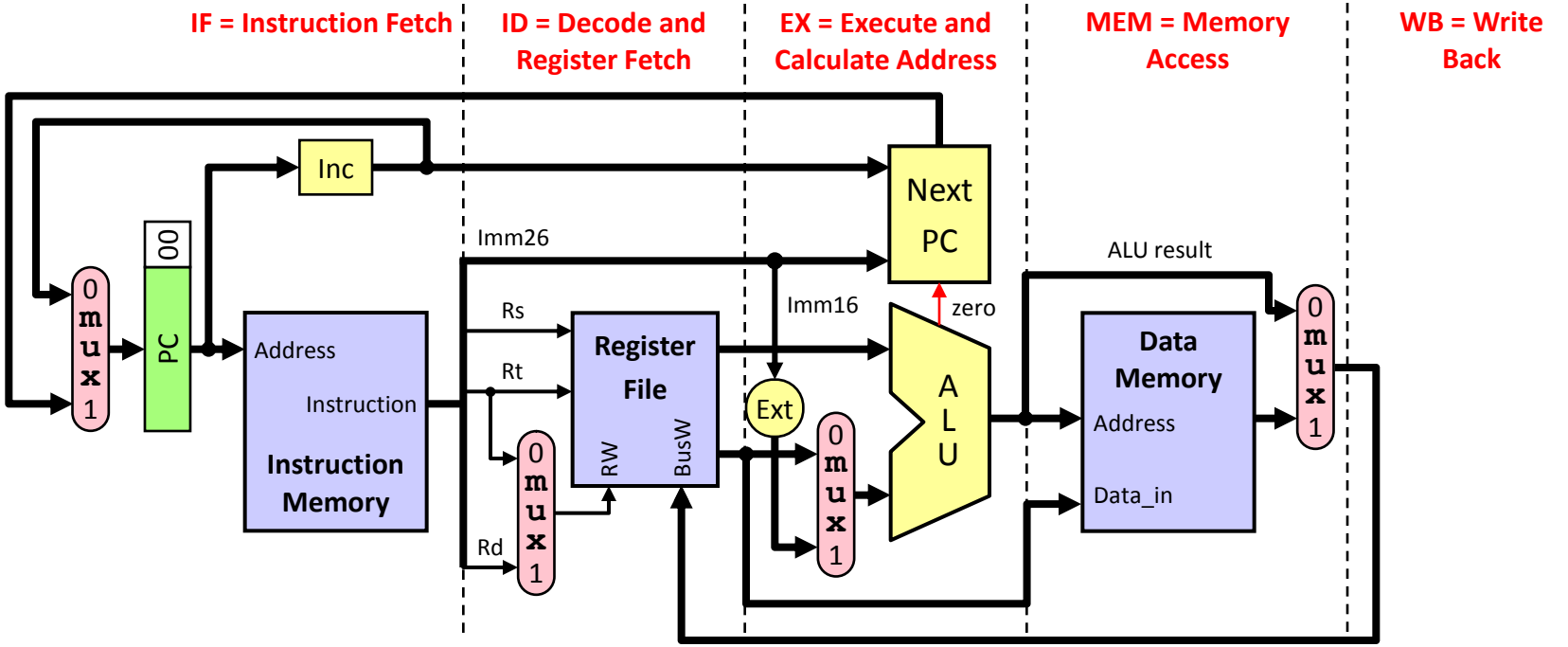
Outline

- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types
- Memory
- I/O

Single-Cycle Datapath

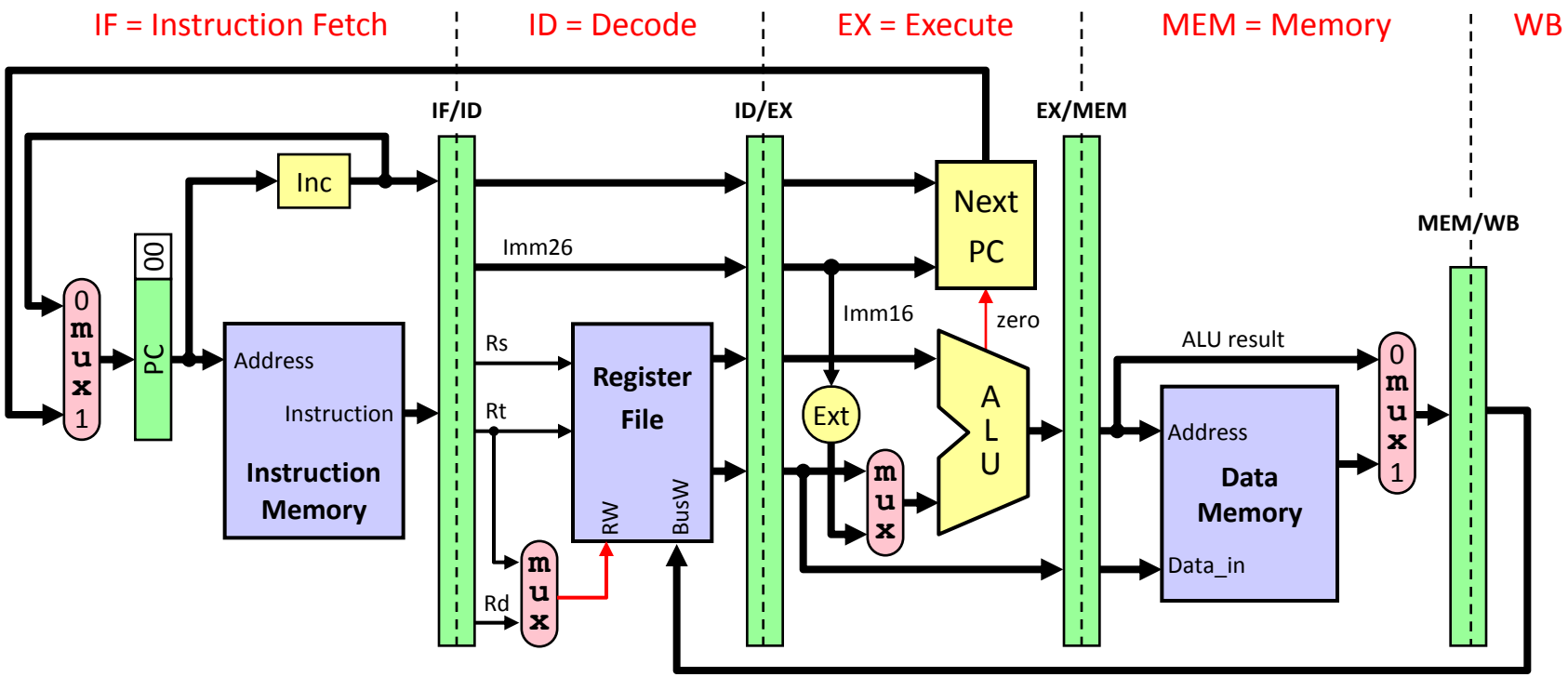
- Shown below is the single-cycle datapath
- How to pipeline this single-cycle datapath?

