Prosty procesor dla framgentu listy rozkazów MIPSa

(rysunki pochodzą z książki Hennessy'ego i Pattersona)

Wstęp

Naszym celem będzie zaprojektowanie prostego procesora realizującego fragment listy rozkazów MIPS:

- rozkazy komunikacji z pamięcią: `lw, sw`
- rozkazy arytmetyczno-logiczne: `add, sub, and, or, slt`
- skoki `beq, j`

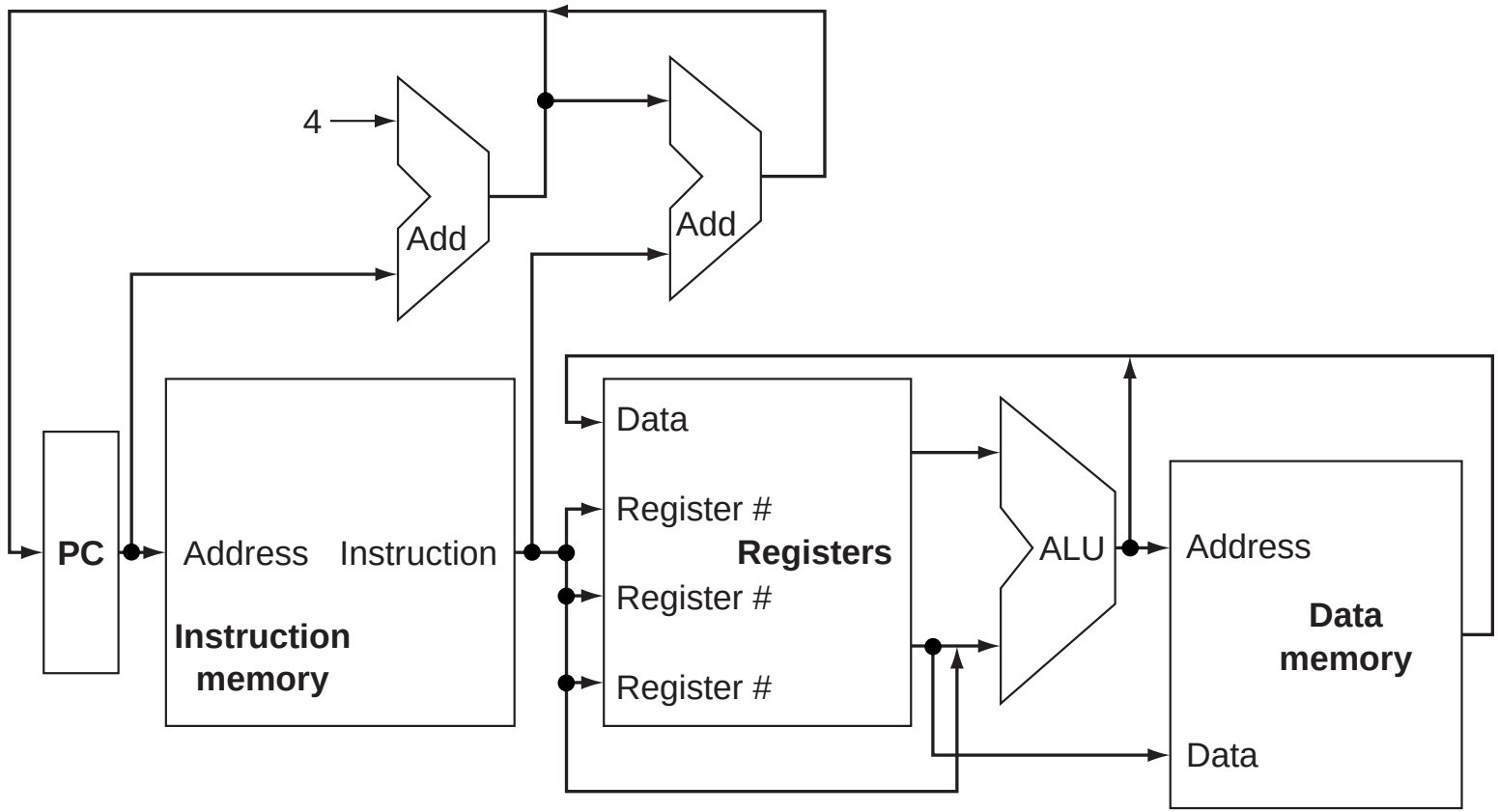
Obejrzymy trzy rozwiązania:

- Implementacja jednocyklowa
- Implementacja wielocyklowa
- Przetwarzanie potokowe

Implementacja jednocyklowa – wstęp

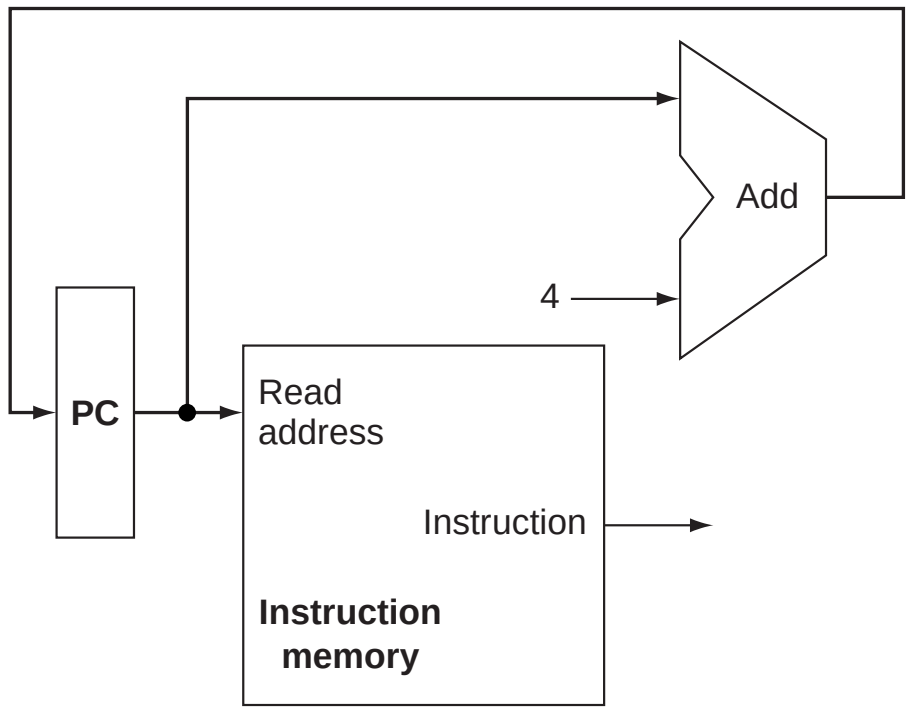
- Wykonanie każdego z naszych rozkazów zaczyna się tak samo:
  - Wysłanie licznika rozkazów (PC) do jednostki pamięci i odczytanie stamtąd odpowiedniego rozkazu.
  - Odczytanie wartości jednego lub dwóch rejestrów (specyfikowanych przez pięciobitowe pola rozkazu)
- Dalsze czynności zależą od rozkazu, chociaż są one podobne, szczególnie dla rozkazów z jednej grupy. Każdy rozkaz (oprócz `j`) użyje np. ALU.
- W naszej pierwszej implementacji wykonanie każdego rozkazu będzie zajmowało jeden cykl zegarowy.
- Zapis (do pamięci lub do rejestrów) następuje raz w całym cyklu (na zboczu opadającym).
- Odczyt jest możliwy w każdym momencie.

Ogólny schemat połączeń

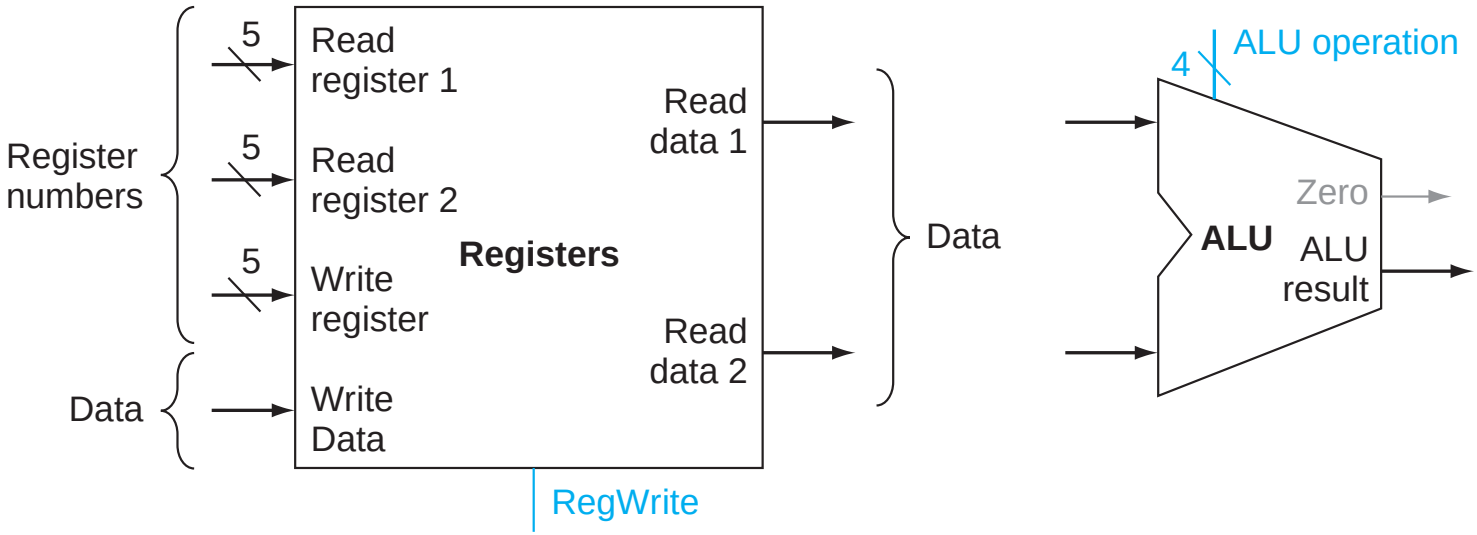


Pobranie rozkazu

Zakładamy, że mamy dwie jednostki pamięci: osobną dla rozkazów (*instruction memory*) oraz dla danych (*data memory*).



