Processor Datapath

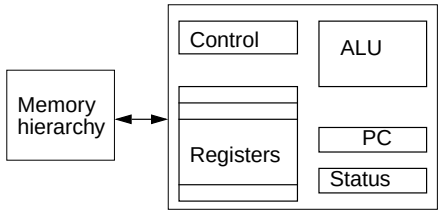
Levels in Processor Design

- We can talk about design at a variety of levels (from low to high):
  - Circuit design: transistors, resistors, capacitors, etc. Building gates, flip-flops, etc.
  - Logic design: putting gates (AND, OR, XOR, etc) and flip-flops together to build blocks such as registers, adders, memory. See CSE370.
  - Register transfer level: describes the execution of instructions by showing how information is transferred and manipulated between adders, registers, memory, etc.
  - Processor description: the ISA.
  - System description: includes memory hierarchy, IO, number of processors, etc.

Register Transfer Perspective

- We'll use either block diagrams or pseudocode to describe the operation/design of a simple processor which implements a subset of the MIPS ISA.
- We'll implement just a subset of the ISA:
  - Memory reference: lw and sw
  - Arithmetic: add, sub, and, or, stli
  - Control: beq, jump
- Key components:
  - Combinational: the output is a function of the inputs (e.g. an adder)
  - Sequential: state is remembered (e.g. a register)

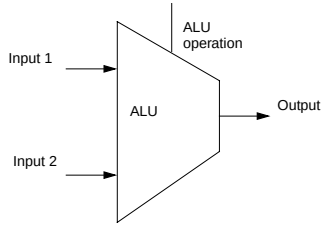
Data Path and Control Unit



- Data path:
  - *Combinational* (ALU) + *Sequential* (Registers, PC, Status)
  - How data moves between components, what operations are performed on data.
- Control unit:
  - Sends signals to data path elements
  - Tells what data to move, where to move it, what ops to perform

Combinational Elements: ALU

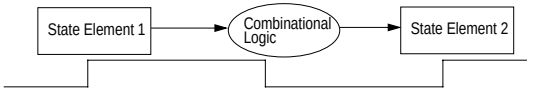
- ALU computes (combinational) output from its two inputs.
- Performs functions needed to execute arithmetic and logical instructions.
- Combinational logic has a “critical path” which determines the length of time needed for the output to stabilize given stable inputs. (These days: ~1ns).



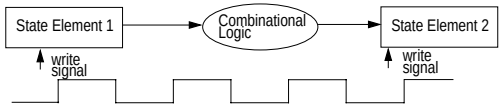
Synchronous Design

- Use a periodic clock, which controls when signals can be read and when they can be written. Values in storage elements can only be updated on clock edges.
- The clock determines when events occur, ie, when signals sent by control unit are obeyed in the datapath.

Changes occur on every clock edge:

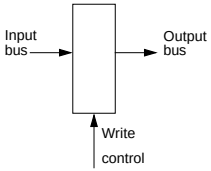


Changes occur on clock edges when a write signal is asserted (this allows combinational logic to take several cycles):



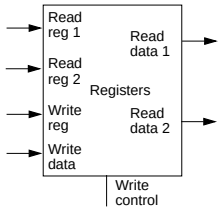
Building Blocks: Storage elements

- The basic building block is the register.
- Our registers store 32 bits.
- A register will only be written on the clock edge AND when the write control line is asserted.
- It can be read and written on the same clock, but the value read will be the OLD value.



Building Blocks: Register File

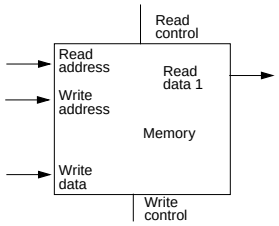
- Register file is an array of registers (32 in MIPS)
- ISA tells us that we should be able to read 2 registers and write 1 register in a given instruction.
- We need to know which registers to read/write, and what data to write.



- Typical access time is around 1ns.

