Computer Organization Term Project

NIT HAMIRPUR

TOPIC:- Design and Encode 8 bit Processor

Problem:- Data Structures and Algorithms – HEAP SORT

Group – 9

Students :-

185502 Mohit Kumar Verma

185537 Shaurya Pandey

185544 Karan Bhardwaj

Submitted To :-

Kamlesh Dutta Mam

(CSE Department)

DATA STRUCTUTRES AND ALGORITHMS :-

HEAP SORT

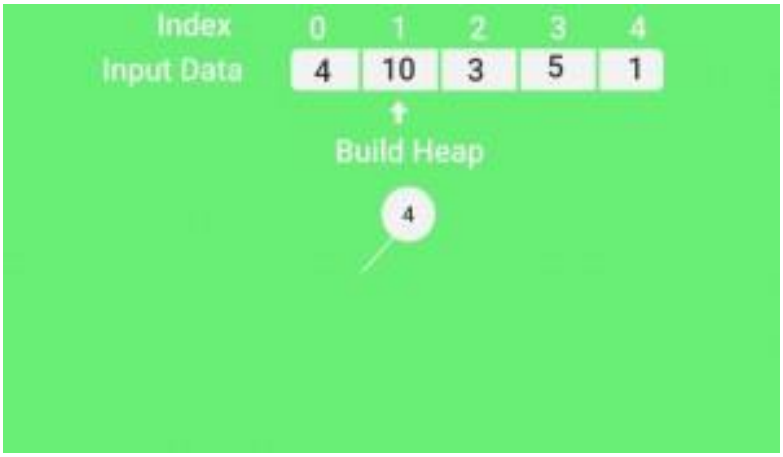
Heap Sort is a Sorting Algorithm in which a Max Heap is created for sorting and this algorithm has least time complexity i.e. O(N.log(n)). It is the best sorting algorithm because it has no worst case or best case scenario.

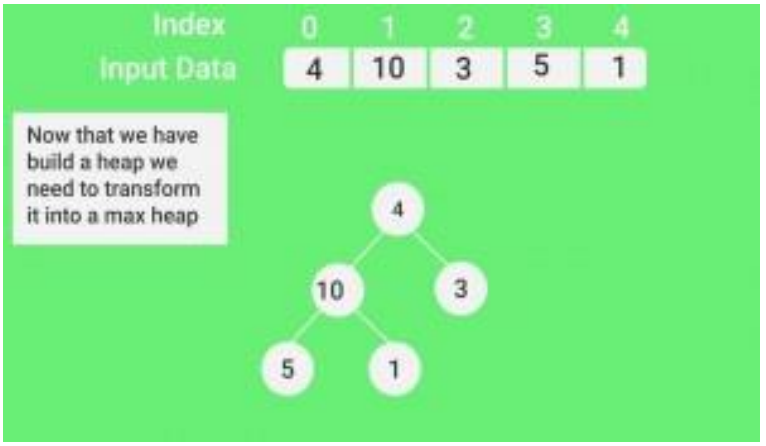
In this algorithm, there are 3 steps.

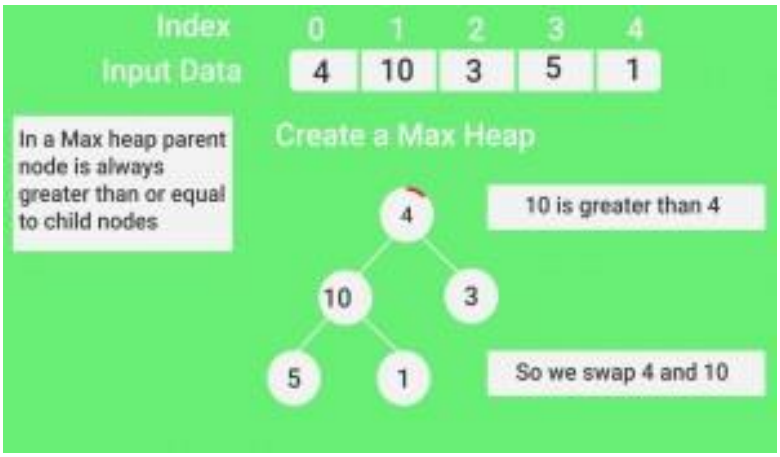
Step-1 : Create a max Heap from given set of elements.

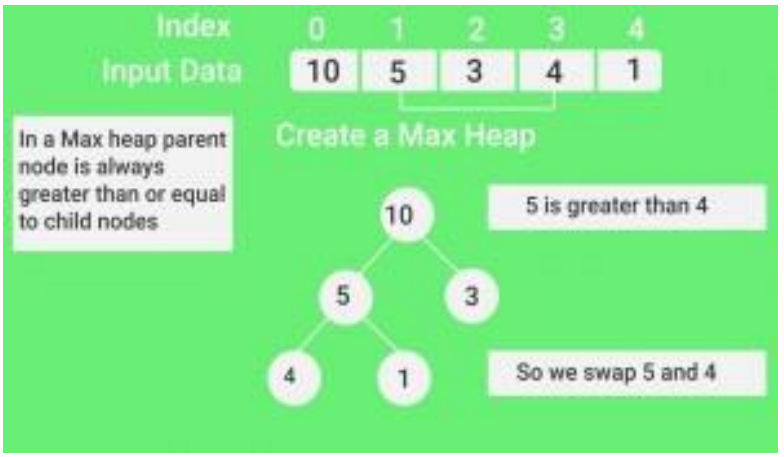
Step-2 : Swap the first and last element and remove the last element from the heap.