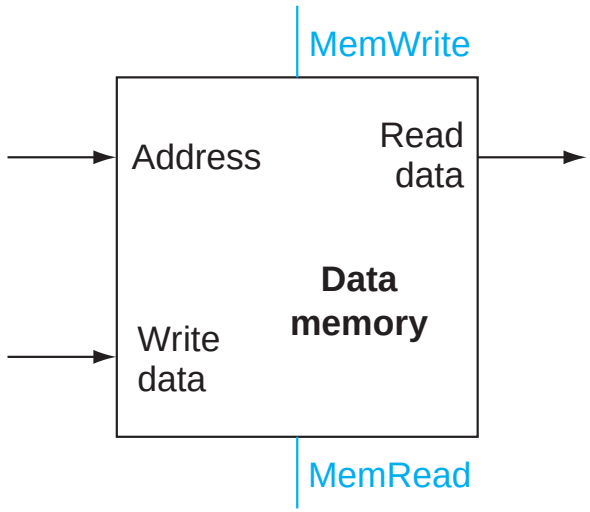
Rejestry i ALU



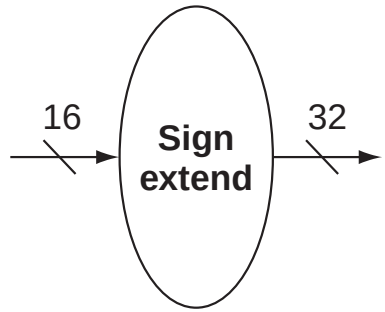
a. Registers

b. ALU

Pamięć danych i rozszerzanie danych



a. Data memory unit

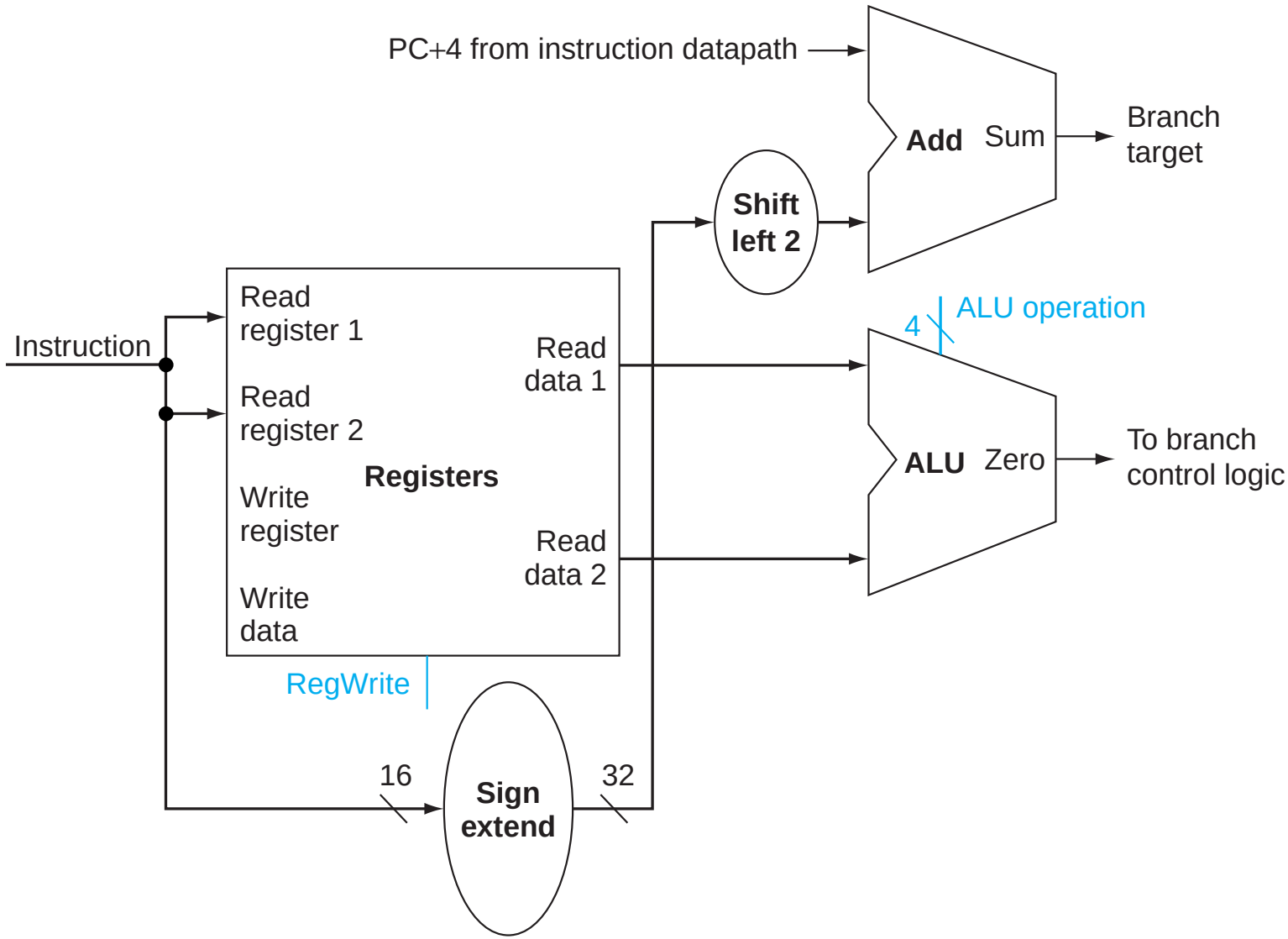


b. Sign-extension unit

Skok warunkowy

- Wykonanie rozkazu skoku warunkowego polega na:
  - Odczytaniu dwóch rejestrów
  - Porównaniu ich przez ALU (operacją odejmowania)
  - Przekształceniu stałej: rozszerzenie stałej 16-bitowej do 32-bitowej i przesunięcie o dwa bity w lewo
  - Dodaniu stałej do PC+4

