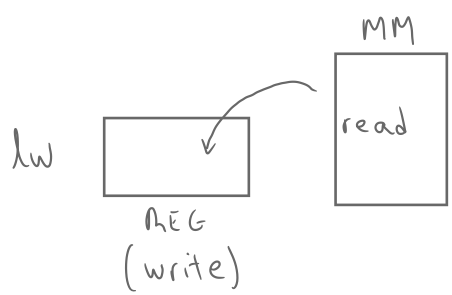
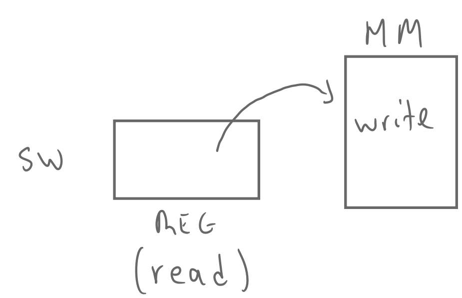
Execution Cycles

- Fetch instruction, update the PC register
  - Instruction memory (read only)
  - Get in with an address
  - The address is stored in the PC register
  - Get out with the instruction
  - PC register is updated [PC + 4]
- Decode
  - The instruction is parsed into fields (R-format has OPCODE 000000)
  - Get the source data
  - Register file (2 read and 1 write port)
  - * Must understand the implementation of the read and write ports (B56 on Patterson)
- Execute
  - ALU: compute logical and arithmetic operations
  - Output: result of the operation, it can represent target address (lw, sw)
    - check for zero
- Write back/memory access
  - Data memory or register file
  - Write-back: R-format
  - Memory access: lw and sw





From Patterson:

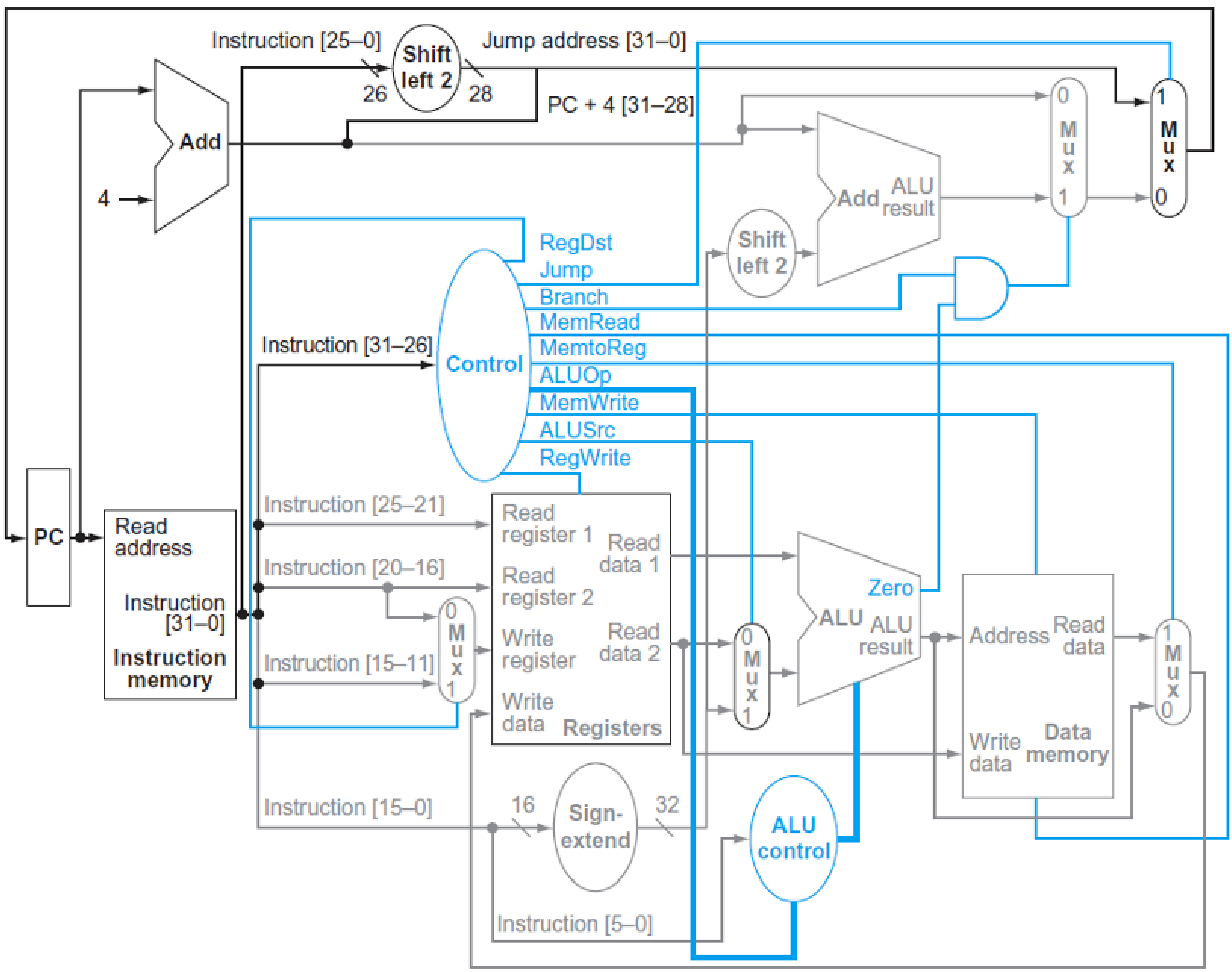
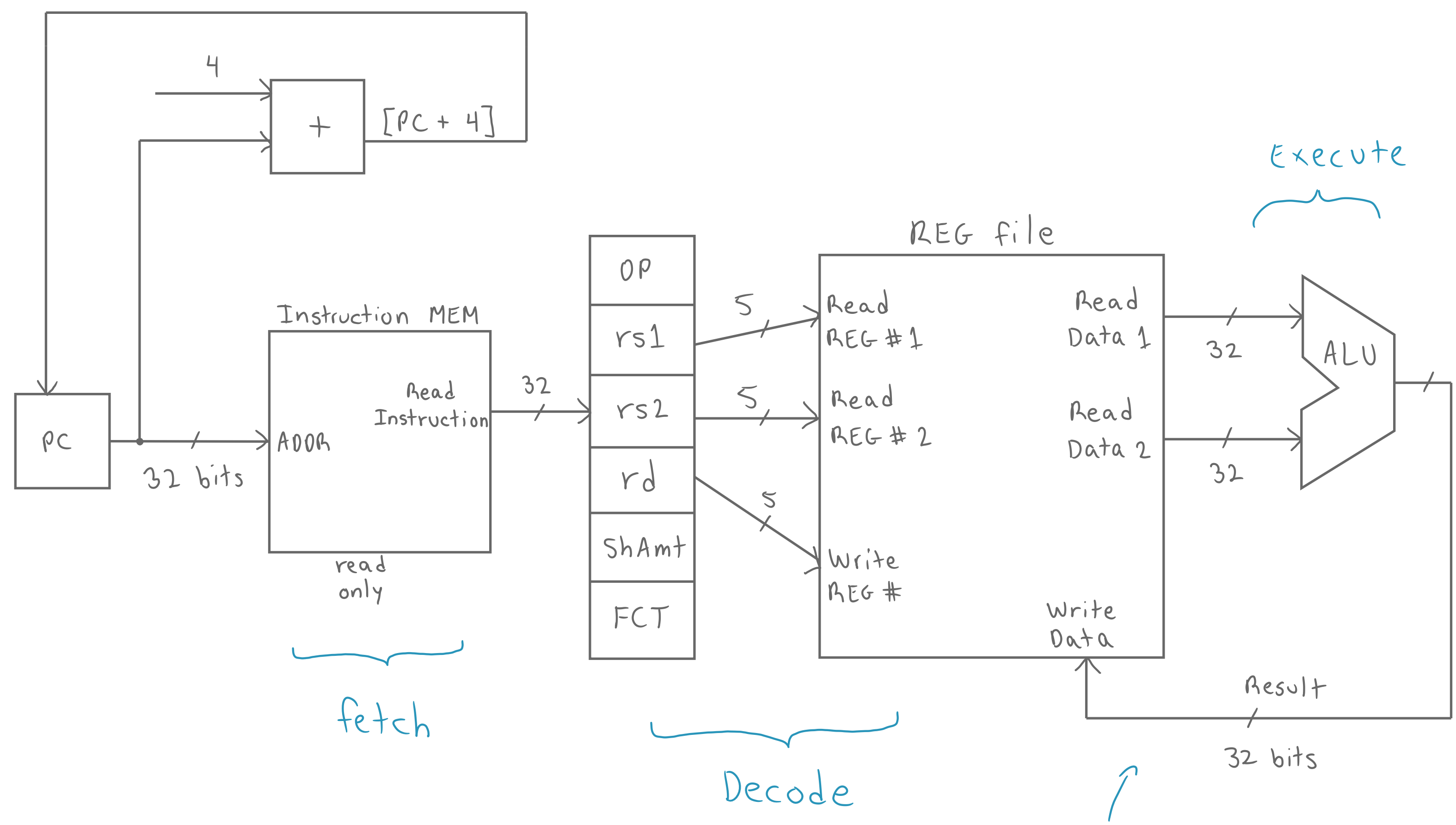