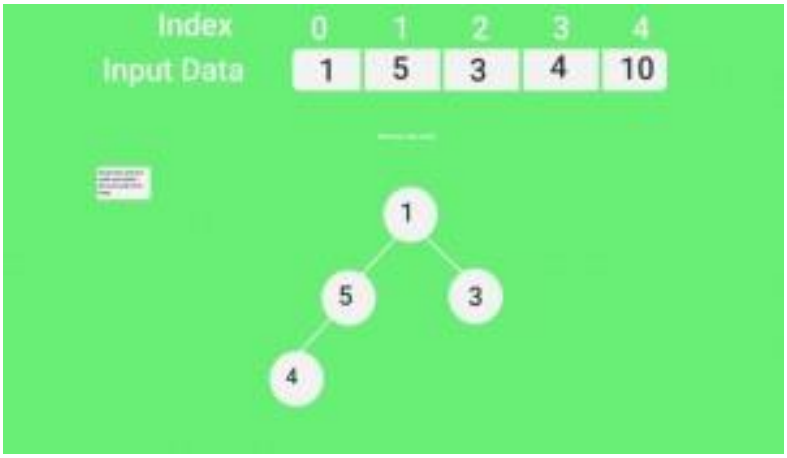
Step-3 : Repeat step 1 and 2.

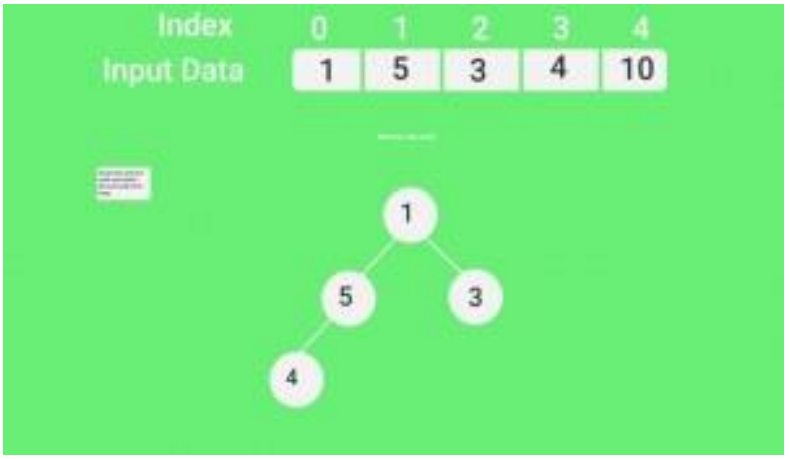












Steps in Heap Sort

C++ Code for Heap Sort (Iterative) :

```cpp
#include <bits/stdc++.h>
using namespace std;

void buildMaxHeap(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        // if child is bigger than parent
        if (arr[i] > arr[(i - 1) / 2])
        {
            int j = i;

            // swap child and parent until parent is smaller
            while (arr[j] > arr[(j - 1) / 2])
            {
                swap(arr[j], arr[(j - 1) / 2]);
                j = (j - 1) / 2;
            }
        }
    }
}

void heapSort(int arr[], int n)
{
    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--)
    {
        // swap value of first indexed with last indexed
        swap(arr[0], arr[i]);

        // maintaining heap after each swapping
        int j = 0, index;

        do
        {
            index = (2 * j + 1);

            // if l_child is less than r_child
            if (arr[index] < arr[index + 1] &&
                index < (i - 1))
                index++;

            // if parent is smaller than child
            if (arr[j] < arr[index] && index < i)
                swap(arr[j], arr[index]);
```

```
            j = index;

        } while (index < i);
    }
}

int main()
{
    int arr[] = {10, 20, 15, 17, 9, 21};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\n");

    heapSort(arr, n);

    // print array after sorting
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

C++ Code For Heap Sort (Recursive) :

```
#include <iostream>
using namespace std;

// creating a max heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // biggest as root
    int l = 2*i + 1; // left
    int r = 2*i + 2; // right

    // if left is grater than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // if right is greater than root
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

```
    // if biggest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // calling again for sub-tree
        heapify(arr, n, largest);
    }
}

// doing heap-sort
void heapSort(int arr[], int n)
{
    // rearrange array
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // removing one by one from heap
    for (int i=n-1; i>0; i--)
    {
        // swapping current element to last element
        swap(arr[0], arr[i]);

        // calling same function on modified max heap
        heapify(arr, i, 0);
    }
}

// printing the array
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// main program for calling everything
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

creations.c File:-

This file contains code for the creation of the registers, memory, pointer, instructions. A structure RAM is created in which everything is created and later on everything is accessed from them directly using pointers.

```c
#define FUNC_EXIT   0
#define FUNC_SET    1
#define FUNC_PRINT  2
#define FUNC_JMP    3
#define FUNC_MOV    4
#define FUNC_DEC    5
#define FUNC_INC    6
#define FUNC_SWAP   7
#define FUNC_HEAP   8


#define P(...) fprintf(stdout,__VA_ARGS__);fprintf(stdout,"\n")
#define S(...) fscanf(stdin,__VA_ARGS__);
#define L(...) fprintf(stderr,"[LOG] ");fprintf(stderr,__VA_ARGS__);fprintf(stderr,"\n")

typedef unsigned char u8 ;

struct RAM  {
    u8 storage[0x100];
    u8 ins[0x200];
    int ip;
    u8 regs[12];
};

typedef struct RAM RAM;
```

function.c File:-

This file contains all the functions that I have used to be executed in my Processor.

This contains:

1. MOV
2. EXIT
3. SET
4. PRINT
5. JMP
6. DEC
7. INC
8. SWAP
9. HEAP