Skok warunkowy

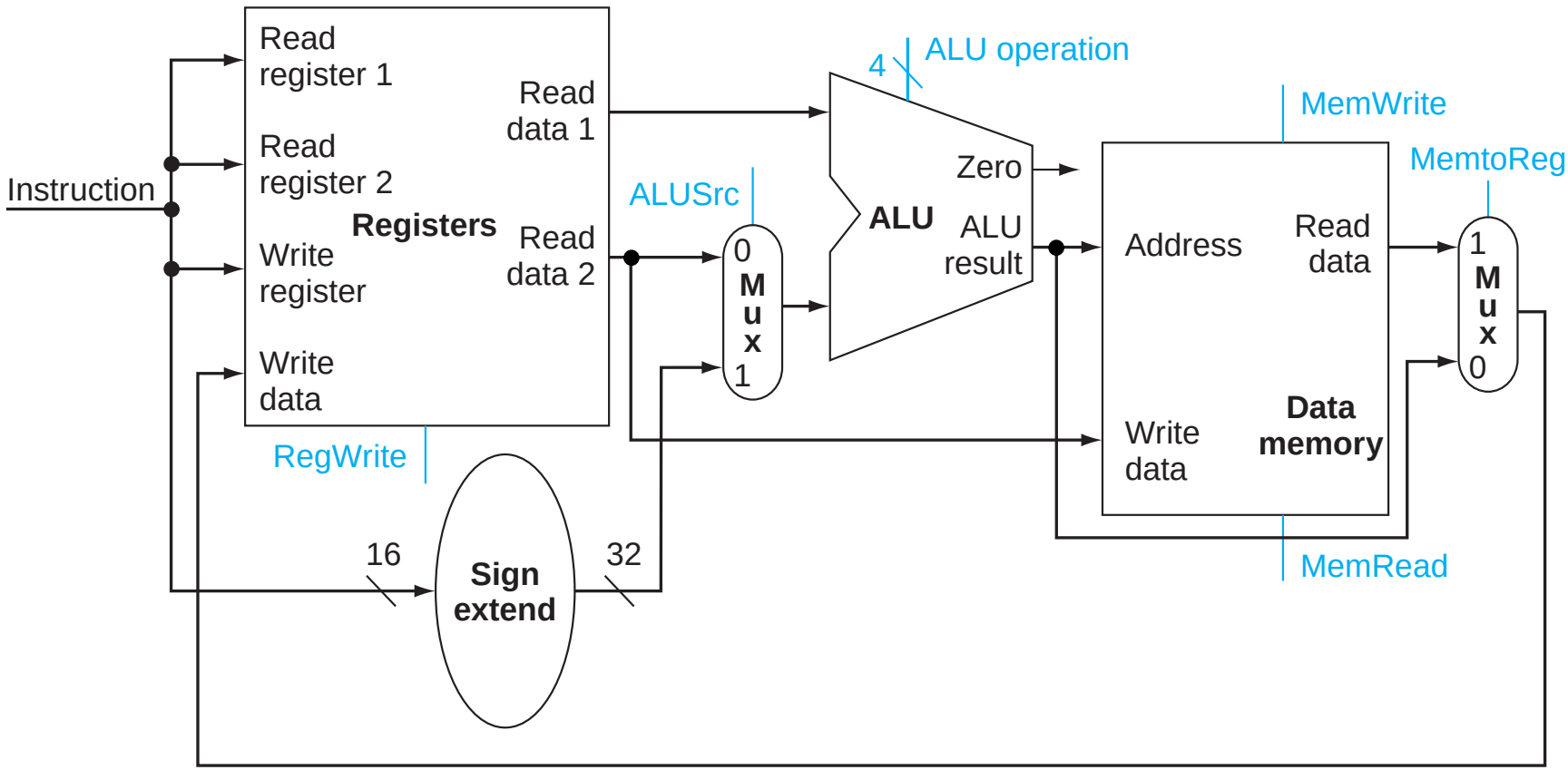


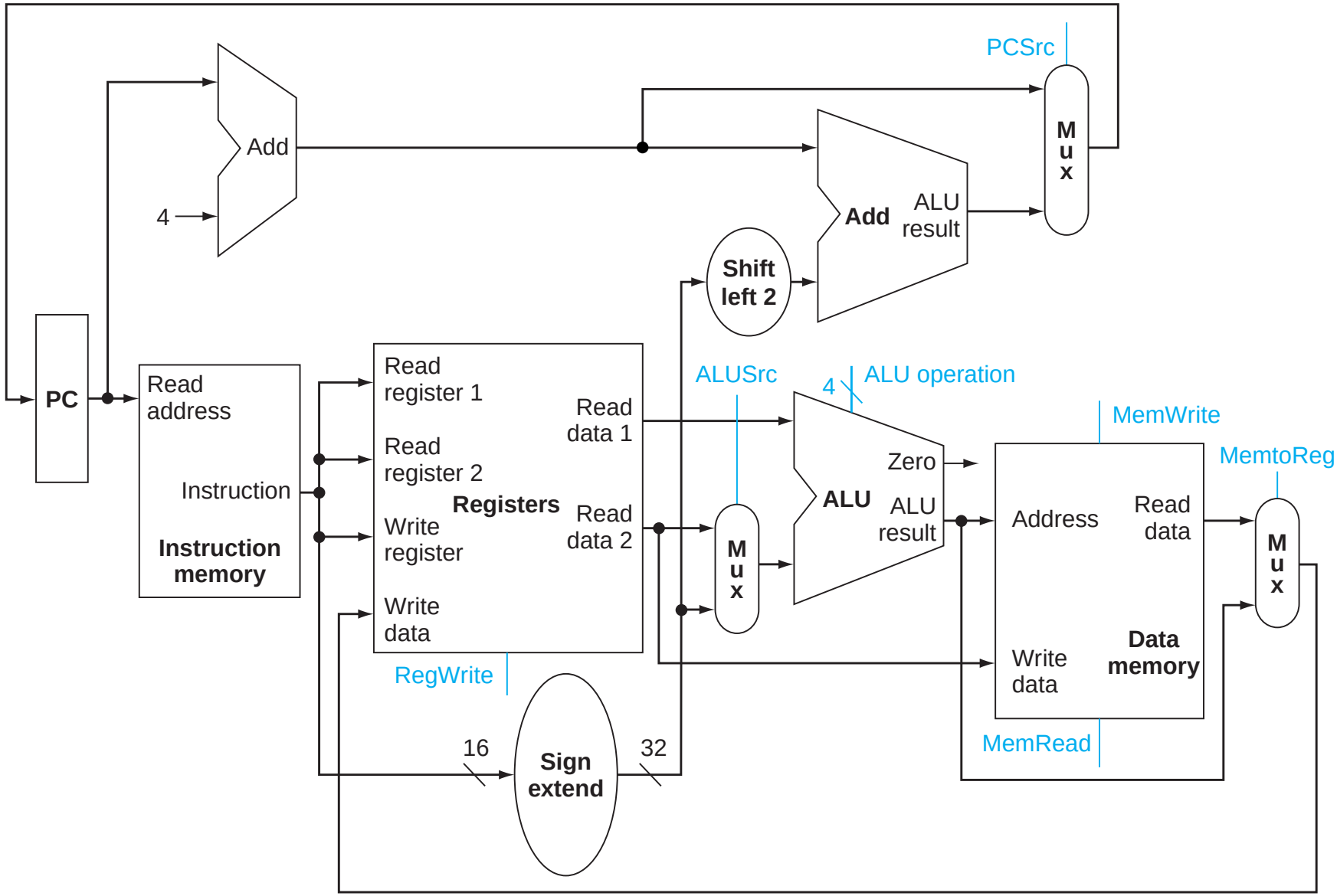
Uwaga o opóźnionych skokach

MIPS realizuje tzw. opóźnione skoki (*delayed branches*) – rozkaz znajdujący się w pamięci bezpośrednio za rozkazem skoku wykonuje się niezależnie od tego, czy skok następuje, czy nie.

Nie implementujemy tego rozwiązania. Podobnie zresztą jak SPIM (chociaż tam można je wymusić, uruchamiając program z odpowiednim parametrem).

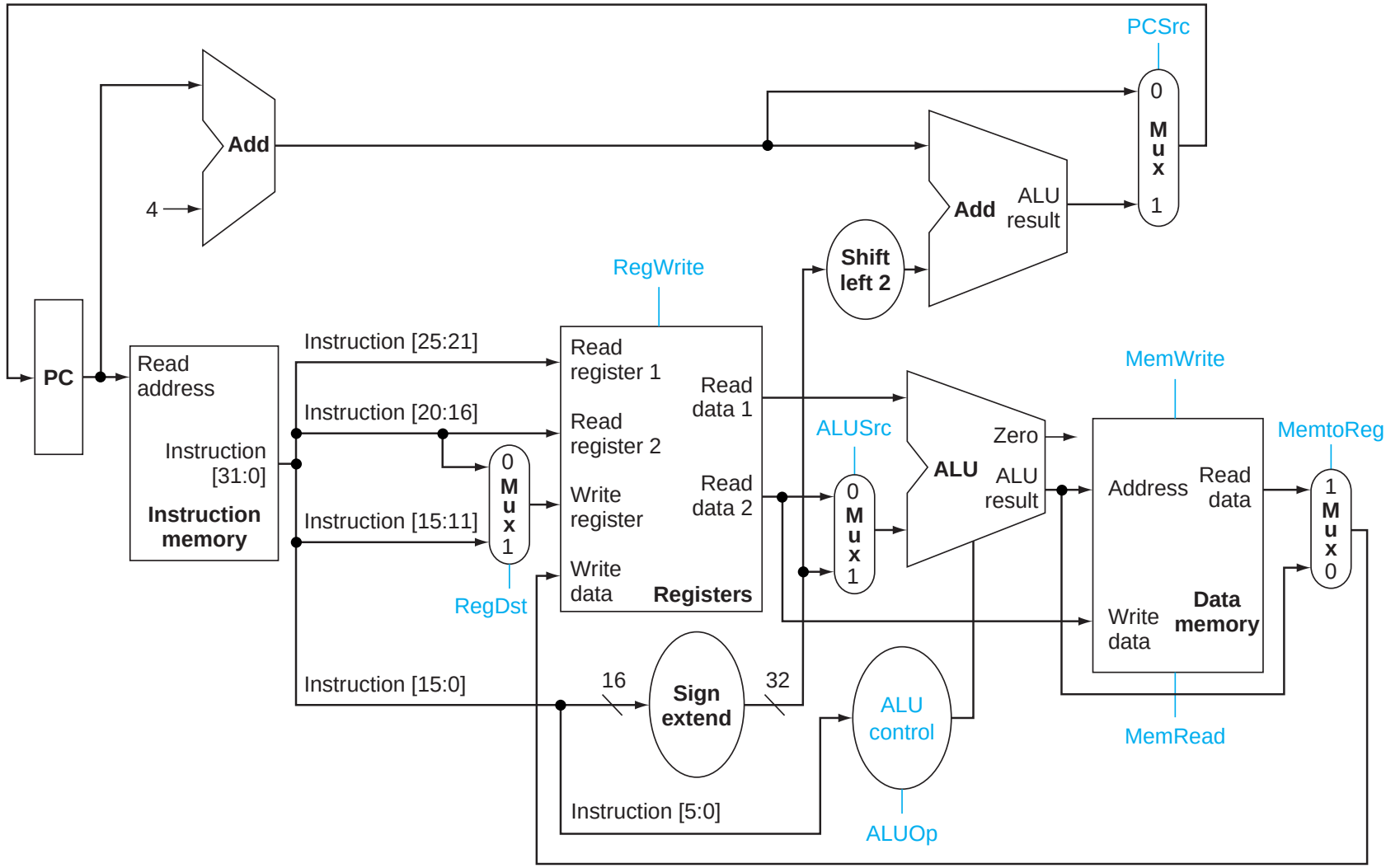
Układ dla rozkazów arytm.-log. oraz `lw, sw`