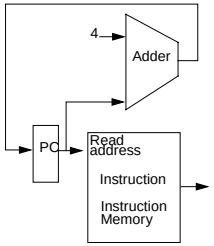
Memory

- Memory is like a register file, but much larger and slower.
- Can only read or write one location per cycle.



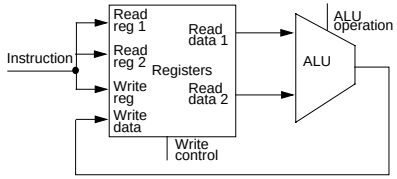
- Typical access time (for primary memory) is around 50ns. For cache memory it is closer to 5ns.

Instruction Fetch Datapath



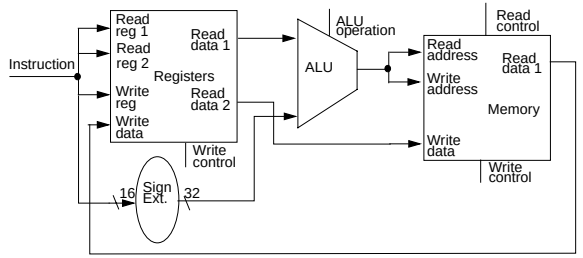
- Our implementation will fully execute one instruction per clock cycle: single cycle implementation.
- The PC tells us the read address.
- On each clock edge, a new value for PC will be latched into the PC register.

Datapath for R-type Instruction



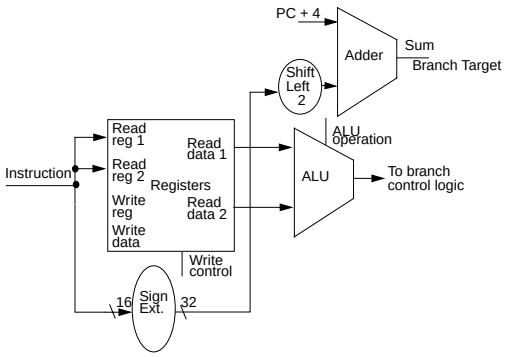
- The instruction bits name the read and write regs (rs, rt, rd).
- On the clock edge, the data is read, which moves through the ALU, hopefully in time to be latched into the write port at the next clock edge.

Datapath for Load/Store



- The instruction bits tell us the registers to use (src/dest and base register) and the 16 bit signed offset.
- We use the ALU to compute the effective address, which is passed along to the data memory.

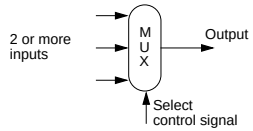
Datapath For Branch



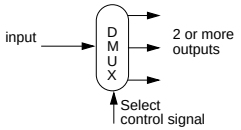
- Question: Why can't we just use the ALU to compute the branch target address?

Combinational Elements: (De)Multiplexor

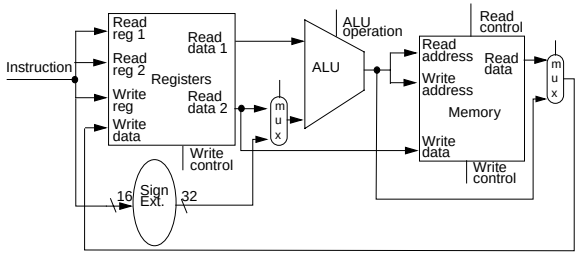
- Multiplexor (mux) selects the value of one of its inputs to be routed to the output:



- Demultiplexor routes its input to one of its outputs:

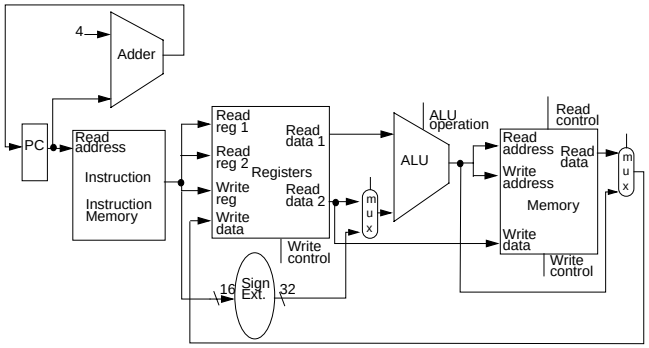


Combining Memory and R-type

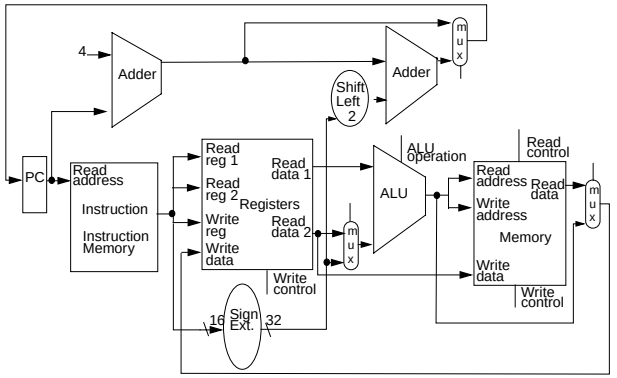


- Note that we add 2 muxes:
  - One to select the second ALU input
  - One to select the source for the register writeback (memory or ALU result)

Adding Instruction Fetch



Full Datapath: Adding Branches



Processor Control

Adding Control

- Control Unit:
  - Decodes instruciton opcode/function field
  - Sends signals to the data path (muxes, reg file, memories)
- Some controls come directly from the instruction:
  - Register fields indicate which register to read/write
  - Immediate field
- Building the control unit is not that complicated:
  - Input signals (opcode/function) are specified by the ISA
  - Output signals can be identified easily from the opcode
  - We can use PLAs (see CSE370) to build hardwired control units

Review of Instruction Format

- The opcode lives in bits 31-26
- The two registers to read are always the rs (25-21) and rt (20-16) registers
- For a load/store, we find the base register in the rs field (25-21)
- The 16 bit offset (for branch or load/store) is in 15-0
- The destination register can be in one of two places:
  - Loads: rt field (20-16)
  - R-type: rd field (15-11)
  - This implies we'll need a mux to select between these two fields.

Where are control signals needed?

- Register File:
  - *RegDst* - selects between rt and rd field as destination register (different for Load-store than R-type)
  - *RegWrite* - do we want to write a register? (R-type, Load-store)
- ALU:
  - *ALUSrc* - selects between immediate or register value as source for ALU (different for R-type and I-type)
  - Also need to select kind of ALU operation (bits 0-5 is function)
- Memory:
  - *MemWrite* - are we writing memory? (store instructions)
  - *MemRead* - are we reading memory? (load instructions)
  - *MemToReg* - selects between memory value or ALU output as writeback to the register file
- *Branch* - are we branching?

How are signals asserted?

- Control unit gets the opcode. Decoding yields:
  - Control for the 3 muxes (RegDst, ALUSrc, MemToReg)
  - Signals for read/write memory
  - Signal for register write
  - Signal for branch (ANDed with output of ALU)
  - Signal to ALU Control unit
- We also have a small control unit for the ALU, which takes the signal from the main control unit, together with the funct field (5-0) from the instruction.
- We'll focus on the main control unit. The ALU control is similar.

Examples

- Using the provided figure (5.22), fill in this table which specifies the control signals for the various instructions:

Instr.	opcode bits: 543210	Reg Dst	ALU Src	Mem to Reg	Reg Write	Mem Read	Mem Write	Br
R-format	000000							
lw	100011							
sw	101011							
beq	000100							

Control Functions

- We can express the functions for the control lines logically:
  - RegDst = !op5 AND !op2
  - ALUSrc = op5
  - MemtoReg = op5
  - RegWrite = !op2 AND !op3
  - MemRead = op5 AND !op3
  - MemWrite = op5 AND op3
  - Branch = op2
- For our subset instruction set, this minimized logic is fine, but for the full instruction set (64 opcodes), we'd want something more general: We would specify the truth table, and implement it using a PLA.