```c
void f_mov(RAM *r)
{
    u8 src, val;
    src = r->ins[++r->ip];
    val = r->ins[++r->ip];
    L("mov command REGS[%d]:%d", src, val);
    r->regs[src] = val;
    return;
}

void f_exit(RAM *r)
{
    L("exit command");
    exit(EXIT_SUCCESS);
}

void f_set(RAM *r)
{
    u8 src, val;
    src = r->ins[++r->ip];
    val = r->ins[++r->ip];
    L("set command DATA[%d]:%d", src, val);
    r->storage[src] = val;
    return;
}

void f_print(RAM *r)
{
    int i = 0;
    L("print command first %d", r->regs[5]);
    for (; i <= r->regs[7]; i++)
        P("storage[%d]:%d regs[%d]:%d\n", i, r->storage[i], i, r->regs[i]);
    return;
}

void f_jmp(RAM *r)
{
    if (r->regs[9] == 1)
    {
        u8 des = r->ins[++r->ip];
        r->ip -= des;
    }
    return;
}

void f_dec(RAM *r)
{
    r->regs[8] -= 1;
}
```

```
void f_inc(RAM *r)
{
    r->regs[8] += 1;
}

void f_swap(RAM *r)
{
    r->regs[7] = r->regs[0];
    r->regs[0] = r->regs[r->regs[8]];
    r->regs[r->regs[8]] = r->regs[7];

    if (r->regs[8] < 1) //condition created for not jumping called with swap
    {
        r->regs[9] = 0;
    }
    ++r->ip;
}

void f_heap(RAM *r)
{
    int arr[6];
    for (int i = 0; i < 6; i++)
    {
        arr[i] = r->regs[i];
    }

    for (int i = 1; i < 6; i++)
    {
        if (arr[i] > arr[(i - 1) / 2])
        {
            int j = i;
            while (arr[j] > arr[(j - 1) / 2])
            {
                int temp=0;
                temp=arr[j];
                arr[j]=arr[(j - 1) / 2];
                arr[(j - 1) / 2]=arr[j];
                j = (j - 1) / 2;
            }
        }
    }

    for (int i = 0; i < 6; i++)
    {
        r->regs[i] = arr[i];
    }

    ++r->ip;
}
```

```
static void (*F[0xff])(RAM *r) = {
    [FUNC_SET] = f_set,
    [FUNC_EXIT] = f_exit,
    [FUNC_PRINT] = f_print,
    [FUNC_JMP] = f_jmp,
    [FUNC_MOV] = f_mov,
    [FUNC_DEC] = f_dec,
    [FUNC_INC] = f_inc,
    [FUNC_SWAP] = f_swap,
    [FUNC_HEAP] = f_heap,
};
```

source.asm File:-

This file contains the source code to be run in the processor created.

```
SET 0 12
SET 1 11
SET 2 13
SET 3 5
SET 4 6
SET 5 7
SET 6 6          # total size of array (static)
SET 7 0          # will be used for swapping
SET 8 6          # will be used in the loop (same as REGISTER
6)
SET 9 1          # will be used in jump
SET 10 0         # used in heapify for largest
SET 11 0         # used in heapify for smallest

HEAP             # heapify all elements (creating max heap)
```

```
SWAP            # swaps value of register 0 with the register
 pointed by register 8
DEC             # decreases value of register 8 by 1
JMP 13          # jumping to heap command
PRINT           # print register 0 to 5

EXIT            # stoping the machine
```

write.c File:-

This file contains the code for creating a temporary file and storing the contents of source.asm in that file so that our main source does not get changed.

```c
#include <stdio.h>
#include <stdint.h>

#include "._a.c"


int main ( int argc , char *argv[] , char** env ) {
    FILE *f = fopen(argv[1],"wb");
    fwrite(arr,sizeof(arr),1,f);
    fclose(f);
    return 0;
}
```

main.c File:-

This file contains the main code that will be executed.

Taking the instructions from source.asm file and opening the temporary file where commands are stored and then running the commands are some of the work done by this code.

```c
#include <stdlib.h>
#include <stdio.h>

#include "creations.c"
#include "functions.c"

RAM r;


int main( int argc , char* argv[] , char** env ) {
    FILE* f = fopen(argv[1],"rb");
    fseek(f,0,SEEK_END);
    long _s = ftell(f);
    fseek(f,0,SEEK_SET);
    if (_s > 0x100) return 2;
    fread(r.ins,1,_s,f);
    r.ip = 0;
    while(1) {
        F[r.ins[r.ip]](&r);
        r.ip++;
    }
    fclose(f);
    return EXIT_SUCCESS;
}
```

run.sh Script:-

This script contains commands to run all the files written in C.

To run this script, run the following command -

source ./run.sh source.asm

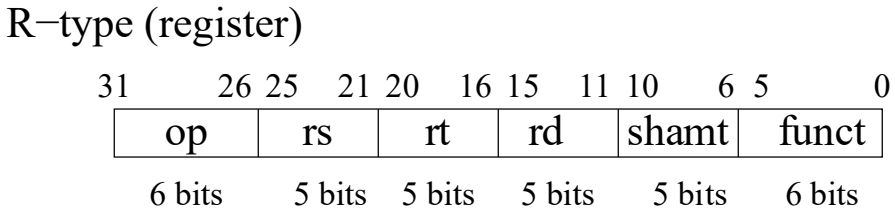
```
echo -ne 'uint8_t arr[] = { ' >._a.c
cat $1 | sed -e 's/^#.*//g' \      # "cat" command is used to create a new file.
    -e 's/MOV/4/g' \               # in this case, _a.c file is created.
    -e 's/SET/1/g' \
    -e 's/EXIT/0/g' \
    -e 's/PRINT/2/g' \
    -e 's/JMP/3/g' \
    -e 's/DEC/5/g' \
    -e 's/INC/6/g' \
    -e 's/SWAP/7/g' \
    -e 's/HEAP/8/g' \
    -e 's/\s/,/g' \
    -e '/^$/d' \
    -e 's/$/,/' >>._a.c
echo -ne '0 } ;' >>._a.c