PCSrc
Add
4
M
u
x
Add
ALU
result
Shift
left 2
PC
Read
address
Instruction
Instruction
memory
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
RegWrite
ALUSrc
4
ALU operation
M
u
x
ALU
Zero
ALU
result
MemWrite
MemtoReg
Address
Read
data
Write
data
Data
memory
MemRead
M
u
x
16
Sign
extend
32

Formaty rozkazów – przypomnienie

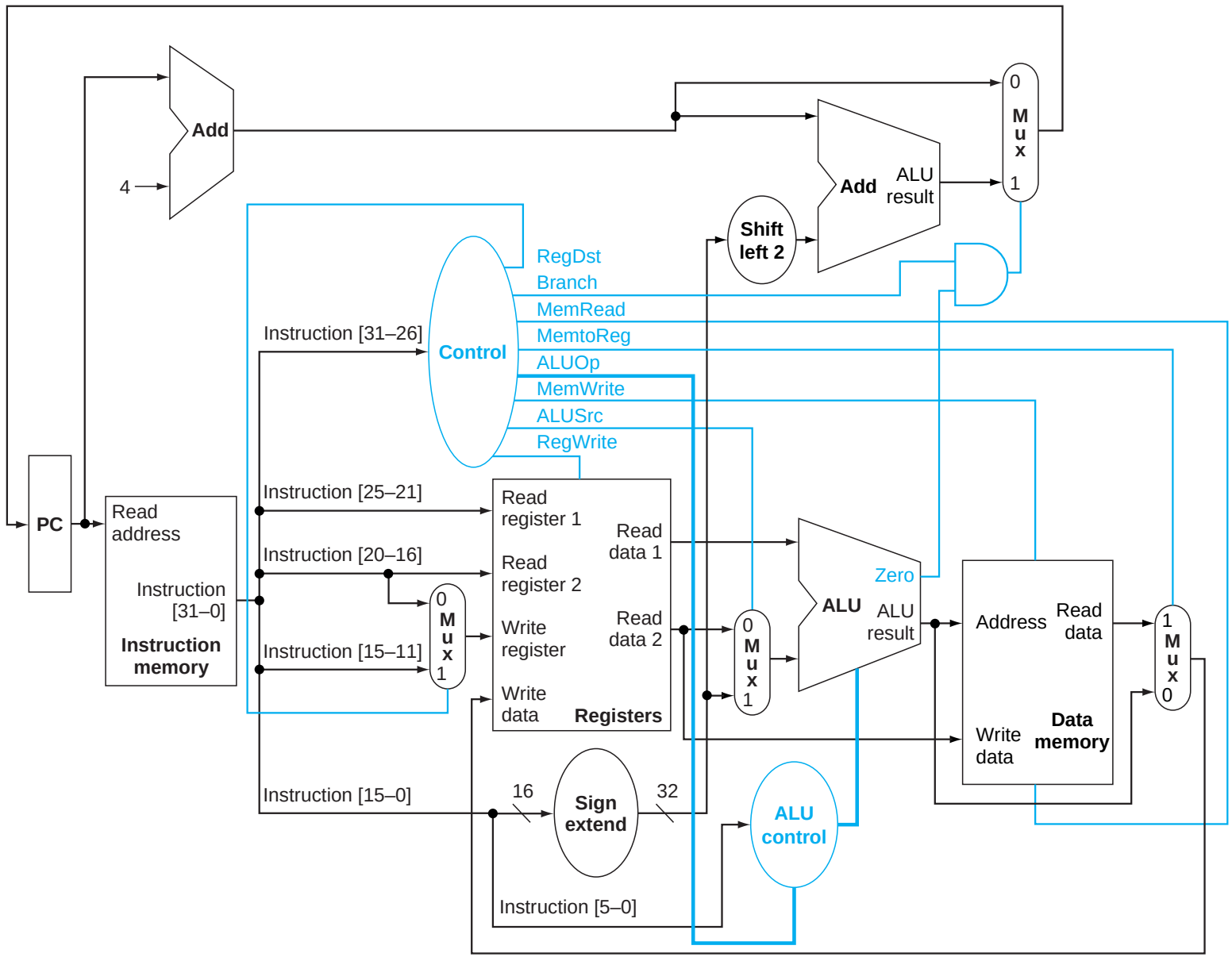
- Typ R:
  - 6 bitów – kod operacji (bity 31:26)
  - 5 bitów – numer rejestru źródłowego 1, rs (bity 25:21)
  - 5 bitów – numer rejestru źródłowego 2, rt (bity 20:16)
  - 5 bitów – numer rejestru wynikowego, rd (bity 15:11)
  - 5 bitów – shamt (my je ignorujemy)
  - 6 bitów – rodzaj operacji, funkcja (bity 5:0)
- Typ I, `lw, sw`:
  - 6 bitów – kod operacji (bity 31:26)
  - 5 bitów – numer rejestru bazowego adresu (bity 25:21)
  - 5 bitów – numer rejestru (źródłowego dla `sw`, wynikowego dla `lw` (bity 20:16)
  - 16 bitów – stała (bity 15:0)
- Typ I, `beq`: jak wyżej, oba numery rejestów to rejestry źródłowe do porównania.


PCSrc
Add
4
0
M
u
x
1
Add
ALU
result
Shift
left 2
RegWrite
PC
Read
address
Instruction
[31:0]
Instruction
memory
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]
Instruction [15:0]
Instruction [5:0]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
RegDst
ALUSrc
ALU
Zero
ALU
result
MemWrite
Address
Read
data
Write
data
Data
memory
MemRead
MemtoReg
16
Sign
extend
32
ALU
control
ALUOp