  Answer: Introduce registers at the end of each stage



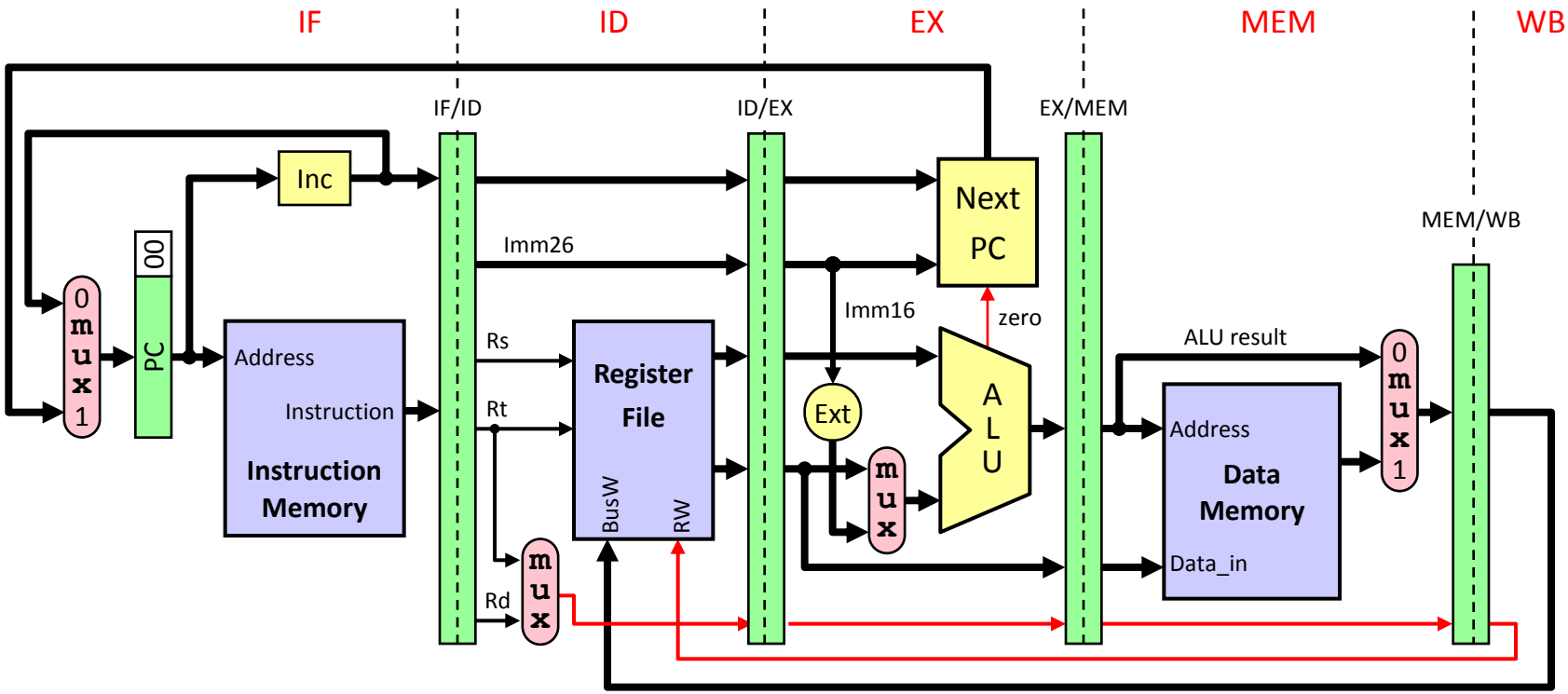
Pipelined Datapath

- Pipeline registers, in green, separate each pipeline stage
- Pipeline registers are labeled by the stages they separate
- Is there a problem with the register destination address?



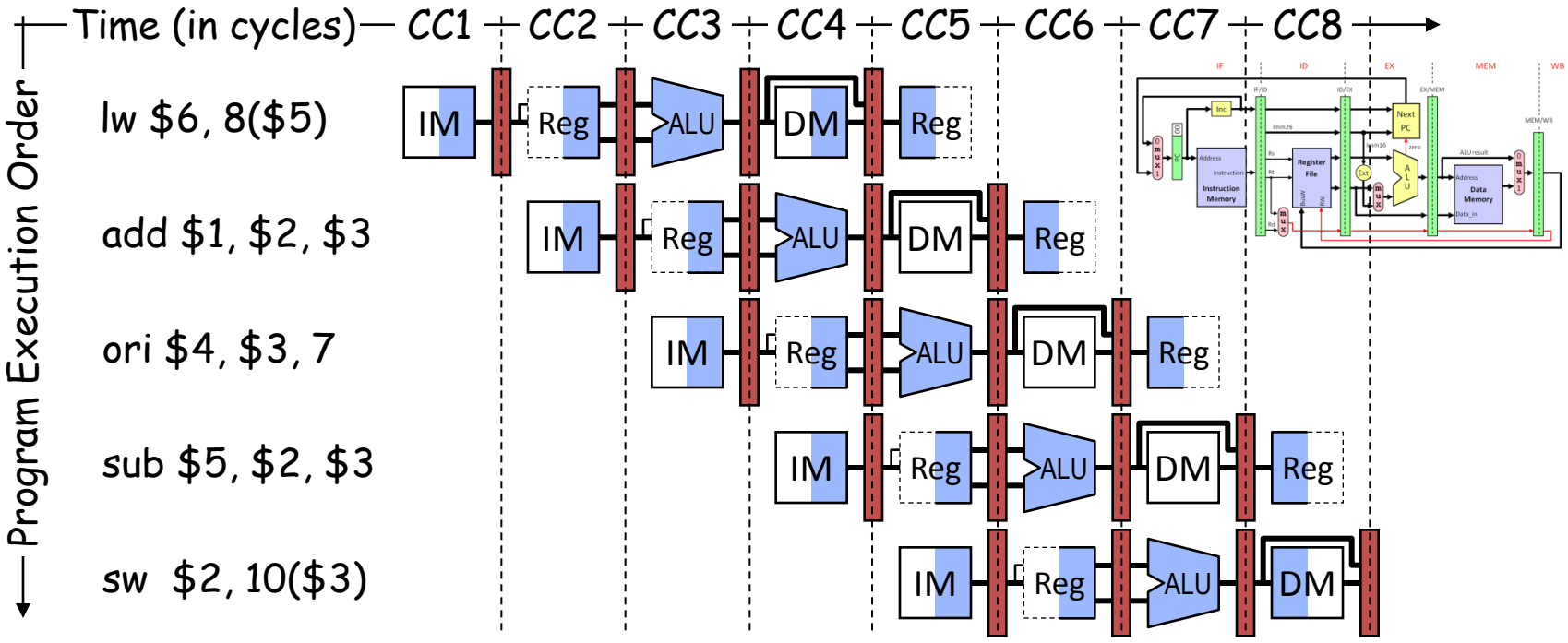
Corrected Pipelined Datapath

- Destination register number should come from MEM/WB
  - Along with the data during the written back stage
- Destination register number is passed from ID to WB stage

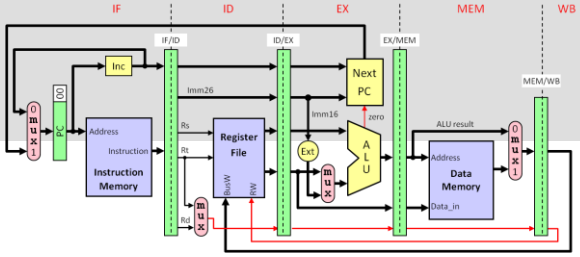


Graphically Representing Pipelines

- Multiple instruction execution over multiple clock cycles
  - Instructions are listed in execution order from top to bottom
  - Clock cycles move from left to right
  - Figure shows the use of resources at each stage and each cycle



Instruction–Time Diagram



- Diagram shows:
  - Which instruction occupies what stage at each clock cycle
- Instruction execution is pipelined over the 5 stages

Up to five instructions can be in execution during a single cycle

ALU instructions skip the MEM stage. Store instructions skip the WB stage

Instruction Order

lw	$7, 8($3)
lw	$6, 8($5)
ori	$4, $3, 7
sub	$5, $2, $3
sw	$2, 10($3)