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

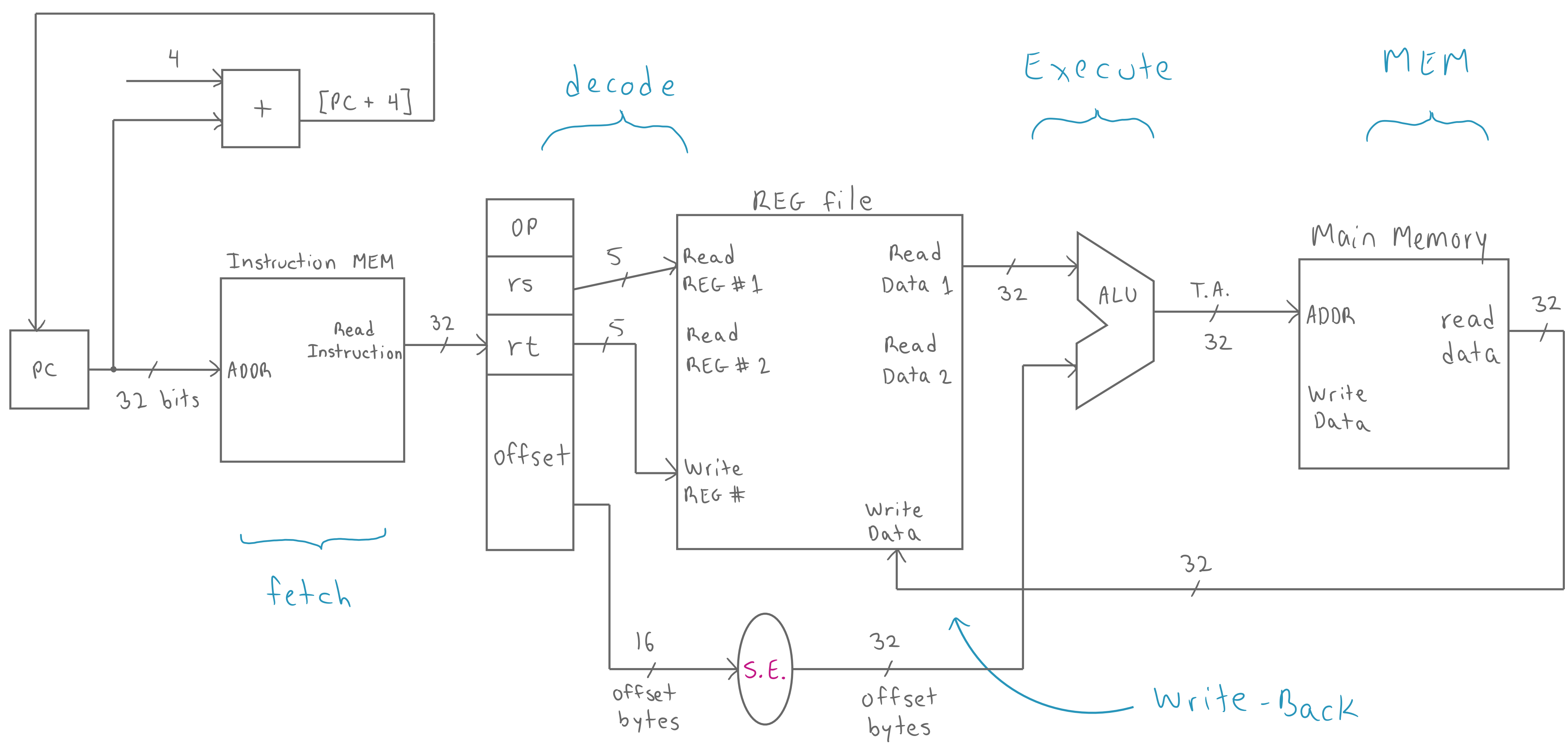
Datapath for R-format



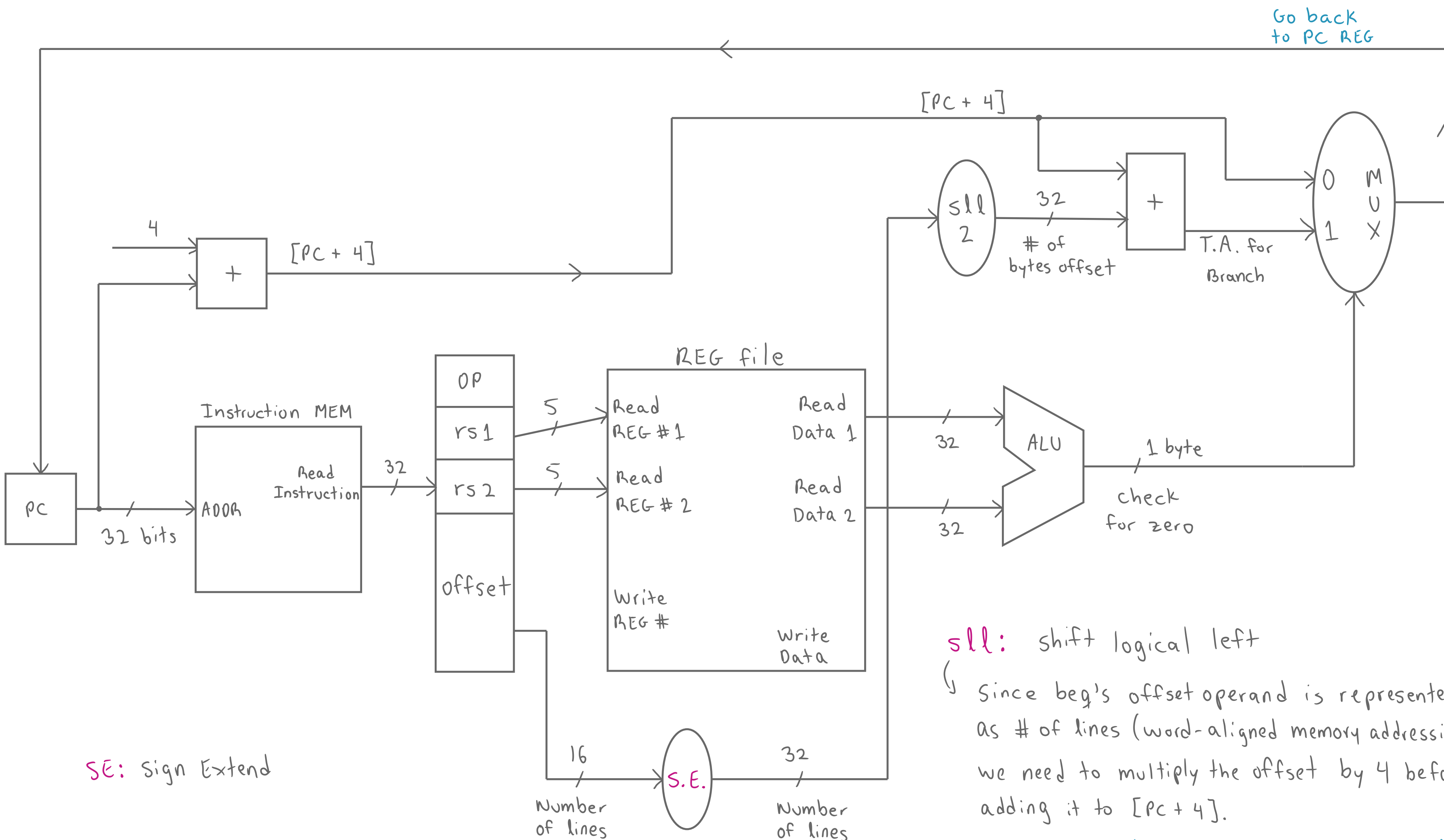
- rs1 → Read REG # 1:
  Read the value stored in register rs1
- rs2 → Read REG # 2:
  Read the value stored in register rs2
- rd → Write REG #:
  So that the correct Reg # is written to during the Write-Back phase

Datapath for lw (Load Word)

SE: Sign Extend



Datapath for BEQ



SE: Sign Extend

sll: shift logical left

Since beq's offset operand is represented as # of lines (word-aligned memory addressing), we need to multiply the offset by 4 before adding it to [PC + 4].

To achieve a times 4 operation, we can left shift the bits twice.

Register Field in Instructions

- First register (Read Register 1): Bits [25-21]
  - used by all instructions to read the first operand
- Second register (Read Register 2): Bits [20-16]
  - Used by:
    - R-format instructions (eg. add, sub)
    - store (sw) to get the value to write memory
    - branch (beq)