gcc --std=c99 -o _b write.c
./_b asm.o
gcc --std=c99 -o bin main.c
./bin asm.o
rm _b ._a.c asm.o bin
```

Datapath and Control

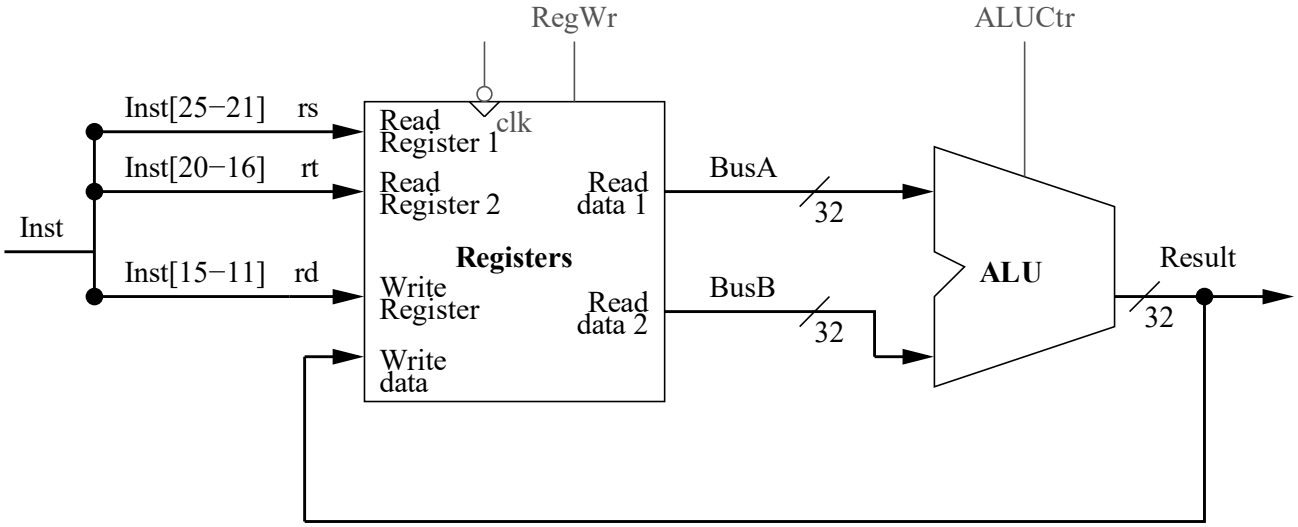
Datapath for R-type instructions

- `R[rd]` ← `R[rs] op R[rt]` Example: `add rd, rs, rt`

Recall that this instruction type has the following format:



The datapath contains the 32 bit register file and and ALU capable of performing all the required arithmetic and logic functions.



Note that the register is read from and written to at the "same time." This implies that the register's memory elements must be edge triggered, or are read and written on different clock phases, to allow the arithmetic operation to complete before the data is written in the register.

This datapath contains everything required to implement the required instructions `add`, `sub`, `and`, `or`, `xor`. All that is required is that the appropriate values be provided for the `ALUCtr` input for the required operation.

The register operands in the instruction field determine the registers which are read from and written to, and the `funct` field of the instruction determine which particular ALU operation is executed.