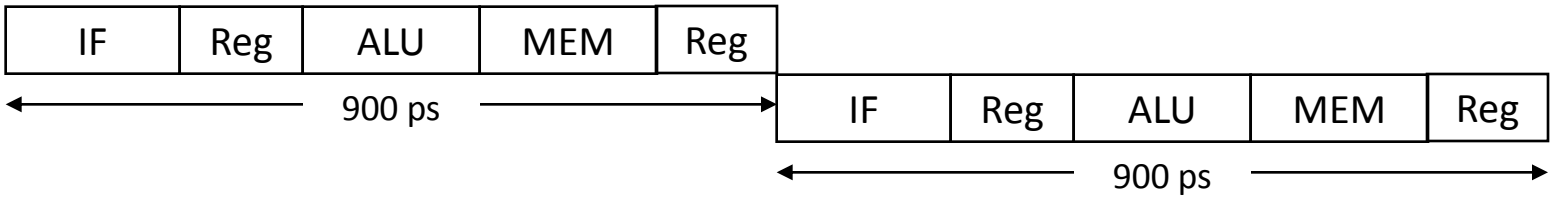
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
lw $7, 8($3)	IF	ID	EX	MEM	WB				
lw $6, 8($5)		IF	ID	EX	MEM	WB			
ori $4, $3, 7			IF	ID	EX	–	WB		
sub $5, $2, $3				IF	ID	EX	–	WB	
sw $2, 10($3)					IF	ID	EX	MEM	–

Time

Single-Cycle vs Pipelined Performance

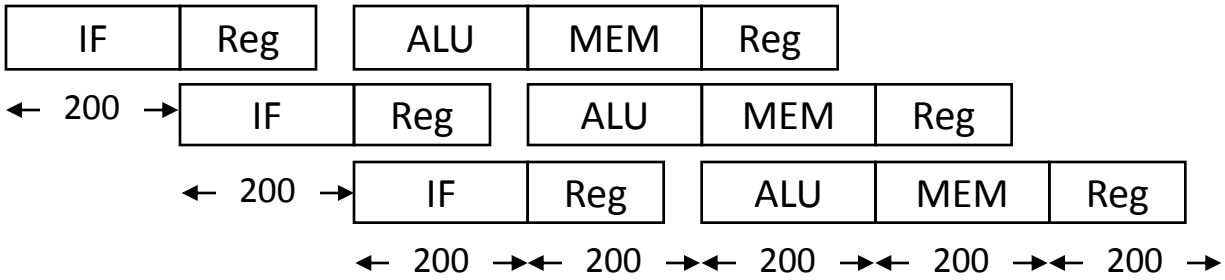
- Consider a 5-stage instruction execution in which …
  - Instruction fetch = ALU operation = Data memory access = 200 ps
  - Register read = register write = 150 ps
- What is the single-cycle non-pipelined time?
- What is the pipelined cycle time?
- What is the speedup factor for pipelined execution?
- Solution

Non-pipelined cycle = 200+150+200+200+150 = 900 ps



Single-Cycle versus Pipelined – cont'd

- Pipelined cycle time = max(200, 150) = 200 ps



- CPI for pipelined execution = 1
  - One instruction completes each cycle (ignoring pipeline fill)
- Speedup of pipelined execution = 900 ps / 200 ps = 4.5
  - Instruction count and CPI are equal in both cases
- Speedup factor is less than 5 (number of pipeline stage)
  - Because the pipeline stages are not balanced

Summary between Datapaths

	Single Cycle	Multiple Cycle	Pipeline
Clock Cycle Time	Long (Long enough for the slowest instruction)	Short (long enough for the slowest instruction step)	Short (long enough for the slowest pipeline stage)
Cycle Per Instruction	1 clock cycle per instruction (by definition)	Variable number of clock cycles per instruction	Fixed number of clock cycles per instruction, one for each pipeline stage
# instruction executing concurrently	1	1	# pipeline stage
Duplicate Hardware	Yes, since we can use a functional unit (FU) for at most one subtask per instruction	No, since the instruction generally is broken into single-FU steps	Yes, to avoid restriction on pipeline execution
Extra Register	No	Yes, to hold results for the next step	Yes, to provide results for the pipeline stage
Performance	Baseline	Faster, but not too fast	Fastest , if pipeline is balanced

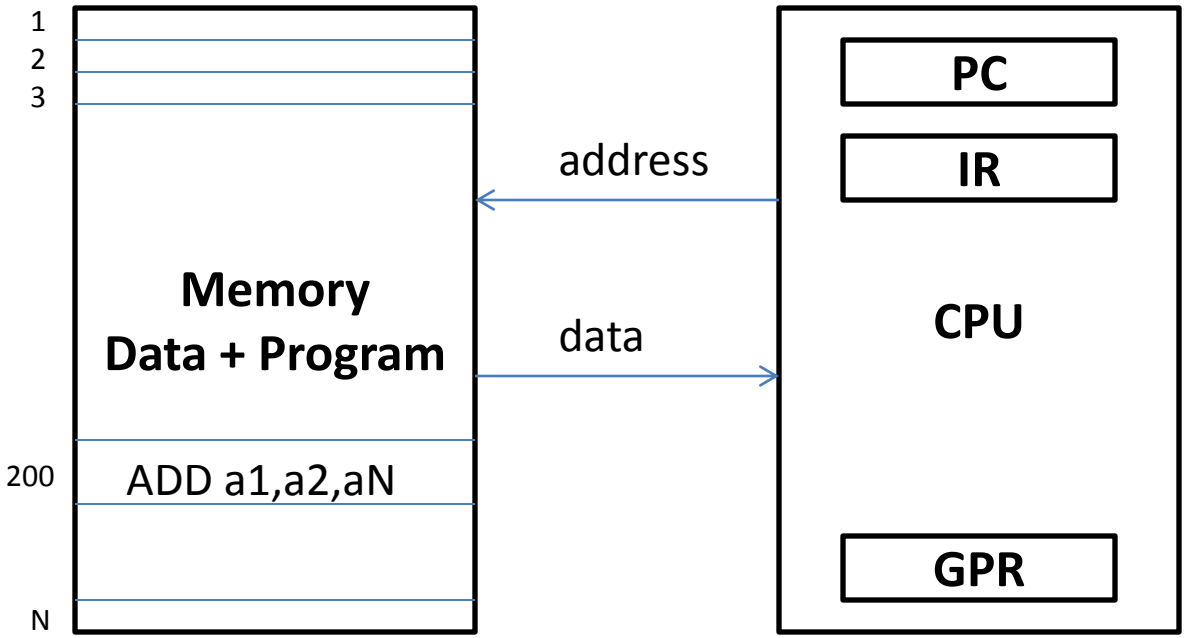
Outline

- Processor
  - Single-cycle datapath
  - Pipeline datapath
  - Processor types
- Memory
- I/O

General Purpose Processors (GPP)

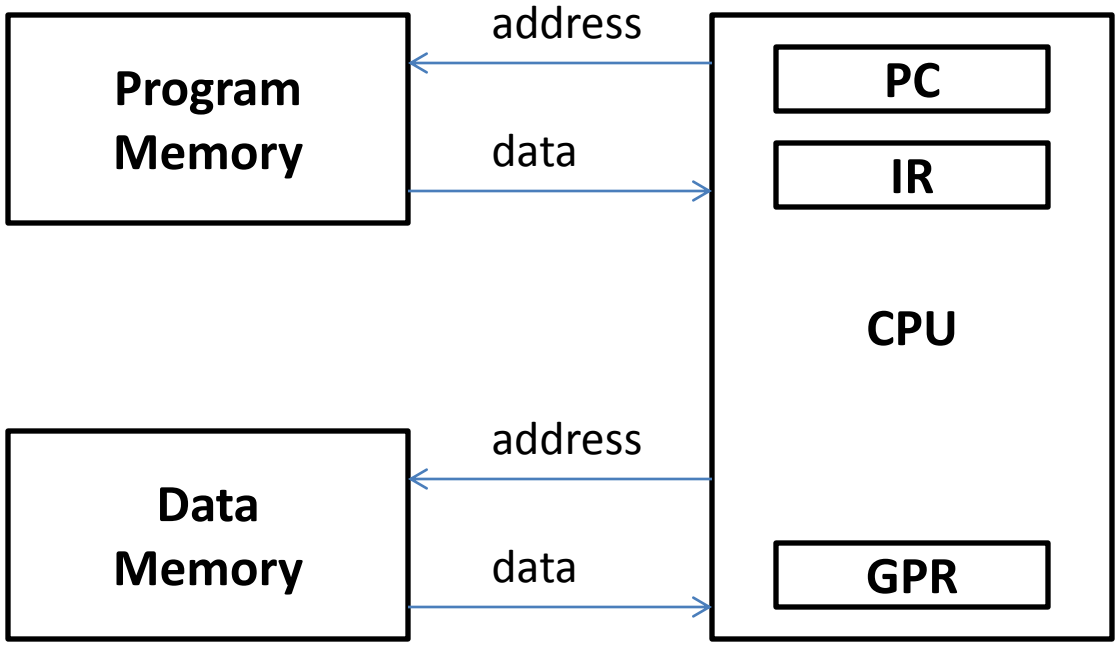
- High performance
  - Highly optimized circuits and technology
  - Use of parallelism
    - superscalar: dynamic scheduling of instructions
    - super-pipelining: instruction pipelining, branch prediction, speculation
  - complex memory hierarchy
- Not suited for real-time applications
  - Execution times are highly unpredictable because of intensive resource sharing and dynamic decisions
- Properties
  - Good average performance for large application mix
  - High power consumption