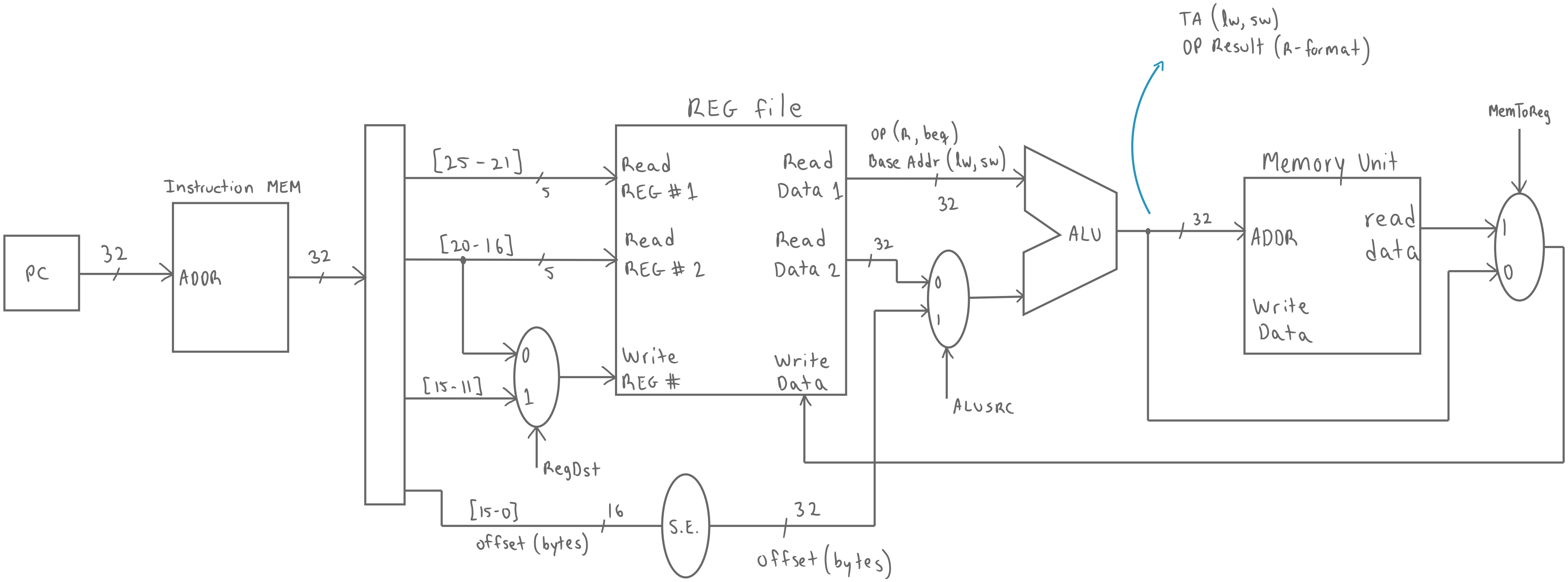
Write Registers

- Load instruction (lw)
  - Destination Register in bits [20-16]
- R-format
  - Destination Register in bits [15-11]

Exam 2 Announcement

- Full datapath diagram will be provided (from Patterson)
- Be able to draw the datapath for any individual instruction
- know what each multiplexer does, inputs/outputs, and control line behavior
- For individual datapaths, include only relevant parts, no unnecessary control lines
- Be able to trace PC updates through the nested multiplexers using control line values

Full Datapath With Control Lines (to accomodate different instructions)



Multiplexers Used as Control Units

R-format:

31 ⟷ 26	25 ⟷ 21	20 ⟷ 16	15 ⟷ 11	10 ⟷ 6	5 ⟷ 0
OP	RS	RT	RD	ShAmt	FCT

lw, sw, beq:

31 ⟷ 26	25 ⟷ 21	20 ⟷ 16	15 ⟷ 0
OP	RS	RT	offset

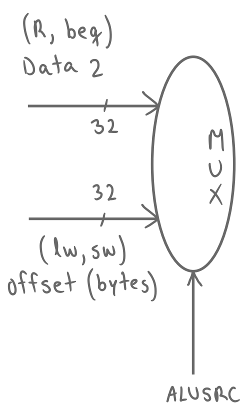
Bit ranges indicated

ALUSRC Mux

Controls what the second input of the ALU will be

R-format:] Data 2 obtained from
beq:] REG # in bits [20-16]

lw:] 32 bits
sw:] offset (bytes)

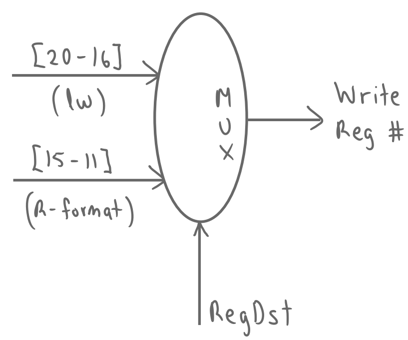


RegDst Mux

Chooses which Register gets written to.

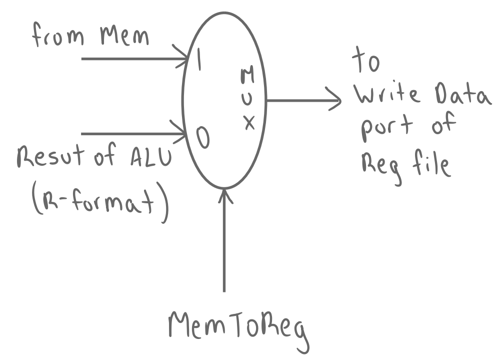
- In R-format, the result is stored into rd
- In lw, we load memory into rt

lw: Dest Reg in bits [20-16]
R-format: Dest Reg in bits [15-11]