Recalling the control inputs for the ALU seen earlier, the values for the control input are:

ALU control lines	Function
000	`and`
001	`or`
010	`add`
110	`subtract`
111	`xor`

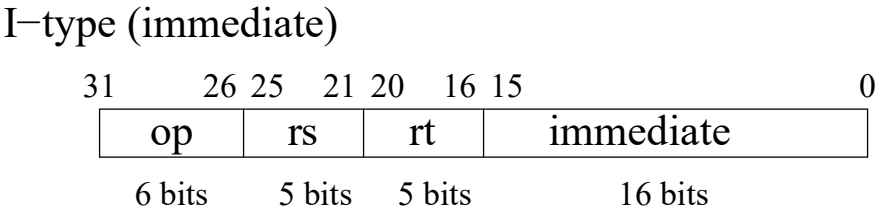
A control unit for the processor will be designed later.

It will set all the required control signals for each instruction, depending both on the particular instruction being executed (the `op code`) and, for r-type instructions, the `funct` field.

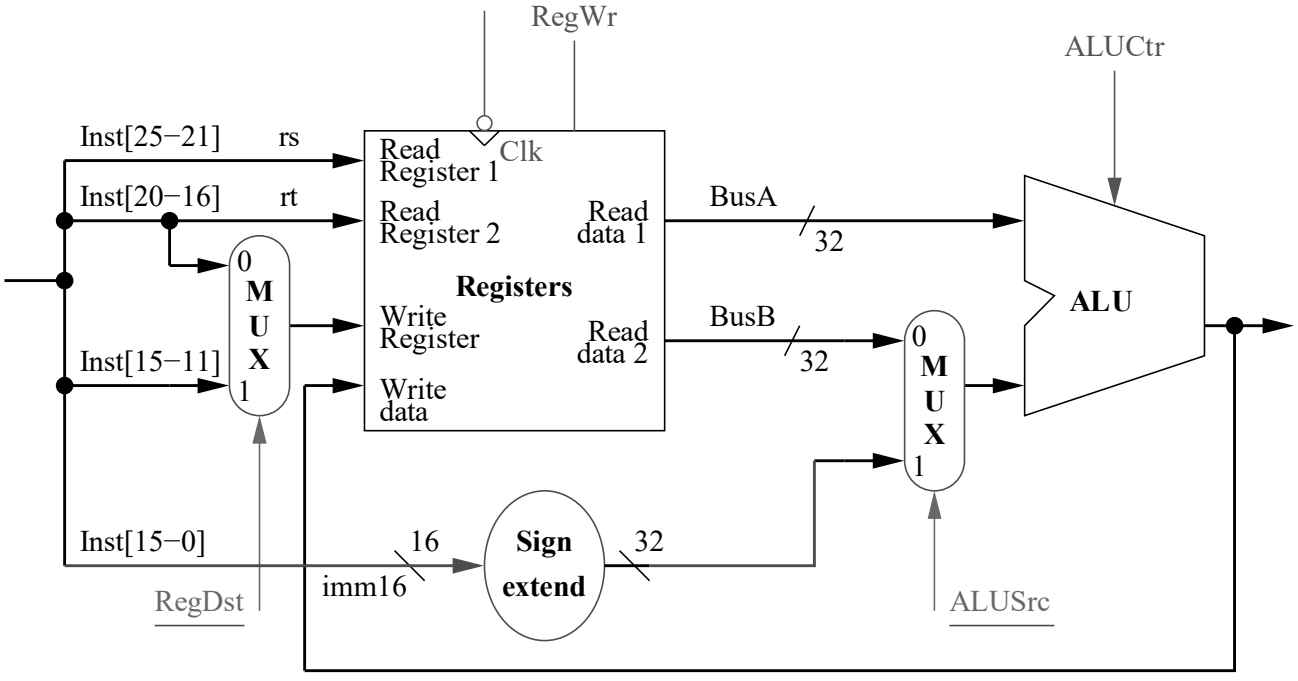
Datapath for Immediate arithmetic and logical instructions

- `R[rt] ← R[rs] op imm16` Example: `addi rt, rs, imm16`

Recall that this instruction type has the following format:



The main difference between this and an r-type instruction is that here one operand is taken from the instruction, and sign extended (for signed data) or zero extended (for logical and unsigned operations.)



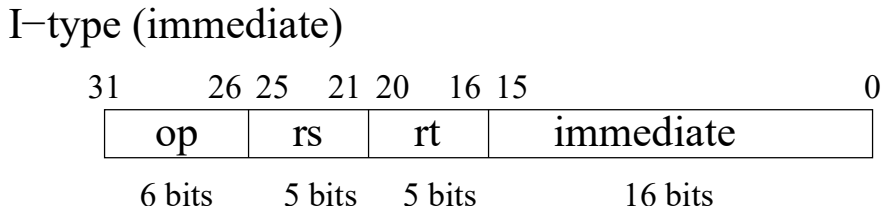
Note the use of MUX's (with control inputs) to add functionality.