GPP + Memory (I): von Neumann Architecture

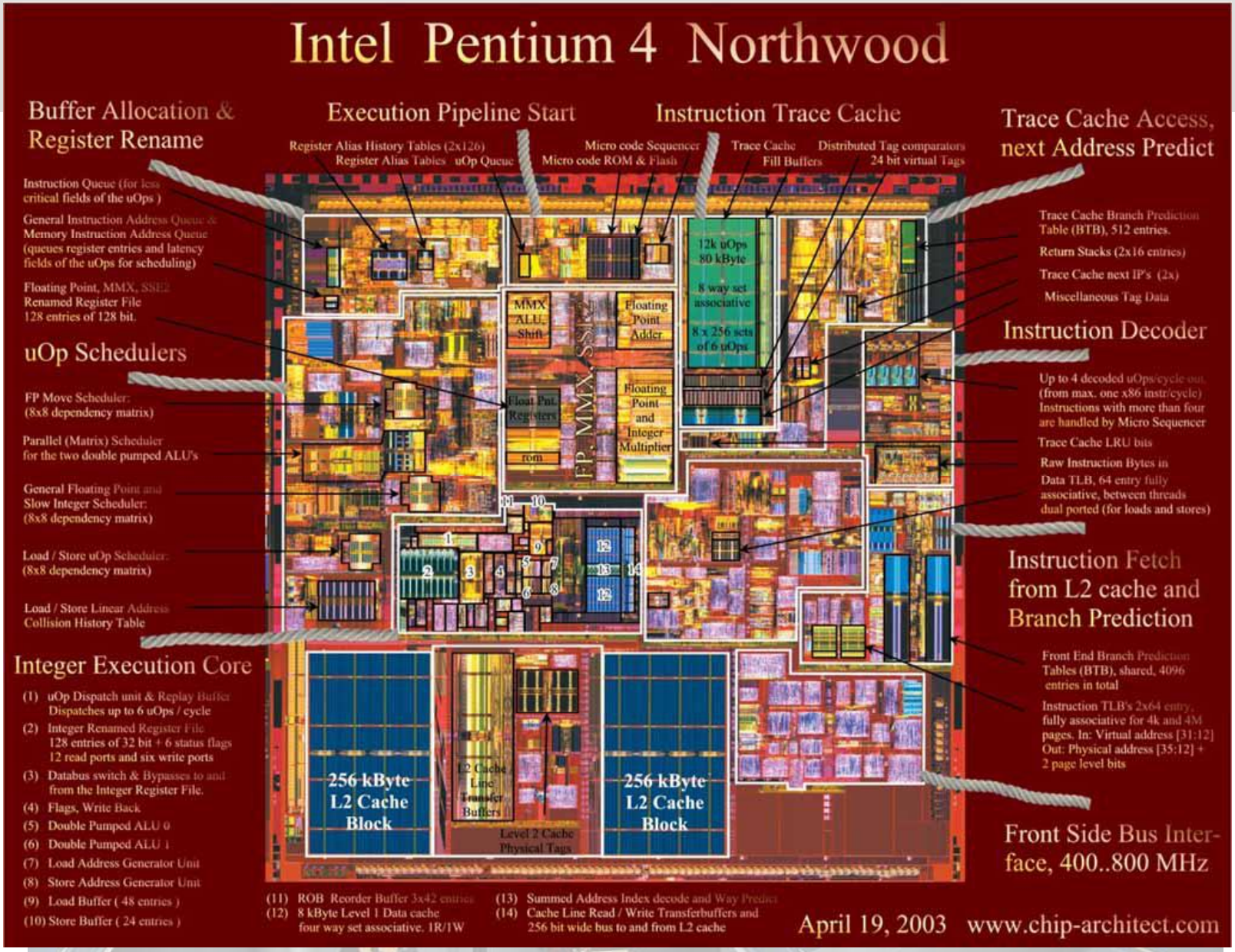


1. PC := 200
2. Fetch => IR := Mem[PC]
3. Decode IR
4. Execute
5. PC := PC + 1
6. goto 2

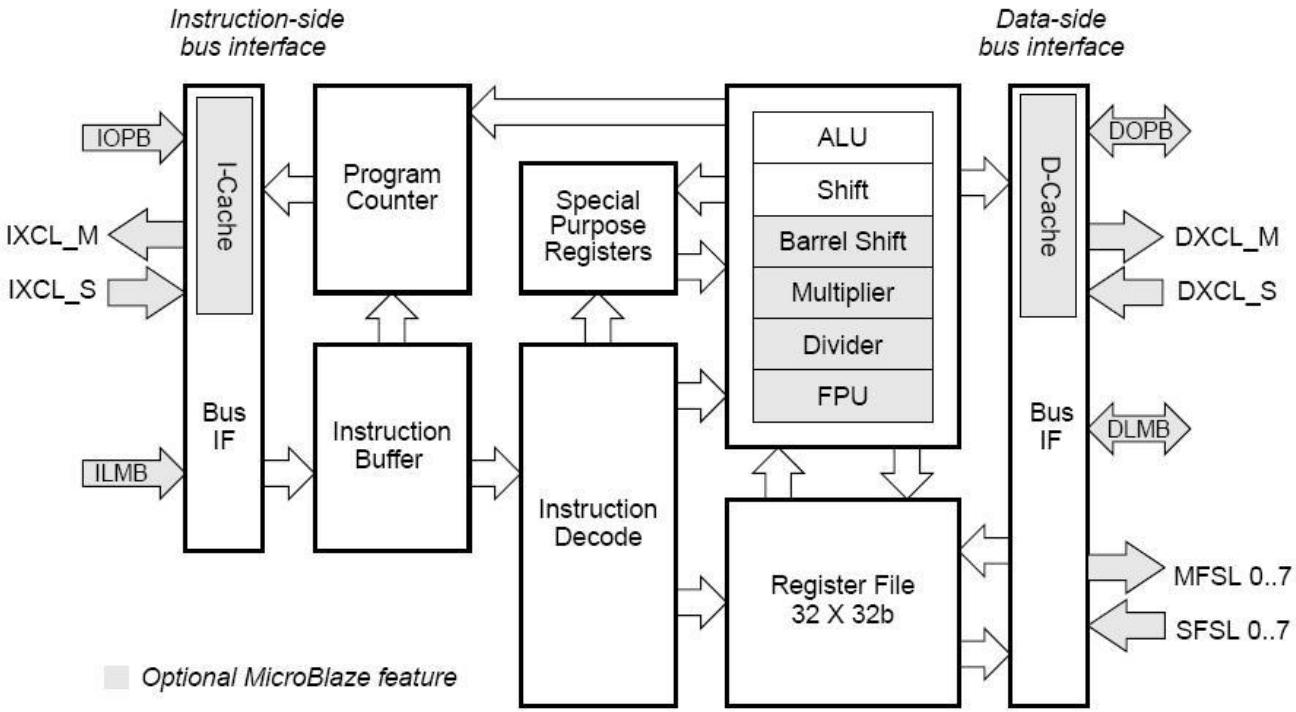
GPP + Memory (II): Harvard Architecture



Intel Pentium 4 Northwood



Simple GPP: Xilinx MicroBlaze



IOPB: Instruction side On chip Peripheral Bus
IXCL_M: Instruction-side Xilinx Cache Link Master
IXCL_S: Instruction-side Xilinx Cache Link Slave
ILMB: Instruction side Local Mory Bus