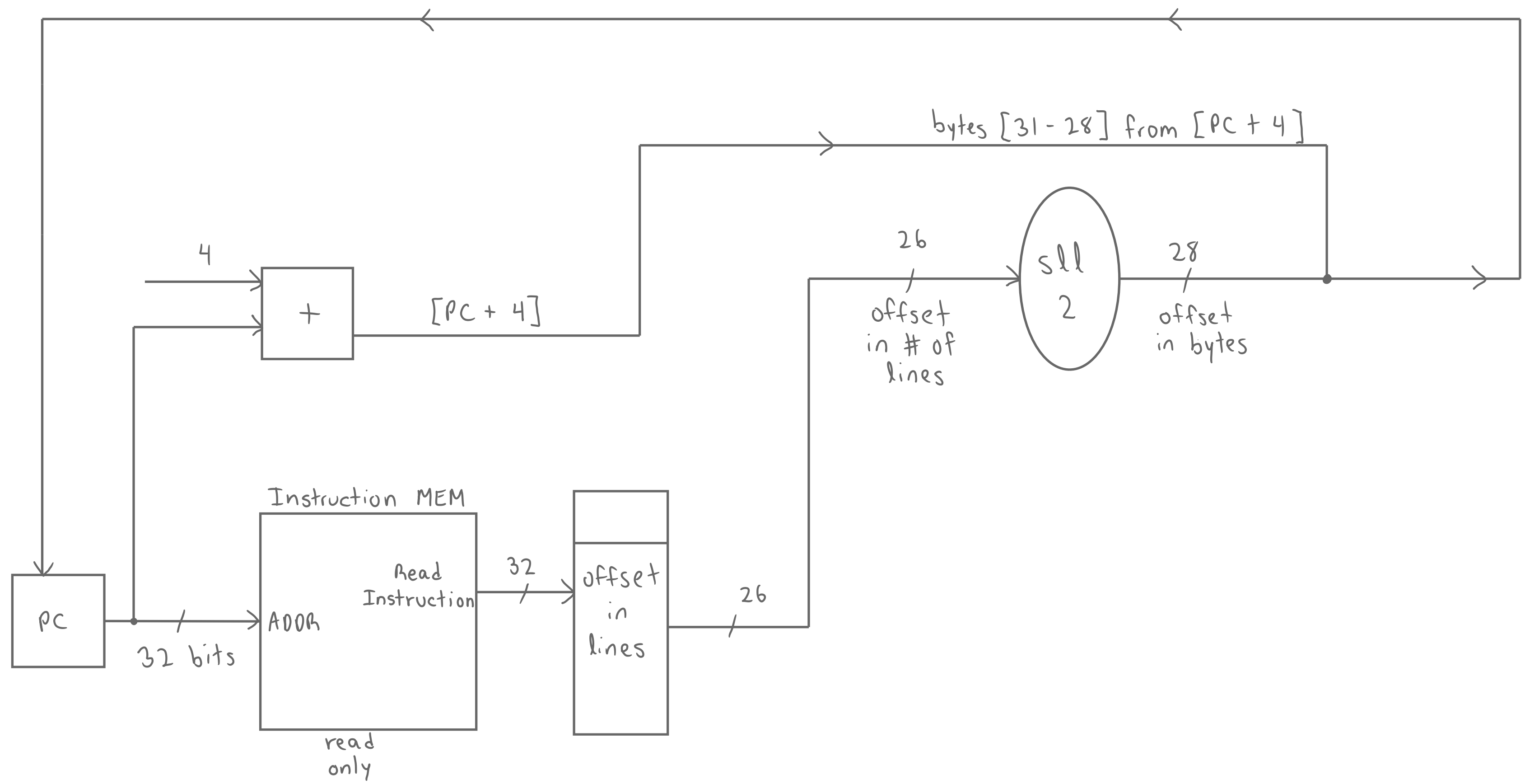


MemToReg Mux

- In lw, we write memory data into a register.
- In R-format, we write the ALU computation result into the register.

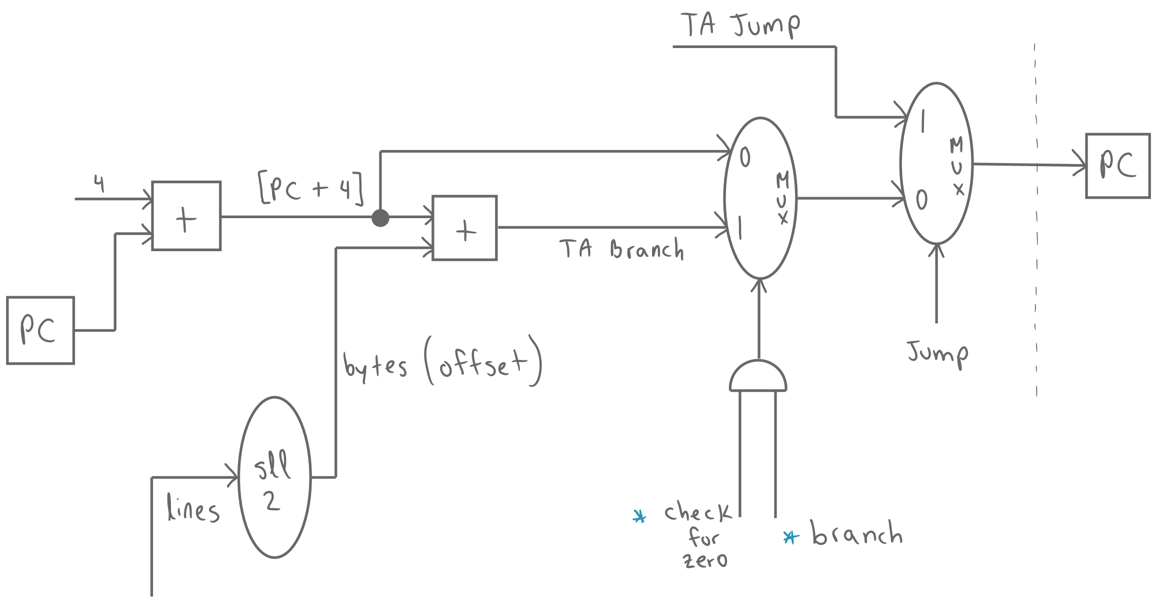


Jump



bytes [31 - 28] from [PC + 4]
4
+
[PC + 4]
26
offset
in # of
lines
sll
2
28
offset
in bytes
Instruction MEM
Read
Instruction
ADDR
read
only
32
offset
in
lines
26
PC
32 bits