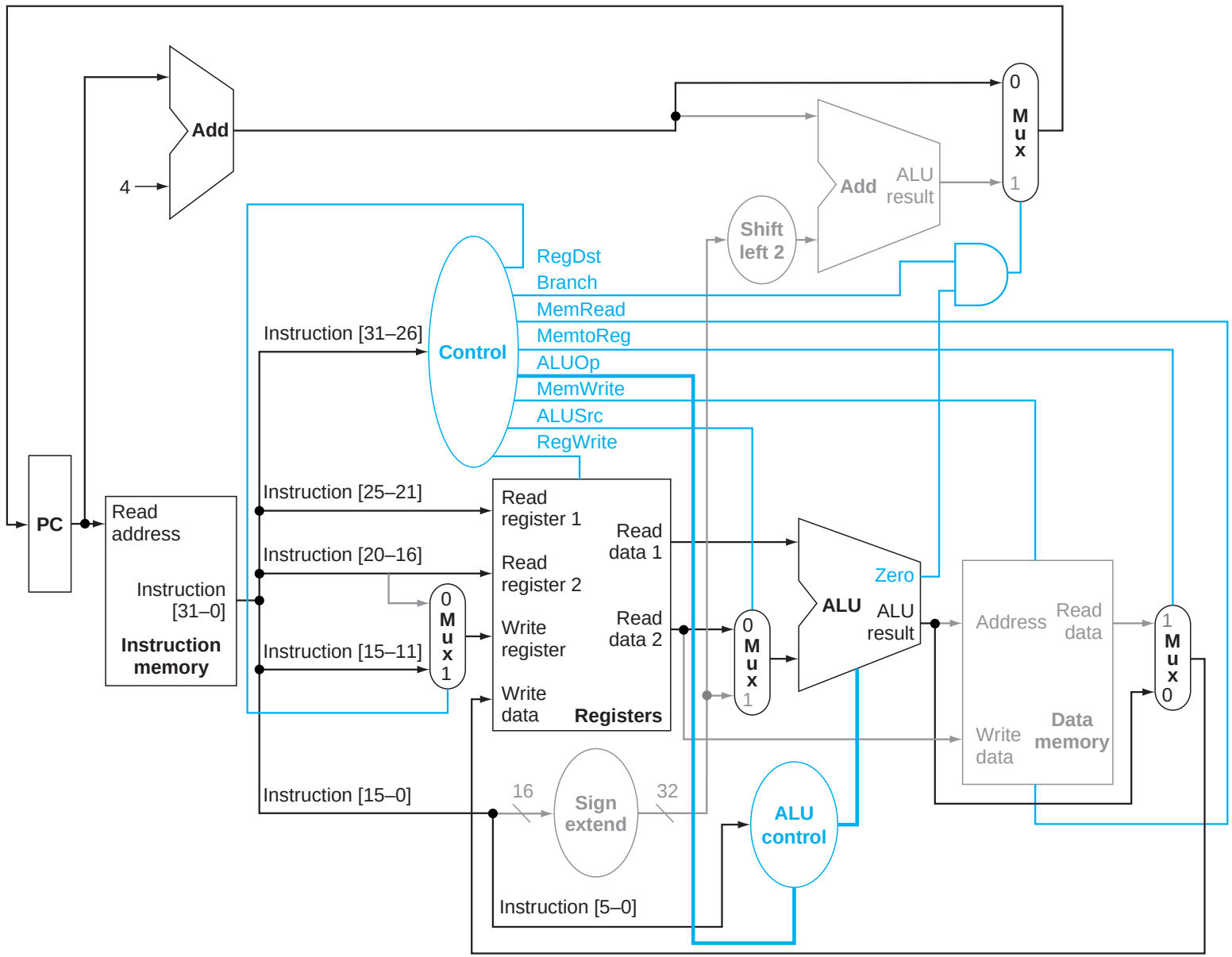
Sterowanie

- Na podstawie bitów 31:26 (kod operacji) generowane są odpowiednie sygnały sterujące (prosty układ kombinacyjny)
- Sygnał ALUOp jest dwubitowy: 00 – `lw` lub `sw` (dodawaj), 01 - `beq` (odejmuj), 10 - typu R
  - Rodzaj operacji dla ALU przy kodzie 10 dospecyfikowywany przez bity funkcyjne 5:0

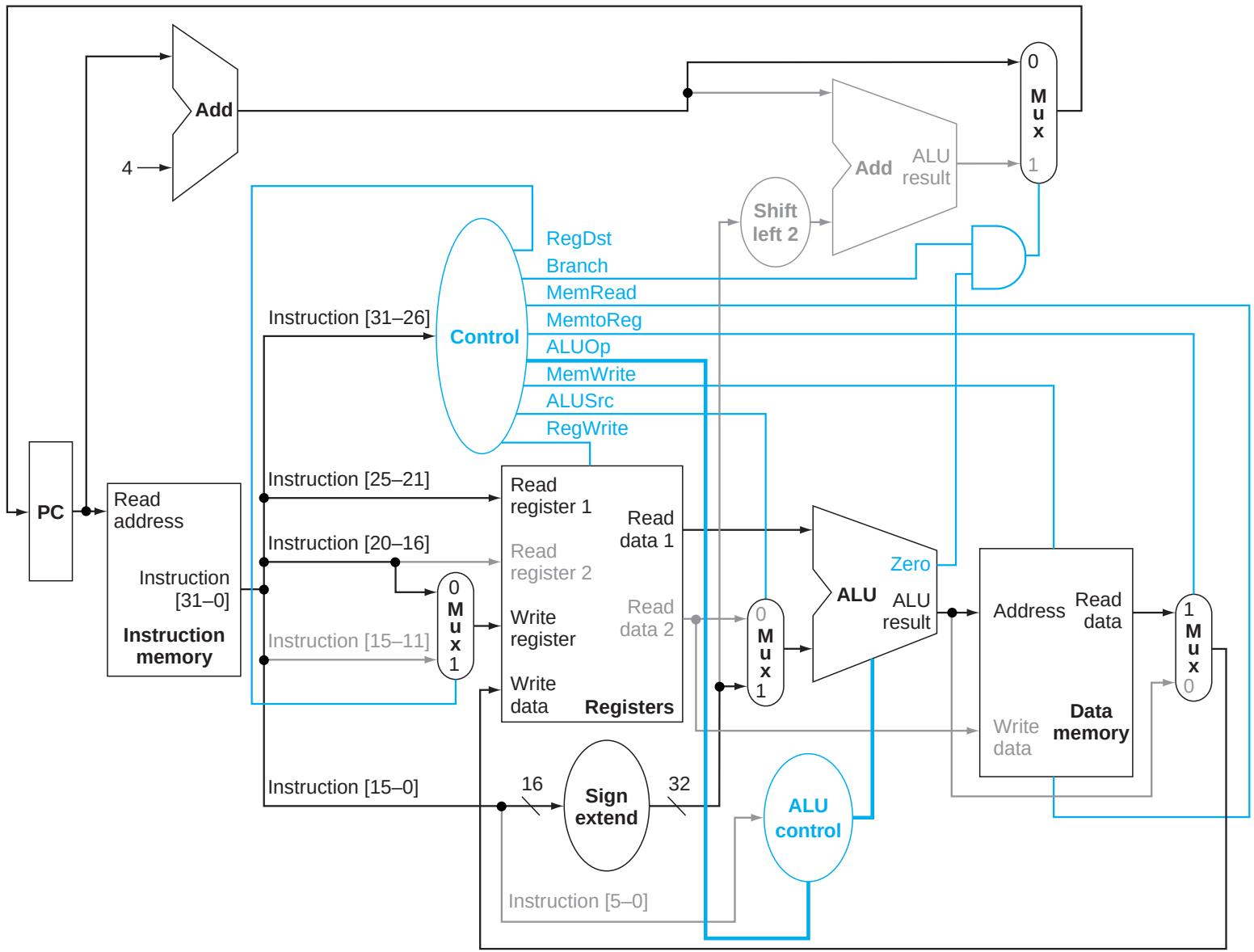
Pełny układ (ale jeszcze bez skoku `j`)



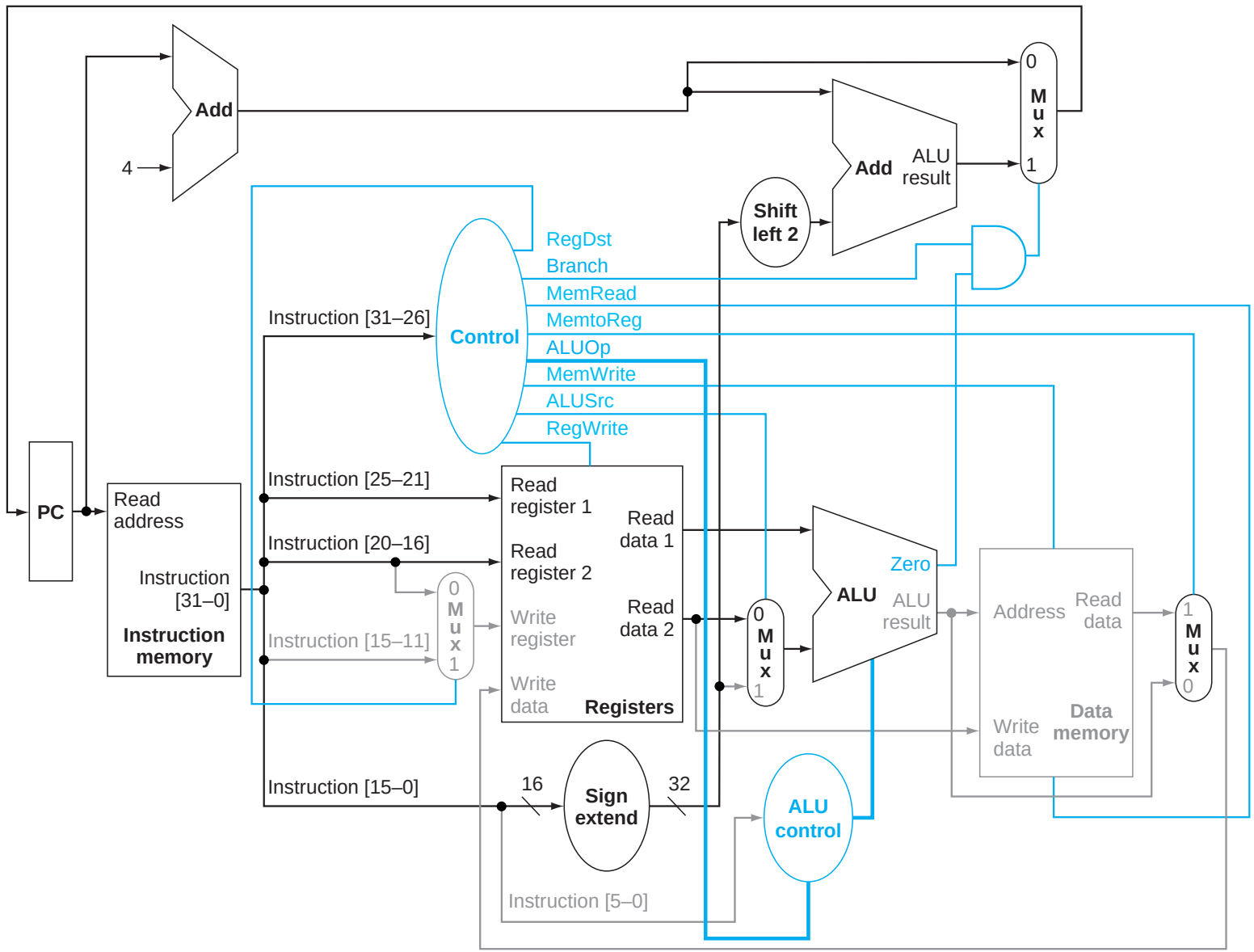
add $t0, $t1, $t2


Add
4
0
M
u
x
1
Add
ALU
result
Shift
left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read
address
Instruction
[31–0]
Instruction
memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
ALU
Zero
ALU
result
ALU
control
16
Sign
extend
32
Address
Read
data
Write
data
Data
memory