DOPB: Data side On chip Peripheral Bus
DXCL_M: Data-side Xilinx Cache Link Master
DXCL_S: Data-side Xilinx Cache Link Slave
DLMB: Data side Local Memory Bus
MFSL: Master Fast Simplex Link
SFSL: Slave Fast Simplex Link

Embedded Processors – RISC vs. CISC

- Complex instruction set CISC (e.g. x86)
  - Map complexity of common instructions directly in machine code
  - Complex instructions can consist of several simple instructions
  - Can lead to subtle timing issues
  - Used in general purpose computing

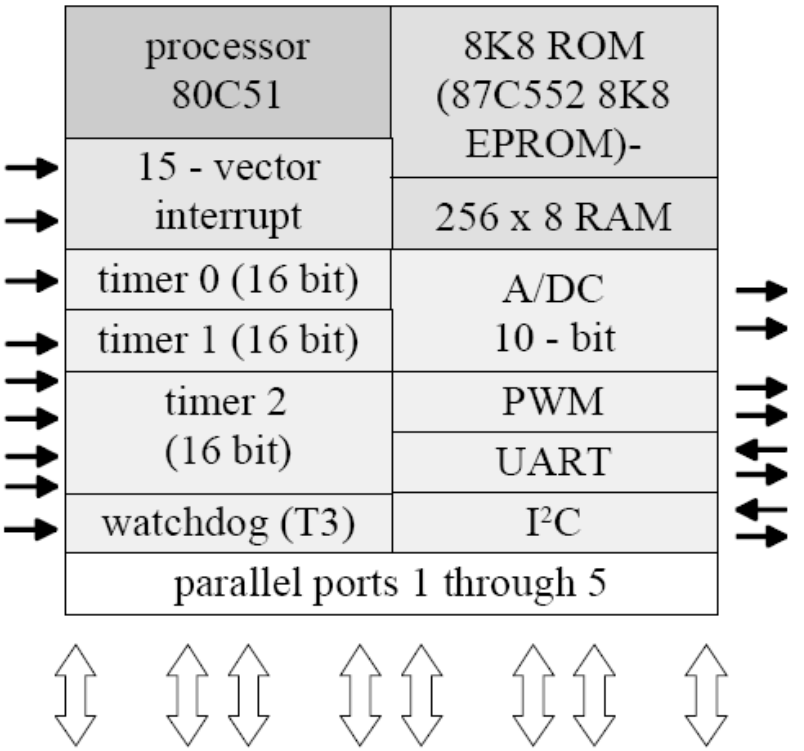
- Reduced instruction set RISC (e.g. ARM – Acorn Risc Machine)
  - Only simple machine instructions; Compiler has to map high-level language onto simple instructions
  - All instructions take the same time
  - Used in embedded systems (Real-time hardware, smart phones, …)

Application Specific Instruction Set Processors

- Micro Controllers (MicroCtrl)
  - Used in Control Dominated Systems
  - Reactive systems with event driven behavior
  - Application examples: cars, consumer electronics (washing machines, dishwashers etc.)
- Digital Signal Processors (DSPs)
  - Used in Data Dominated Systems
  - Streaming-oriented systems with mostly periodic behavior
  - Application examples: signal processing
- Very Long Instruction Word Processors (VLIWs)
  - Used in Data Dominated Systems
  - Application examples: image processing

ASIP: Micro Controllers

- Control-dominant applications
  - Supports process scheduling and synchronization
  - Preemption (interrupt), context switch
  - Short latency times
- Low power consumption
- Peripheral units often integrated
- Suited for real-time applications



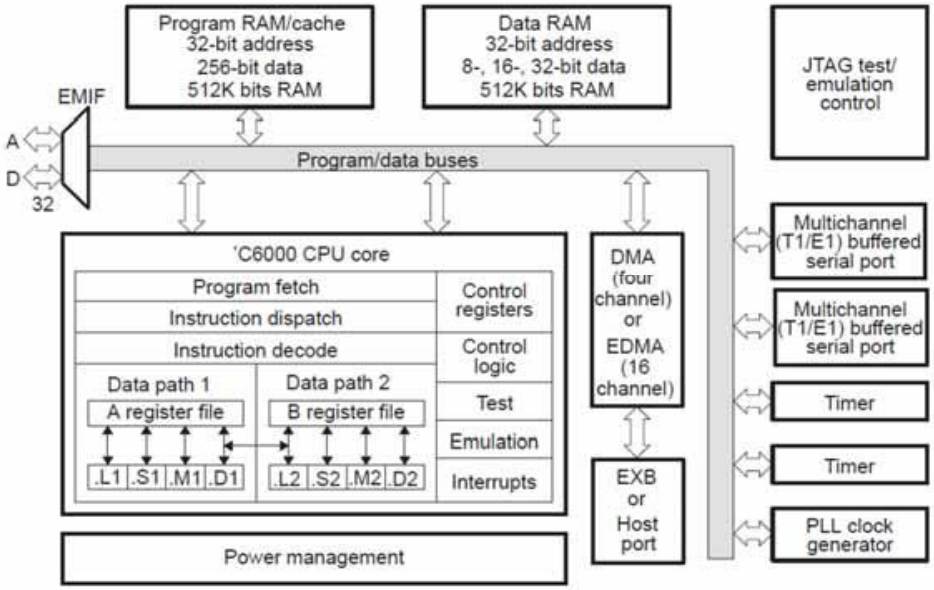
Philips 83 C552:
8 bit-8051 based microcontroller

ASIP: Digital Signal Processors

- Optimized for data-flow applications
- Suited for simple control flow
- Parallel hardware units
- Specialized instruction set
- High data throughput
- Zero-overhead loops
- Specialized memory