Control Logic for Updating PC Register



- R-format, sw, lw:
  - Branch = 0
  - Jump = 0
  - We only update PC to [PC + 4]
- Branch Instruction:
  - Branch = 1
  - Jump = 0
  - We conditionally select either [PC + 4] or TA Branch depending on if check for zero is 0 or 1, respectively
- Jump Instruction:
  - Branch = 0
  - Jump = 1
  - We just update PC to TA Jump

* Branch (is_branch) signal is set to 1 when working with a branch instruction, otherwise 0

* The check for zero flag is set whenever the ALU is used for any reason, not just when evaluating a branch, so we need the AND gate in combination with the branch signal to together act as the control signal for the MUX.

From Patterson Appendix

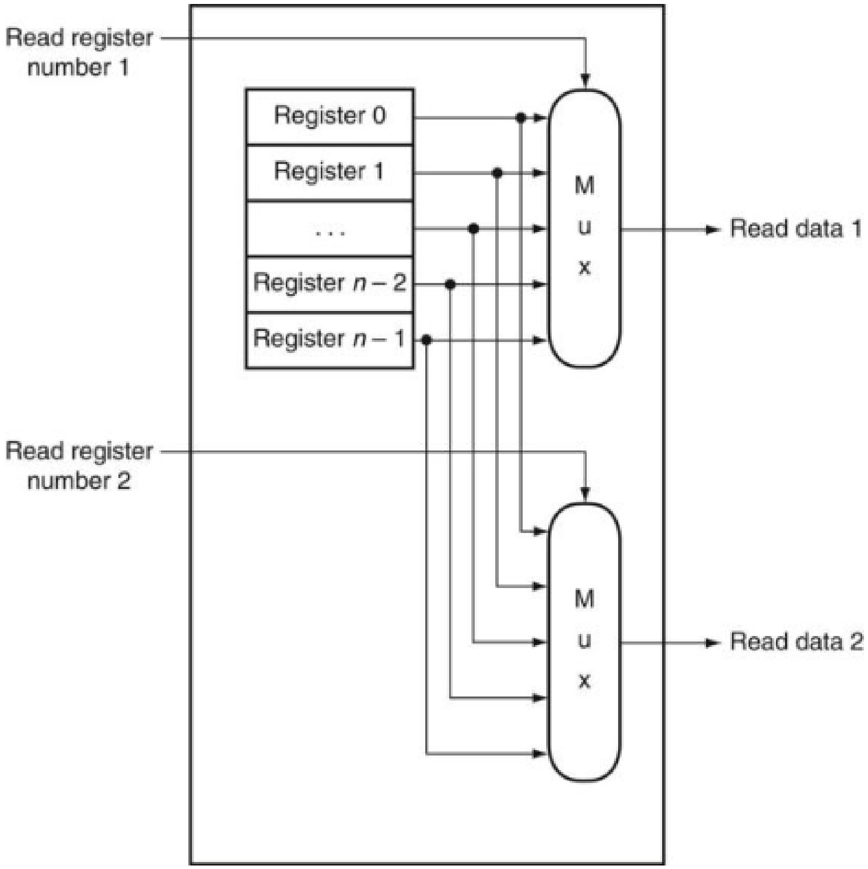


FIGURE B.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of n-to-1 multiplexors, each 32 bits wide.
The register read number signal is used as the multiplexor selector signal. Figure B.8.9 shows how the write port is implemented.