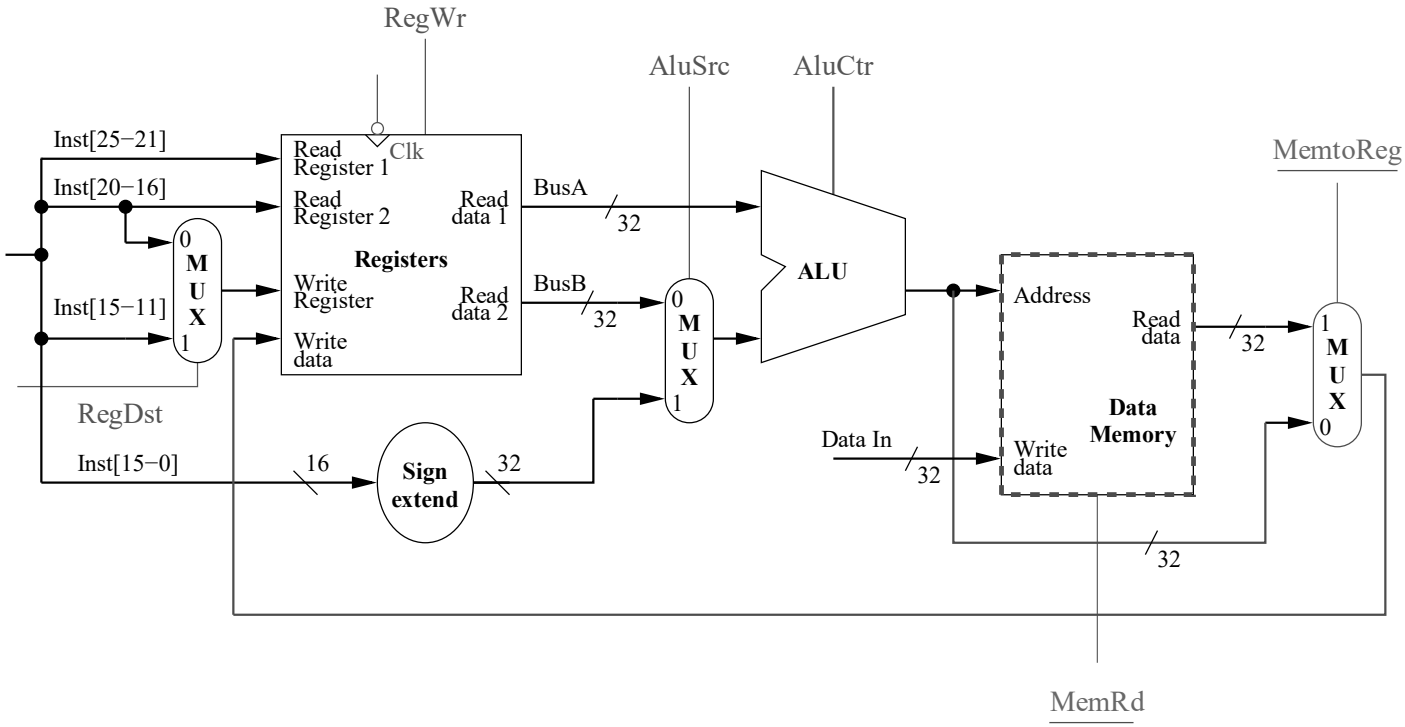
Datapath for the Load instruction

`lw rt, rs, imm16`

- `Addr ← R[rs] + SignExt(imm16)` Calculate the memory address
- `R[rt] ← Mem[Addr]` load the data into register `rt`

This is also an immediate type instruction:



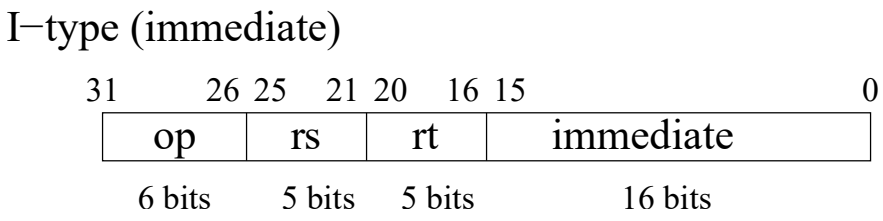


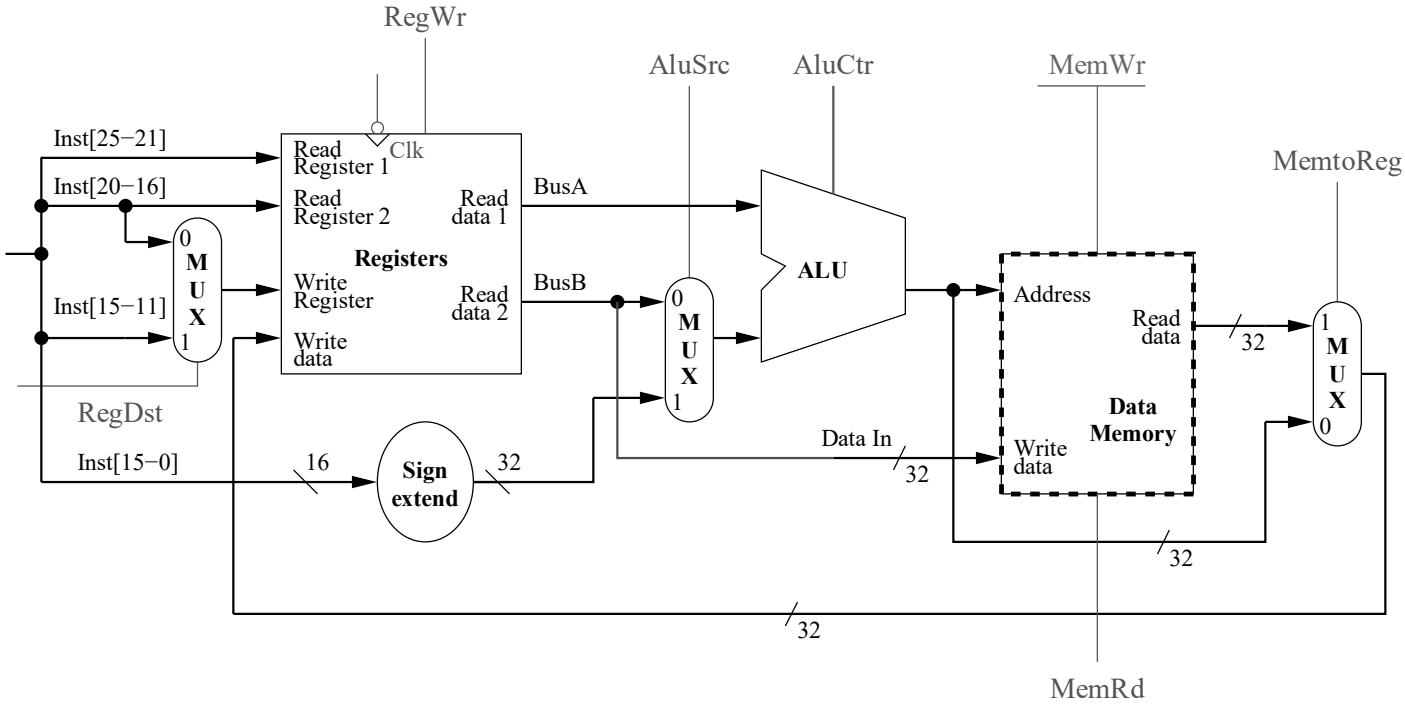
Datapath for the Store instruction

`sw rt, rs, imm16`

- `Addr ← R[rs] + SignExt(imm16)` — Calculate the memory address
- `Mem[Addr] ← R[rt]` — Store the data from register `rt` to memory

This is also an immediate type instruction:





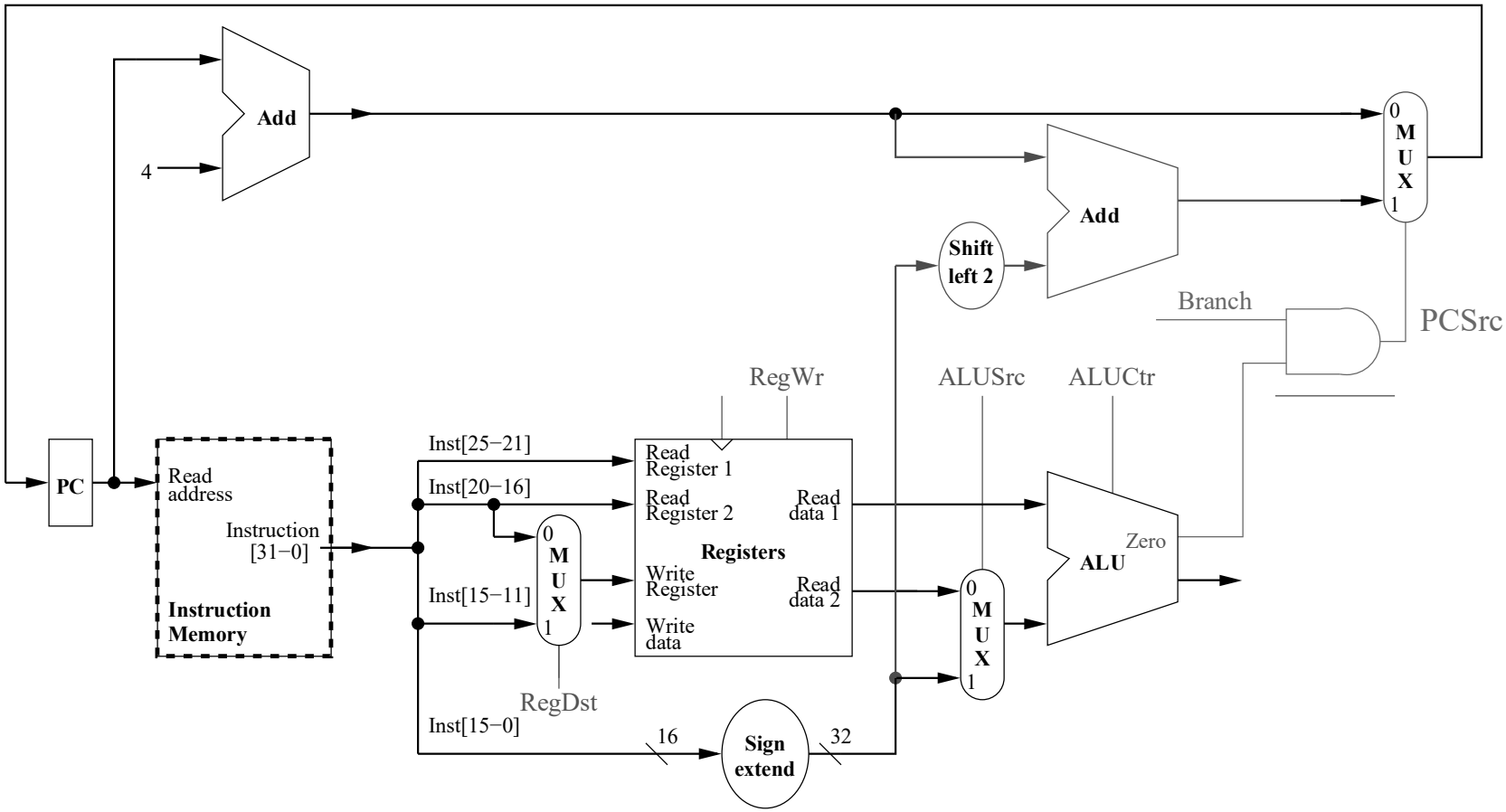
Datapath for the Branch instruction

`beq rt, rs, imm16`

- `Cond` $\leftarrow$ `R[rs] - R[rt]` — Calculate the branch condition
- `if (Cond eq 0)` `PC` $\leftarrow$ `PC + 4 + (SignExt(imm16)` $\times$ `4)` — calculate the address of the next instruction
- `else PC` $\leftarrow$ `PC + 4`

This is also an immediate type instruction.

In the `load` and `store` instructions, the ALU was used to calculate the address for data memory.