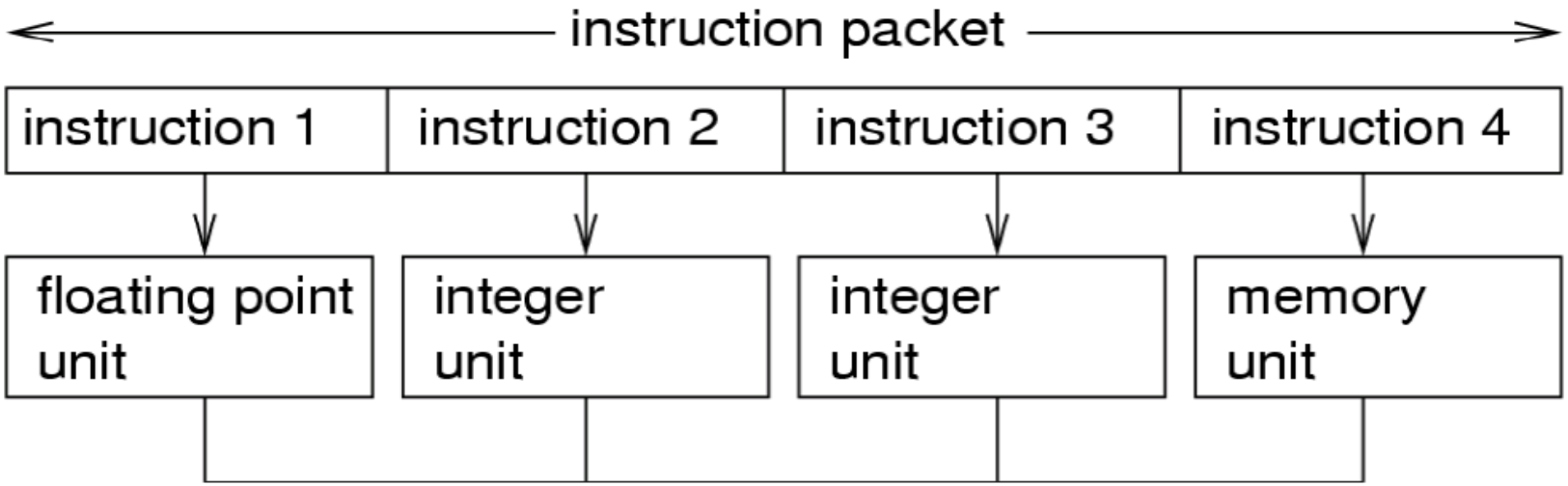
- Suited for real-time applications

Figure 2–1. TMS320C62x/C67x Block Diagram

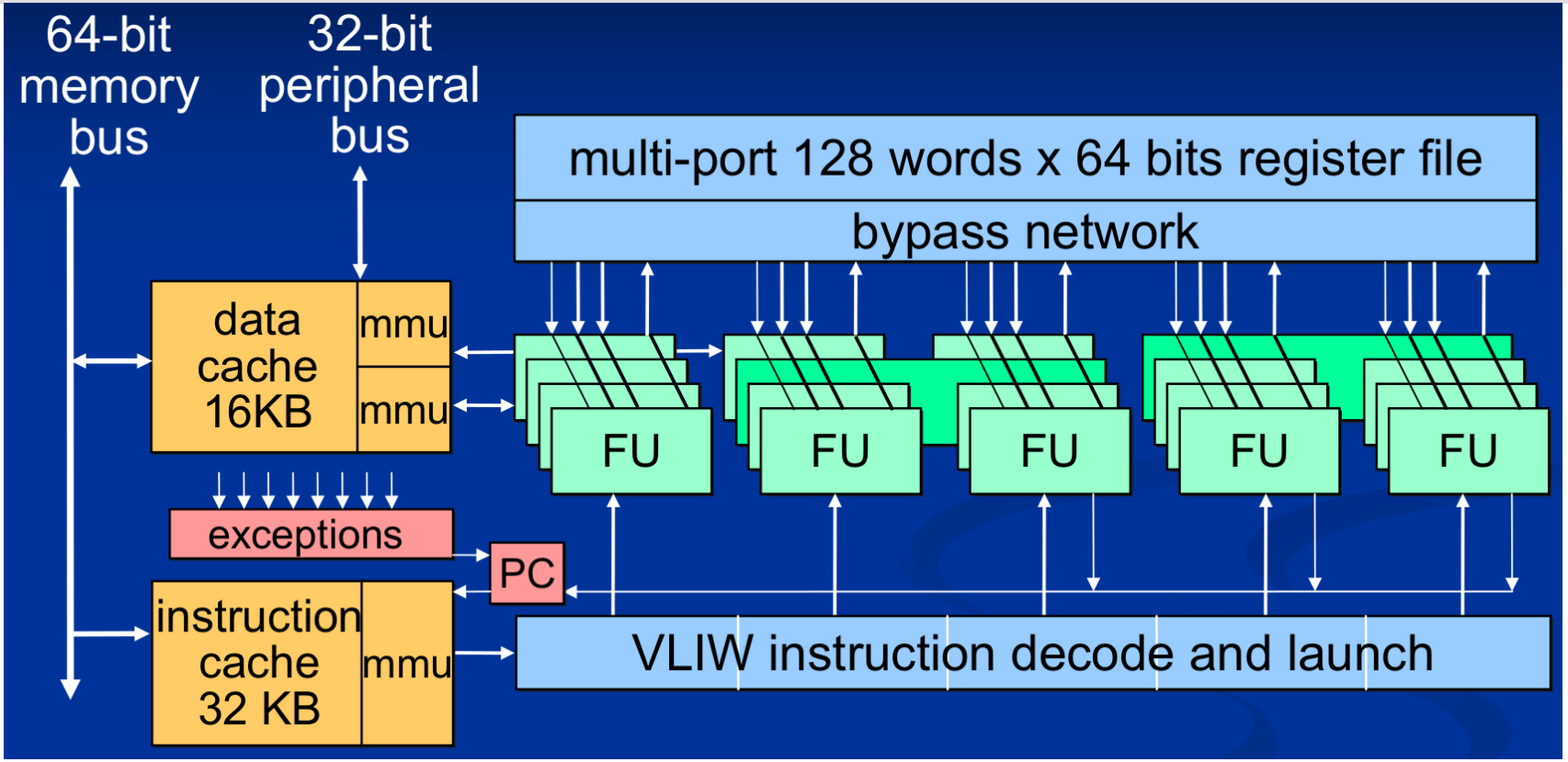


Very Long Instruction Word Processors

- Key idea: detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient).
- VLIW: parallel operations (instructions) encoded in one long word (instruction packet), each instruction controlling one functional unit.
- VLIW processors are an example of the so called Explicit Parallelism Instruction Computers (EPIC)



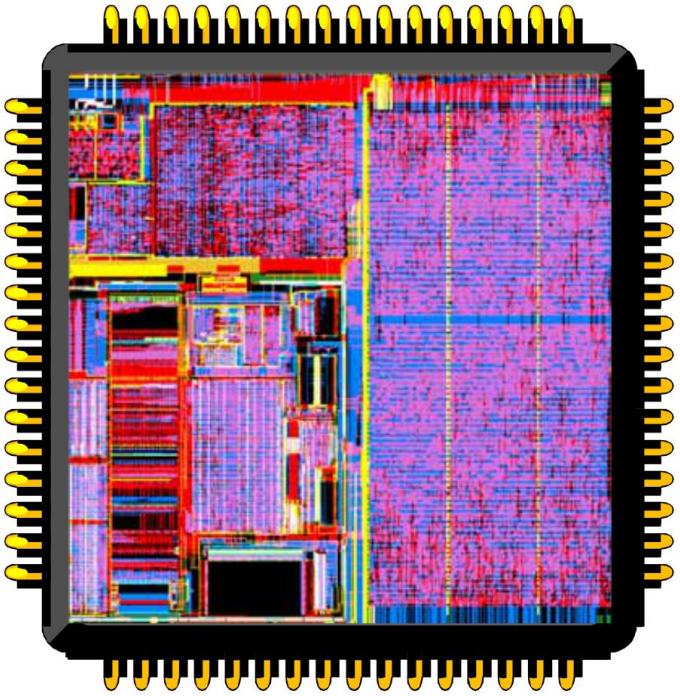
Philips TriMedia VLIW CPU



- 5 issue slots (functional units FU),
- therefore up to 5 instructions can be executed in parallel

Application Specific Integrated Circuits (ASICs)