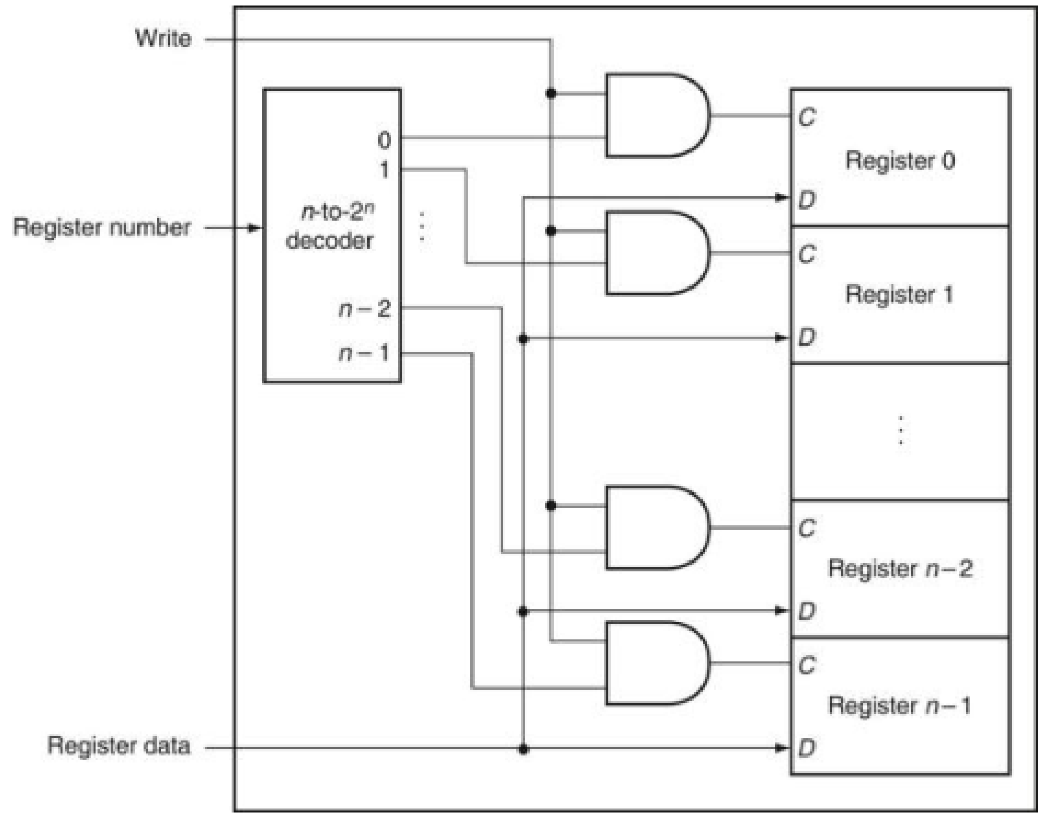


FIGURE B.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers.
All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.

Datapath Control Signals

The control signal receives the opcode (bits 31-26 of the instruction) and sets all relevant control lines to drive the datapath to perform the desired operation. These lines influence components such as multiplexers, the ALU, and memory units.

Control Signals by Instruction type

Each instruction type (R-format, load, store, branch, jump) triggers specific settings in the control lines.

Mux
- RegDst
- MemToReg
- ALUSRC
- Jump
- Branch

Access
- RegWrite

Memory
- MemWrite
- MemRead

	R-format	Load	Store	Branch	Jump
RegDst	1	0	X	X	X
MemToReg	0	1	X	X	X
ALUSRC	0	1	1	0	X
Branch	0	0	0	1	0
Jump	0	0	0	0	1
RegWrite	1	1	0	0	0
MemWrite	0	0	1	0	0
MemRead	0	1	0	0	0

Note:

- X indicates that the value is irrelevant for that instruction.
- RegDst determines the destination register (R-format uses rd, load uses rt)
- MemToReg selects source for register write (ALU result or memory)
- ALUSrc selects second ALU operand (register read vs. immediate)

Single Cycle Implementation: Performance

Fixed Clock Cycle (single cycle CPU) is an approach used to evaluate instruction timing

- the entire instruction executes in one clock cycle. → CPI = 1
- clock cycle is determined by the longest instruction path
- once determined, it is fixed for all instructions

Component Delays (Given in picoseconds)

- Memory Unit: 200 ps
- ALU, Adders: 100 ps
- Register file: 50 ps

* the adder for the PC is not considered to take any time

R-format

fetch: Read Instruction memory ⇒ 200 ps

Decode: Read Reg file (read 2 source reg in parallel) ⇒ 50 ps

Compute: Access ALU ⇒ 100 ps

Write Back: Write into the Reg file ⇒ 50 ps

400 ps

Store

fetch: Read Instruction memory ⇒ 200 ps

Decode: Read Reg file (read 2 source reg in parallel) ⇒ 50 ps

Compute: Access ALU (compute target Address) ⇒ 100 ps

Memory Access (write data to memory) ⇒ 200 ps

550 ps

Load

fetch: Read Instruction memory ⇒ 200 ps

Decode: Read Reg file (reads only 1 source Address) ⇒ 50 ps

Compute: Access ALU (compute target Address) ⇒ 100 ps.

Memory Access (read data from memory) ⇒ 200 ps

Write Back (write to Reg file) ⇒ 50 ps

600 ps

Beq

fetch: Read Instruction memory ⇒ 200 ps

Decode: Read Reg file (read 2 source reg in parallel) ⇒ 50 ps

Compute: Access ALU (check for zero)

Done in Parallel ⇒ 100 ps

Compute: Target Address (use an Adder)

350 ps

* Since the longest instruction is that of lw (600 ps), then that means that a program of 50 instructions, regardless of type, would take 50 × 600 ps to complete

Inefficiencies and the motivation for Pipelining

Using a fixed-length clock cycle based on the slowest instruction introduces inefficiencies. Instructions that require less time (like R-format or Branch) are forced to wait for the full cycle length of the longest instruction (eg. load).

This means a lot of clock time is wasted on idle components during the execution of faster components.

Pipelining addresses this by overlapping instruction stages. Instead of executing one instruction at a time from start to finish, pipelining divides execution into stages (eg. fetch, decode, execute, memory, write-back), allowing multiple instructions to be in different stages simultaneously. this boosts throughput and better utilizes hardware resources.