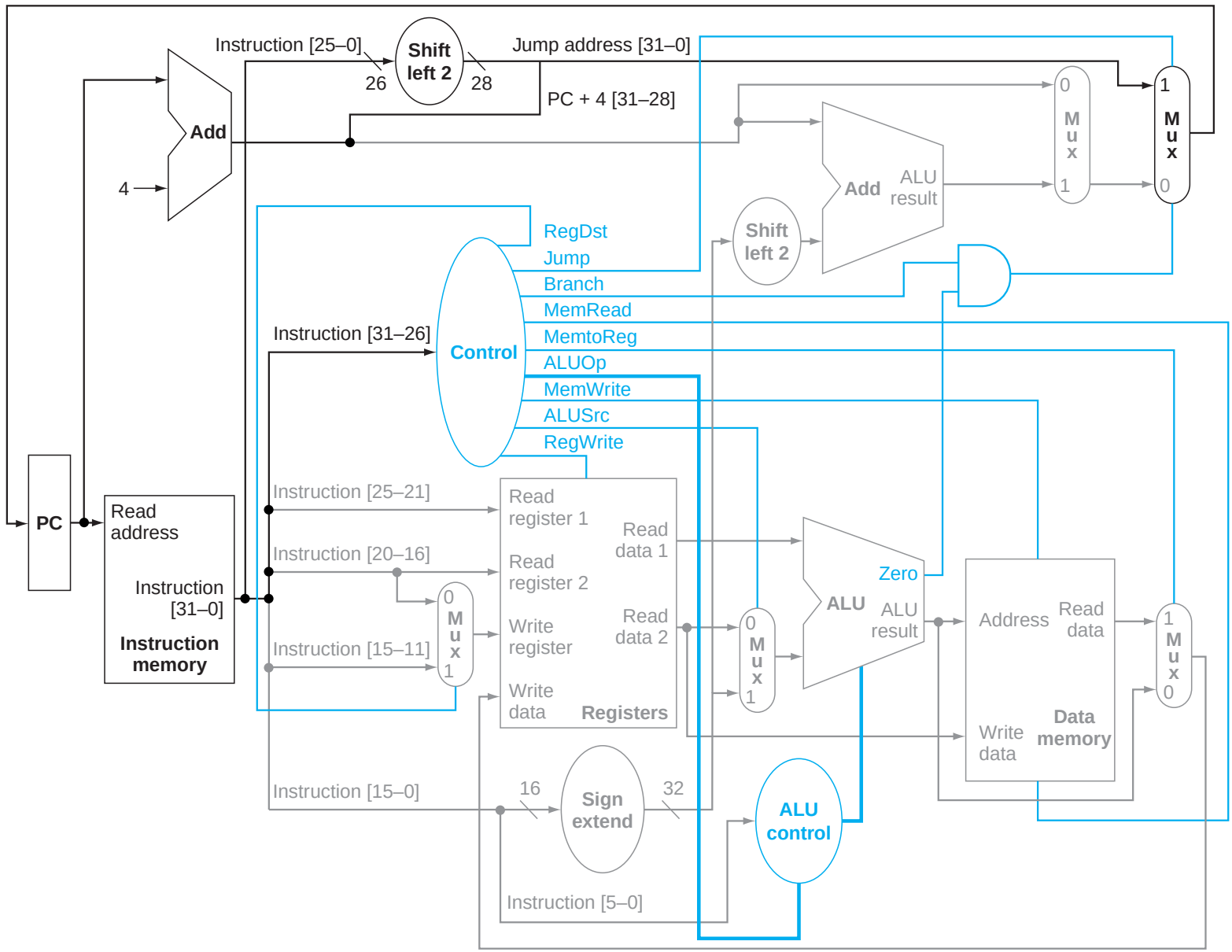
lw $t0, 8($t2)


Add
4
0
M
u
x
1
Add
ALU
result
Shift
left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read
address
Instruction
[31–0]
Instruction
memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
16
Sign
extend
32
ALU
control
Zero
ALU
ALU
result
Address
Read
data
Write
data
Data
memory

beq $t0, $t1, 8


Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Write data
Data memory

Dodajemy skok bezwarunkowy



Wady i zalety implementacji jednocyklowej

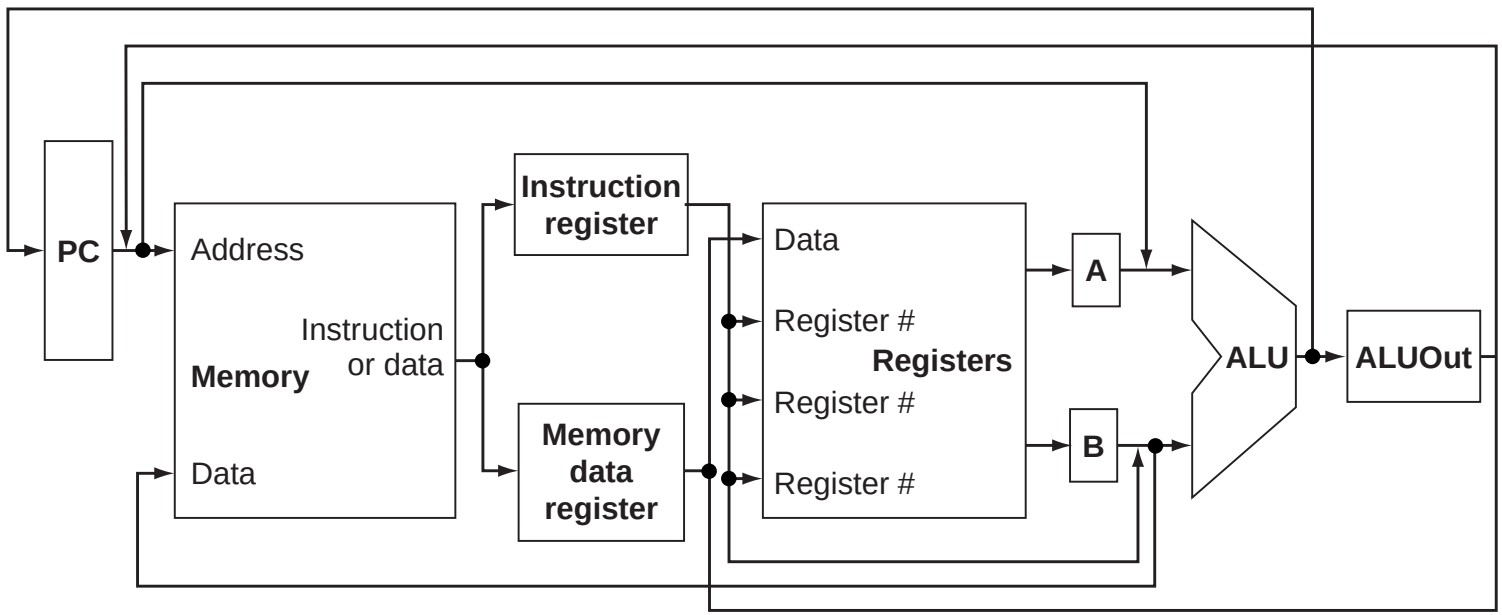
- Przedstawiona implementacja nie jest używana w praktyce
- Konieczność duplikacji sprzętu – dodatkowe sumatory
- Cykl zegarowy dopasowany do najwolniejszego rozkazu (operacje na pamięci)
- Zaleta: prostota
- Inne rozwiązania: zmienna długość cyklu – skomplikowane technicznie, implementacja wielocyklowa – o tym za chwilę, przetwarzanie potokowe – o tym za tydzień, ...

Wady i zalety implementacji jednocyklowej

- Przykładowe czasy działania naszych podukładów i wykonywania rozkazów:
  - IMem 200 ps, RegRead 100, ALU 200, DMem 200, RegWrite 100
  - Rozkaz typu R: $200 + 100 + 200 + 0 + 100 = 600$
  - `lw`: $200 + 100 + 200 + 200 + 100 = 800$
  - sw: $200 + 100 + 200 + 200 + 0 = 700$
  - Skok warunkowy: $200 + 100 + 200 + 0 + 0 = 500$
  - Skok bezwarunkowy: $200 + 100 + 0 + 0 + 0 = 300$
- Potrzebujemy zatem cyklu długości 800 ps.
- Rozważmy następujący zestaw rozkazów: 25% odczytów pamięci, 10% zapisów, 45% rozkazów typu R, 15% skoków warunkowych, 5% skoków bezwarunkowych
- Średni czas wykonania instrukcji:
  $0,25 \cdot 800 + 0,10 \cdot 700 + 0,45 \cdot 600 + 0,15 \cdot 500 + 0,05 \cdot 300 = 630$ ps.