It is possible to do this for the `branch` instructions as well, but it would require first performing the comparison using the ALU, and then using the ALU to calculate the address.
This would require two clock periods, in order to sequence the operations correctly.
A faster implementation would be to provide another adder to implement the address calculation. This is what we will do, for the present example.

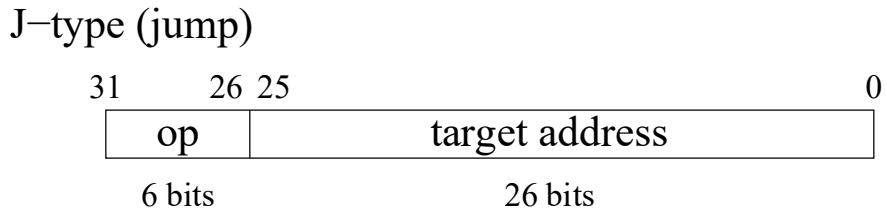

Add
4
Add
Shift
left 2
0
M
U
X
1
Branch
PCSrc
RegWr
ALUSrc
ALUCtr
PC
Read
address
Instruction
[31−0]
Instruction
Memory
Inst[25−21]
Inst[20−16]
Inst[15−11]
Inst[15−0]
0
M
U
X
1
RegDst
Read
Register 1
Read
Register 2
Registers
Write
Register
Write
data
Read
data 1
Read
data 2
0
M
U
X
1
Zero
ALU
16
Sign
extend
32

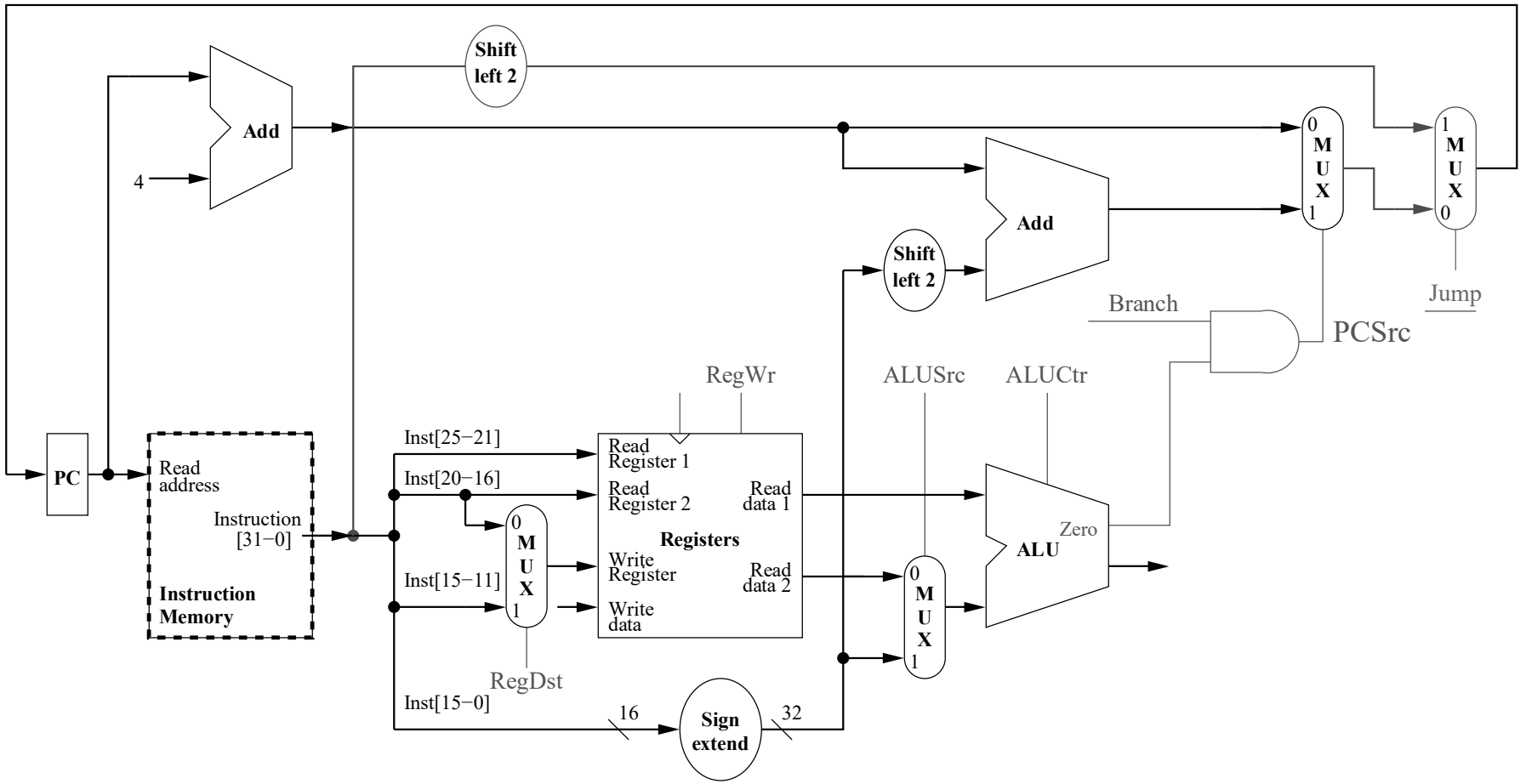
Datapath for the Jump instruction

`j target`

- `PC<31:2>` $\leftarrow$ `PC<31:28>` `concat target<25:0>` — Calculate the jump address by concatenating the high order 4 bits of the PC with the target address

Here, the address calculation is just obtained from the high order 4 bits of the PC and the 26 bits (shifted left by 2 bits to make 28) of the target address.