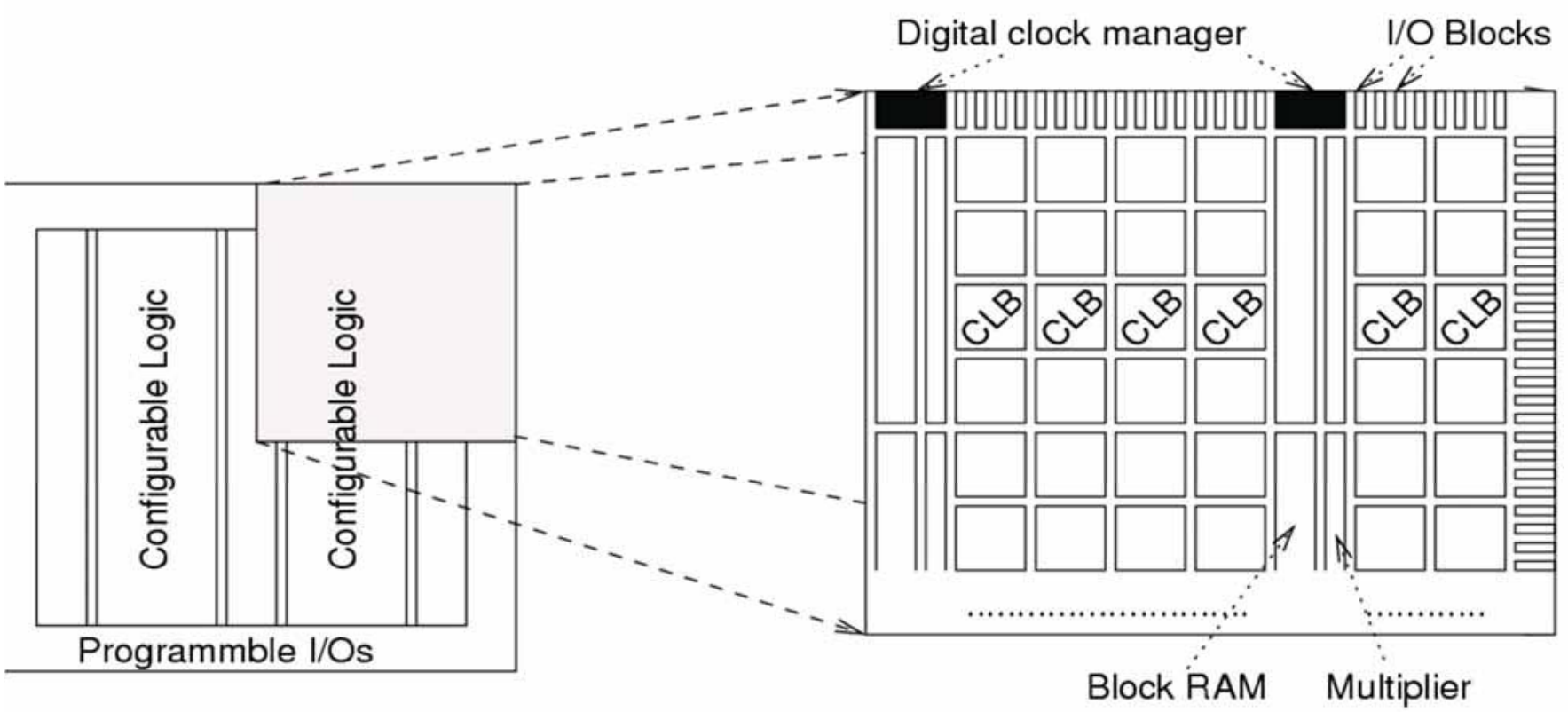
- Custom-designed circuits necessary
  - if ultimate speed or
  - energy efficiency is the goal and
  - large numbers can be sold.
- Approach suffers from
  - long design times,
  - lack of flexibility (changing standards)
  - high costs, i.e., Millions of $ mask costs



Reconfigurable Processing Units (RPUs)

- Full custom chips (HW) may be too expensive, software (SW) too slow.
- Combine the speed of HW with the flexibility of SW
  - HW with programmable functions and interconnect.
  - HW (Re-)Configurable at design-time or at run-time (dynamic reconfiguration)
- Field Programmable Gate Arrays (FPGAs)
  - Currently the most sophisticated and used RPUs
  - Applications
    - Fast and very cheap prototyping of (MP-)SoCs
    - Encryption,
    - Fast “object recognition“ (medical and military)
    - Adapting mobile phones to different standards
- Very popular devices from
  - XILINX (Virtex II(Pro), Virtex 4, Virtex 5, Virtex 6, Virtex 7)
  - Altera (Cyclone, Arria, Stratix)
  - Actel and others

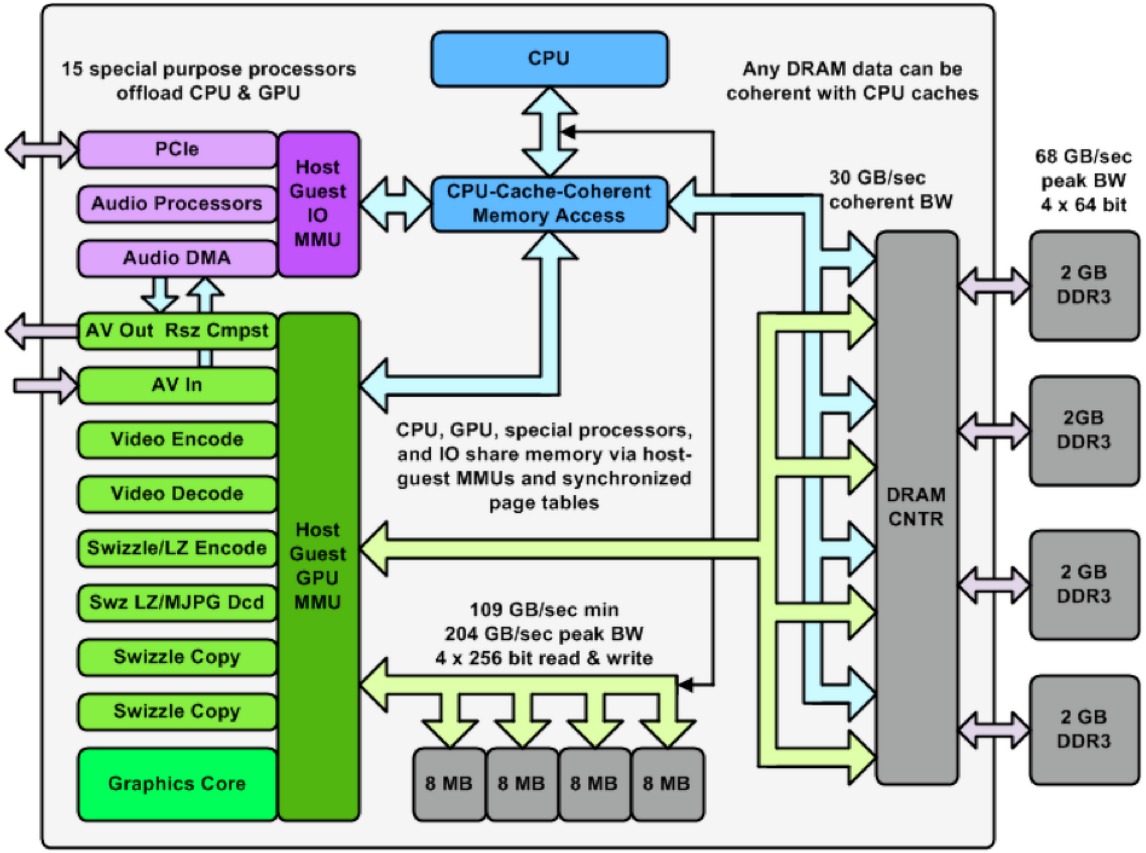
Floor-plan of VIRTEX II FPGAs



- Configurable Logic Block (CLB)
- Digital Clock Manager (DCM)
- Input/Output Blocks (IOB)

System-on-Chip (SoC)