Implementacja wielocyklowa

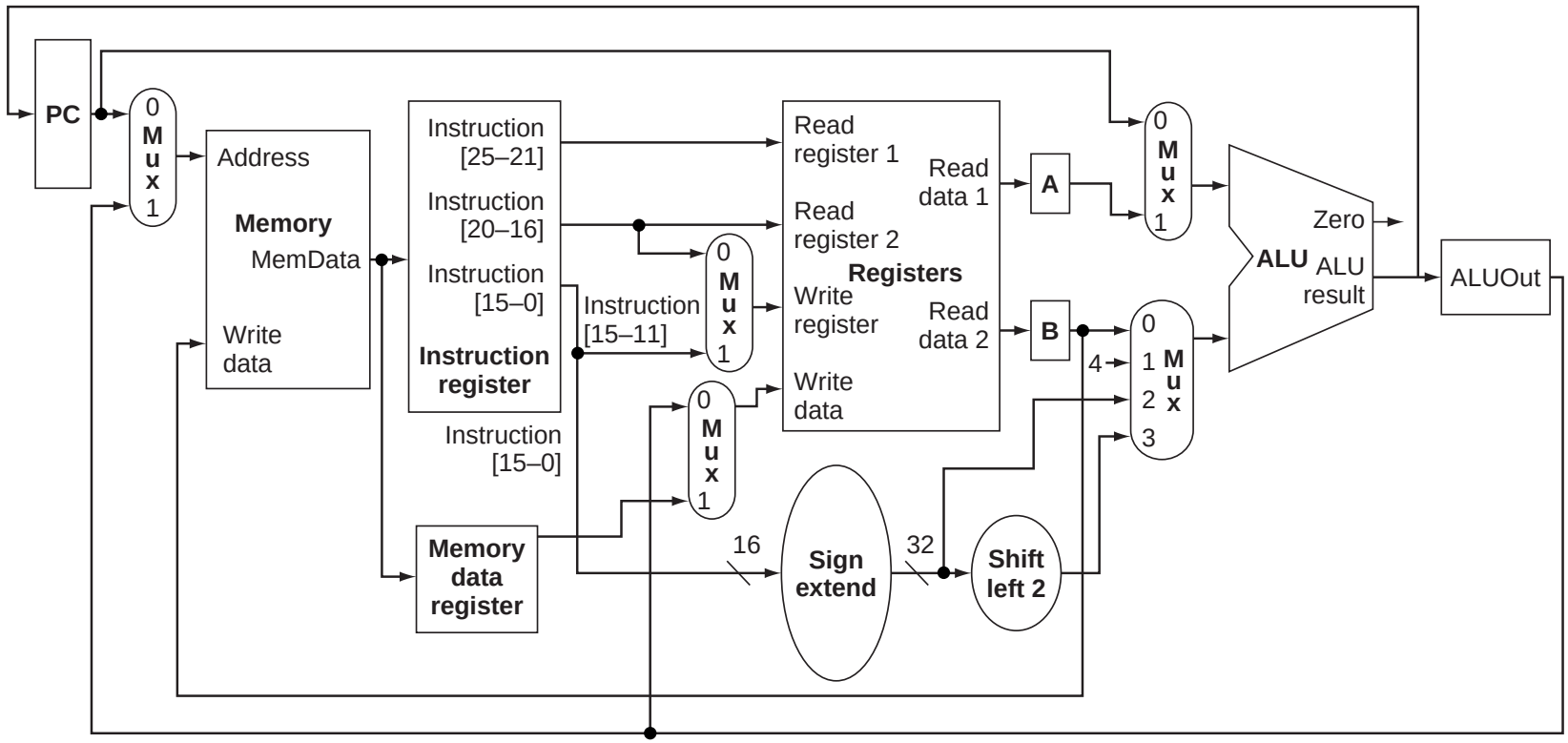
- Krótszy cykl zegarowy
- rozkazy dzielone na fazy (kroki), każda faza – jeden cykl.
- Rozkazy mogą mieć różne liczby faz.
- Jedna ALU (bez żadnych dodatkowych sumatorów).
- Wspólna pamięć programu i danych.
- Kilka dodatkowych rejestrów przechowujących dane potrzebne w kolejnych fazach



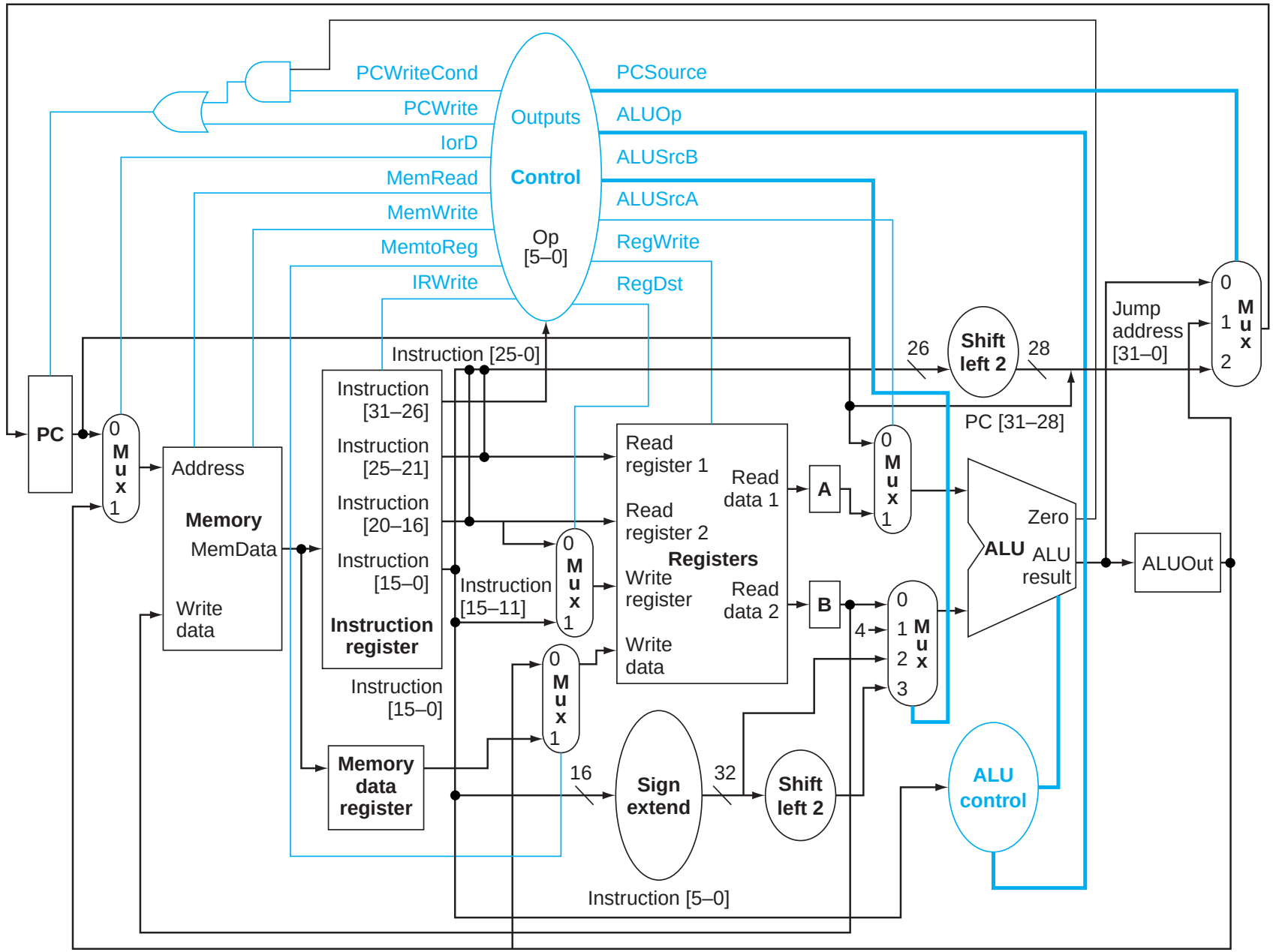
Implementacja wielocyklowa – cd.

- Każdy z rejestrów MDR, A, B, ALUOut przechowuje dane z fazy k dla fazy $k+1$
- Rejestr IR przechowuje rozkaz przez wszystkie fazy jego wykonania.
- Dlatego tylko IR potrzebuje sygnału zezwolenia na zapis.
- Niektóre układy będą używane więcej niż raz na rozkaz, dlatego musimy dodać kilka multiplekserów (i rozszerzyć wcześniejsze) – np.:
  - mamy teraz dwa źródła adresów dla pamięci
  - ALU wylicza adresy skoków

Implementacja wielocyklowa (bez skoków `j`)



Pełna implementacja wielocyklowa



F1 - Pobranie rozkazu

- `IR <= Memory[PC]`
- `PC <= PC + 4`

Powyższe operacje mogą wykonywać się równolegle. Sygnały sterujące:

- MemRead - 1
- IRWrite - 1
- IorD - 0
- ALUSrcA - 0 (PC)
- ALUSrcB - 01 (stała 4)
- ALUOp – 00 (dodawanie)
- PCSource – 0
- PCWrite – 1

F2 - Dekodowanie oraz odczytu rejestrów

- `A <= Reg[IR[25:21]]`
- `B <= Reg[IR[20:16]]`
- `ALUOut <= PC + (sign-extend (IR[15-0] « 2)`

Powyższe operacje wykonujemy przy każdej instrukcji - niezależnie od kodu operacji. Zauważmy, że nawet jeśli konkretna instrukcja tego nie wymaga, to wykonane operacje w niczym nie zaszkodzą. Sygnały sterujące:

- ALUSrcA - 0 (PC)
- ALUSrcB - 11 (przesunięta i rozszerzona stała z rozkazu)
- ALUOp - 00 (dodawanie)

F3 - Wykonanie, obliczenie adresu lub skok

Pierwszy cykl, który zależy od kodu operacji. ALU wykonuje jedną z czterech możliwych akcji (na danych przygotowanych w poprzednich cyklach):

- Obliczenie adresu: `ALUOut <= A + sign-extend (IR[15:0])`
  - ALUSrcA - 1, ALUSrcB - 10, ALUOp - 00 (dodawanie)
- Operacja arytmetyczno-logiczna (typu R): `ALUOut <= A op B`
  - ALUSrcA - 1, ALUSrcB - 00, ALUOp - 10 (decyduje pole funk.)
- Skok warunkowy: `if (A==B) PC <=ALUOut`
  - ALUSrcA - 1, ALUSrcB - 00, ALUOp - 01 (odejmowanie), PCWriteCond - 1, PCSource - 01
- Skok: `PC <= konkatenacja(PC[31:28], IR[25:0], 00)`
  - PCWrite - 1, PCSource - 10

F4 - Dostęp do pamięci lub dokończenie R-rozkazu

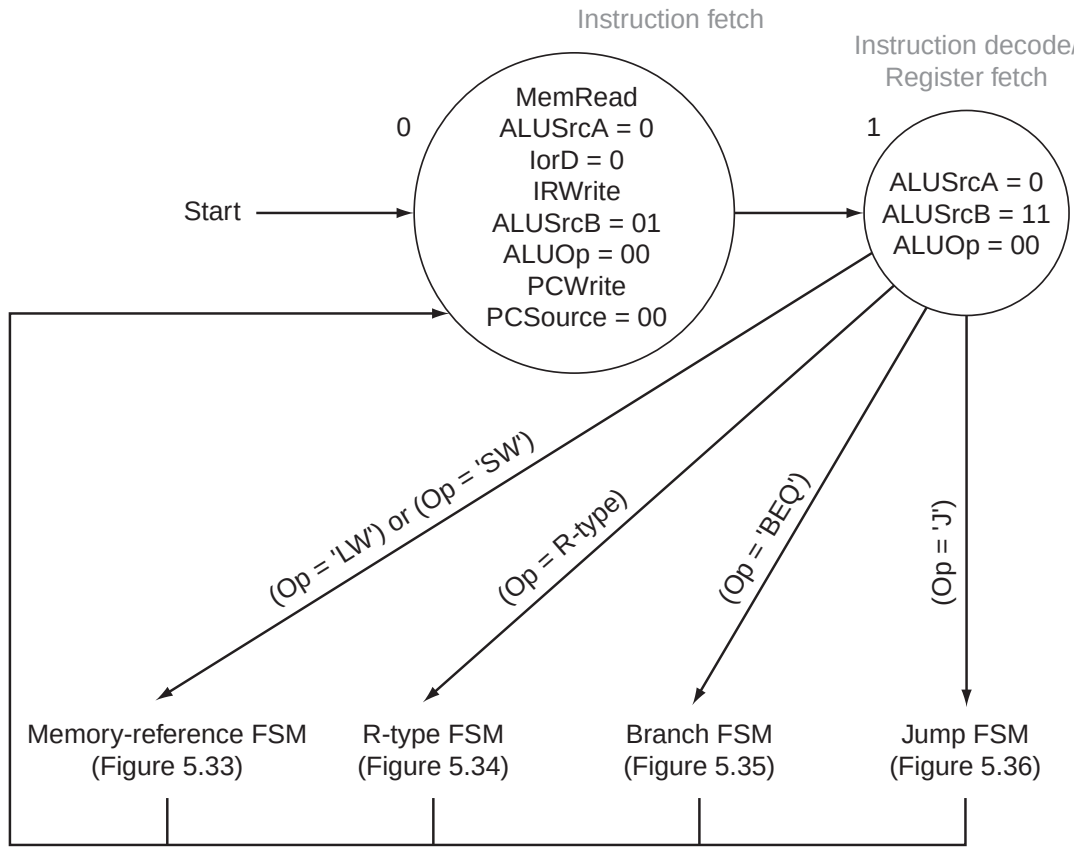
- Dostęp do pamięci: `MDR <= Memory [ALUOut]` lub `Memory [ALUOut] <= B`
  - MemRead lub MemWrite - 1, IorD - 1
- Dokończenie rozkazu arytmetyczno-logicznego: `Reg[IR[15-11]] <= ALUout`
  - RegDst - 1, RegWrite - 1, MemtoReg - 0,

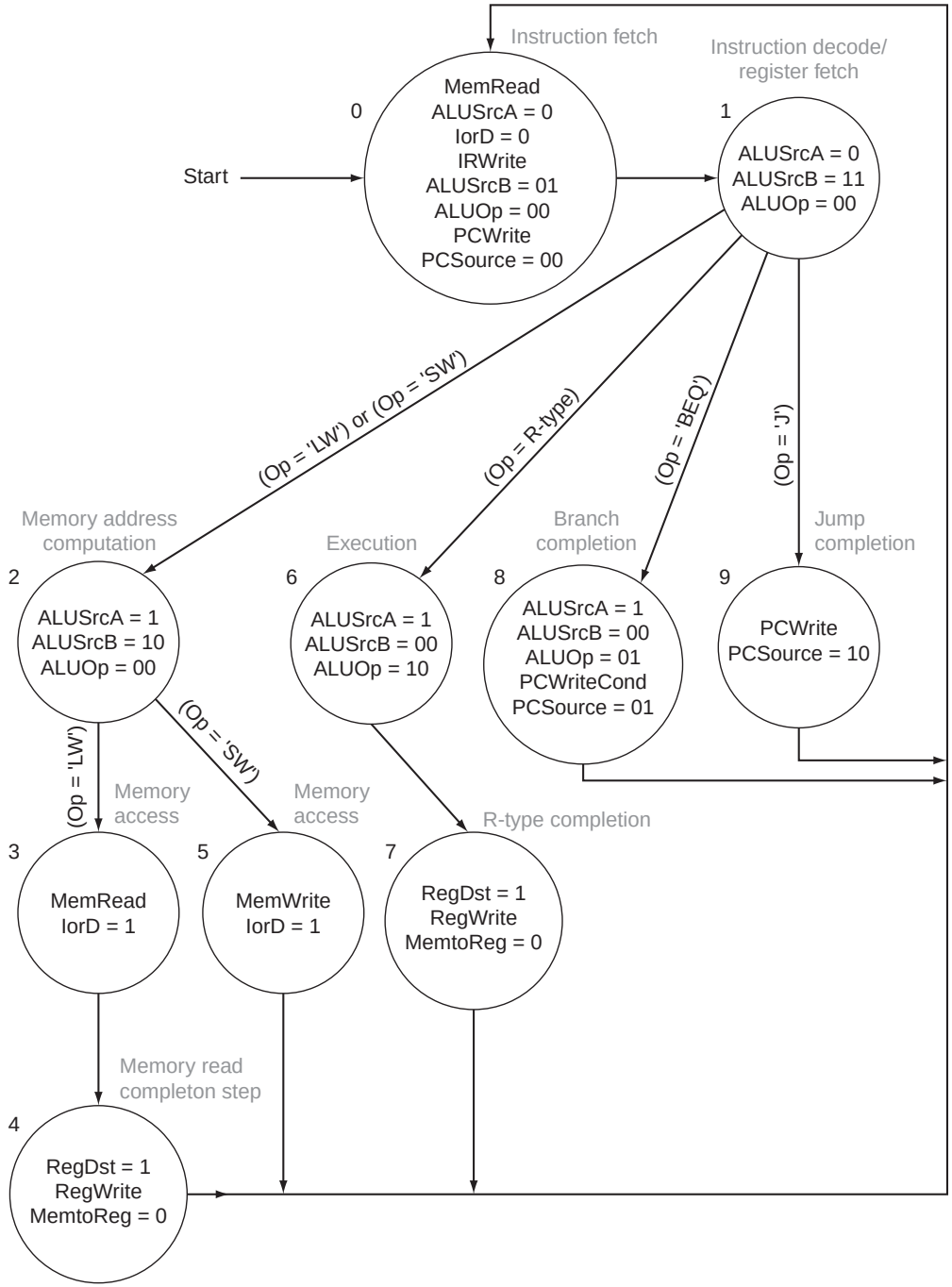
F5 - Dokończenie odczytu pamięci

- `Reg[IR[20-16]] <= MDR`
  - RegDst - 0, RegWrite - 1, MemtoReg - 1,

Jednosta sterująca

Tym razem jednostaka sterująca będzie układem sekwencyjnym, dokładniej automatem Moore'a. Wejście – kod operacji, stany to fazy, wyjścia – sygnały sterujące.




Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0