SoC Components



XBOX

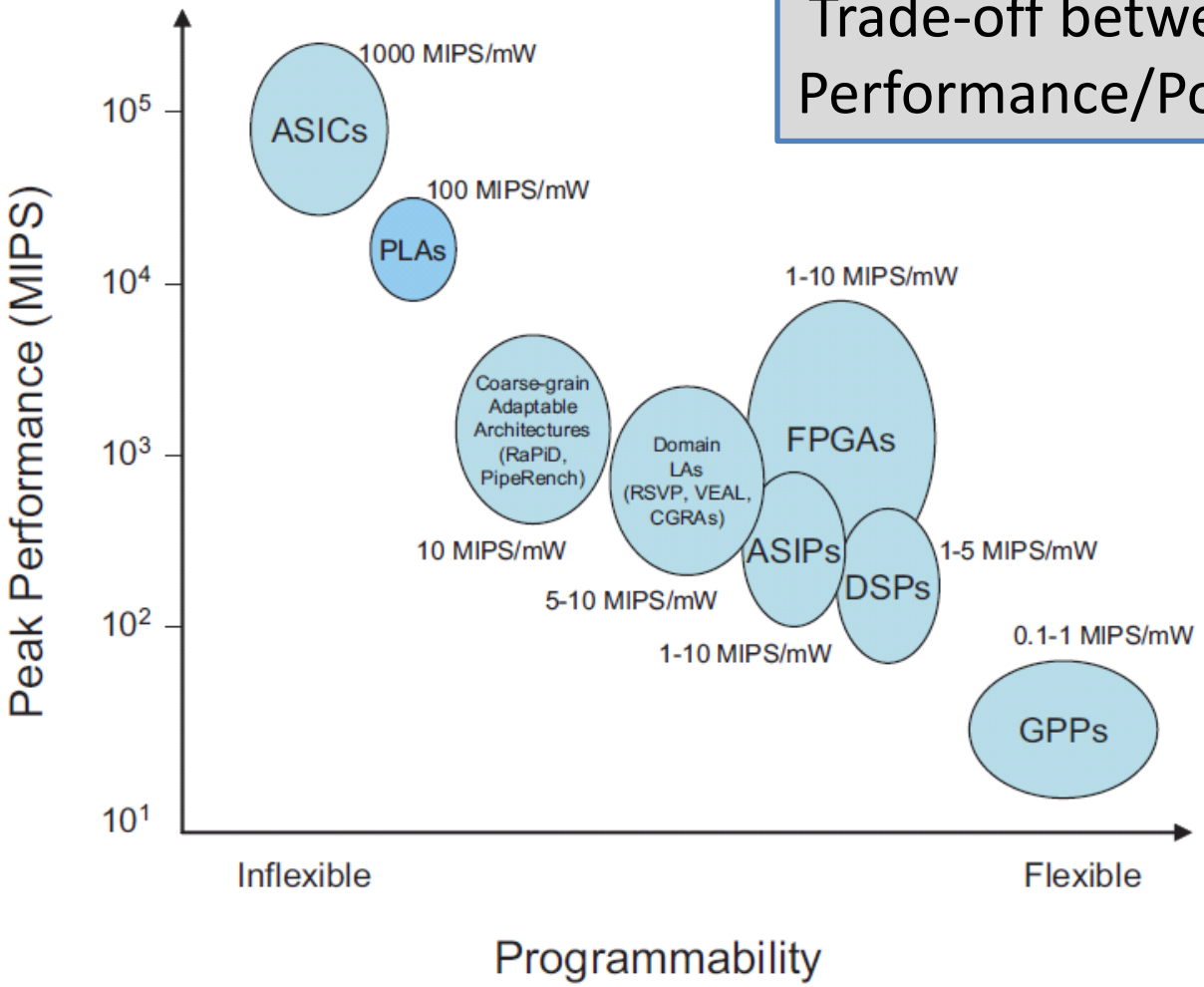
4

System Specialization

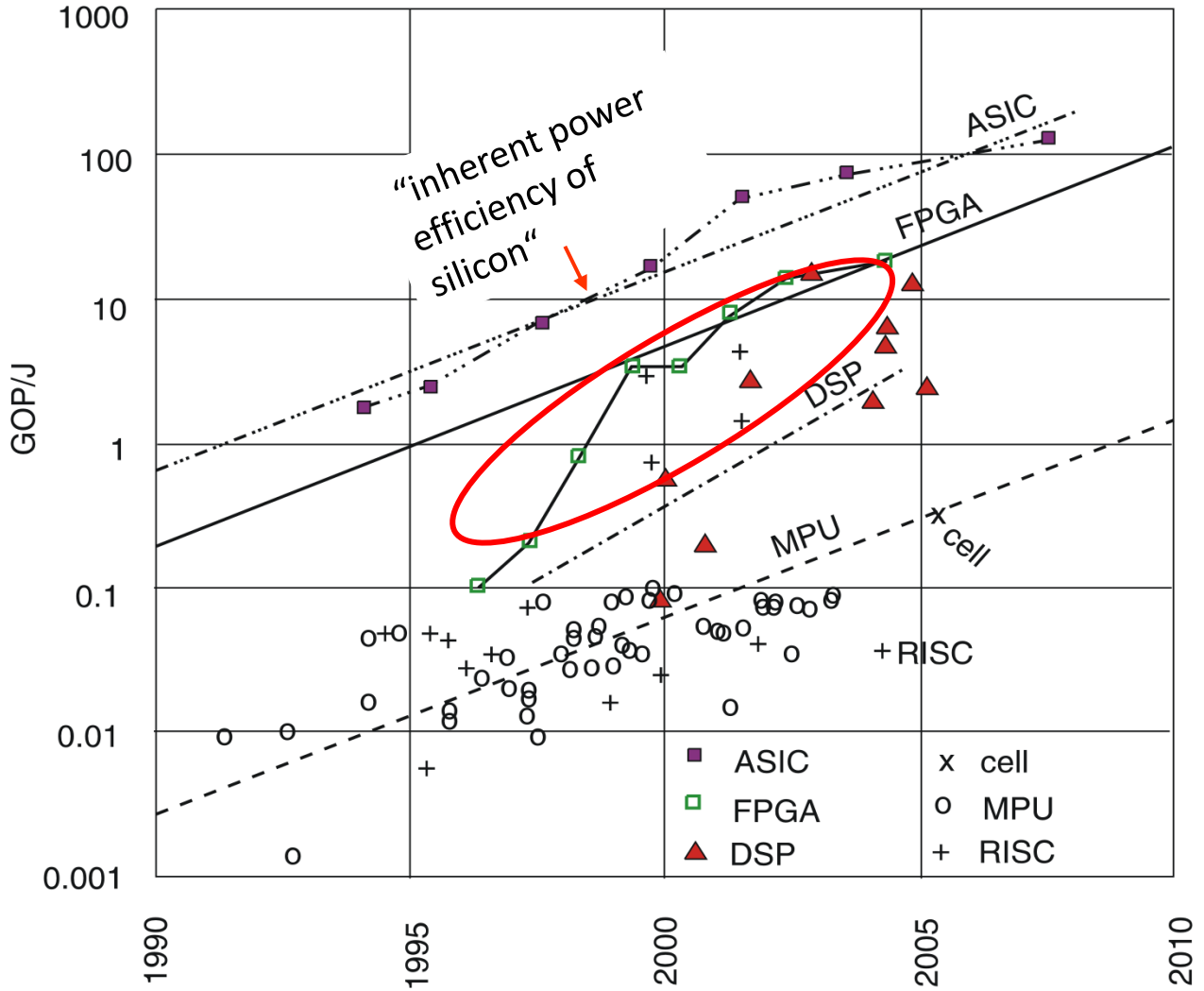
- The main difference between general purpose highest volume microprocessors and embedded systems is specialization.
- Specialization should respect flexibility
  - application domain specific systems shall cover a class of applications
  - some flexibility is required to account for late changes, debugging
- System analysis required
  - identification of application properties which can be used for specialization
  - quantification of individual specialization effects

Why Implementation Alternatives?

Trade-off between Flexibility and Performance/Power Efficiency



Energy Efficiency



© Hugo De Man, IMEC, Philips, 2007