The additions to the datapath are straightforward.




Shift
left 2
Add
4
0
M
U
X
1
1
M
U
X
0
Add
Shift
left 2
Branch
PCSrc
Jump
RegWr
ALUSrc
ALUCtr
PC
Read
address
Instruction
[31−0]
Instruction
Memory
Inst[25−21]
Inst[20−16]
Inst[15−11]
Inst[15−0]
0
M
U
X
1
RegDst
Read
Register 1
Read
Register 2
Registers
Write
Register
Write
data
Read
data 1
Read
data 2
0
M
U
X
1
Zero
ALU
16
Sign
extend
32

Putting it together

The datapath was shown in segments, some of which built on each other.

Required control signals were identified, and all that remains is to:

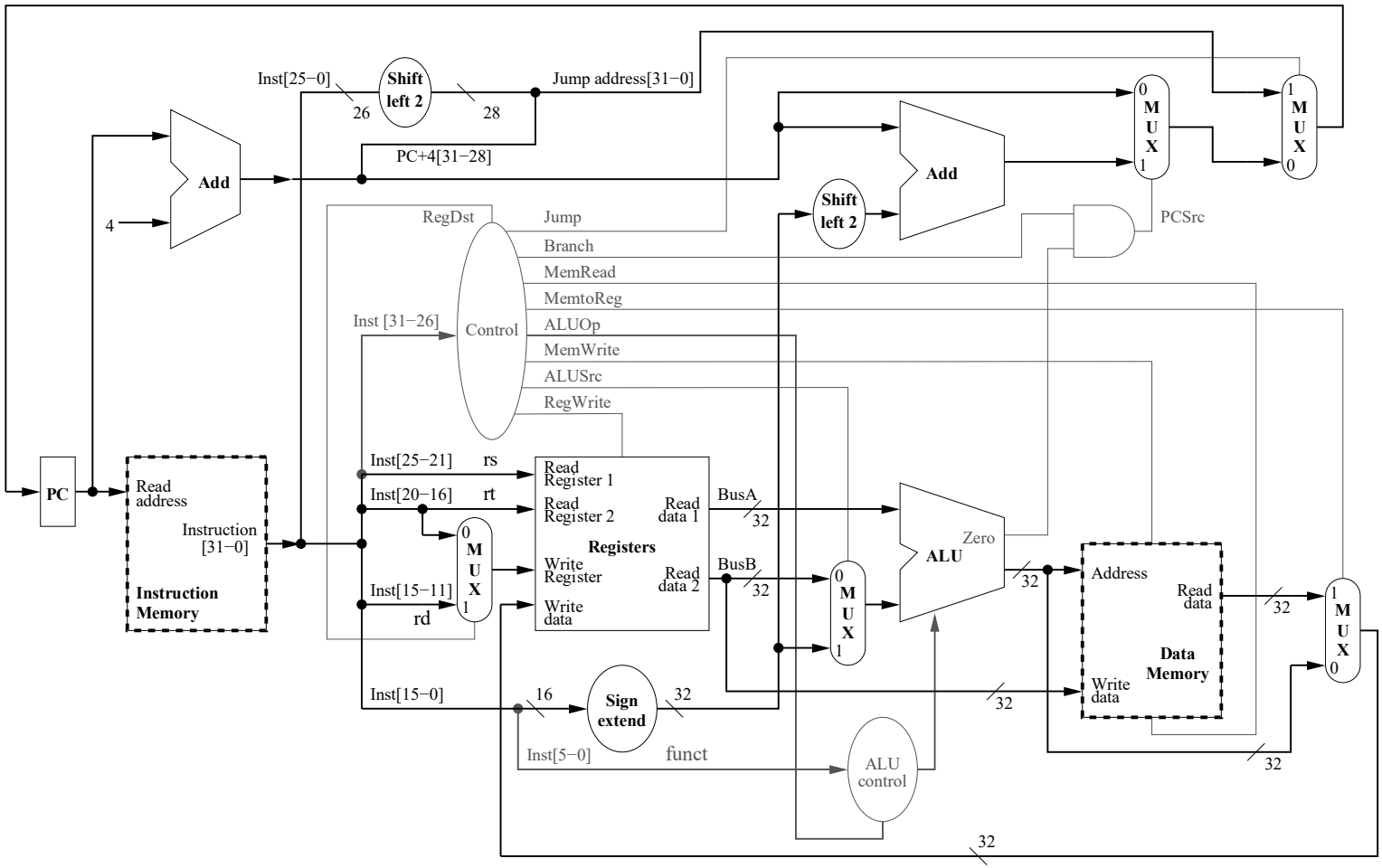
1. Combine the datapath elements
2. Design the appropriate control signals

Combining the datapath elements is rather straightforward, since we have mainly built up the datapath by adding functionality to accommodate the different instruction types.
When two paths are required, we have implemented both and used multiplexors to choose the appropriate results.

The required control signals are mainly the inputs for those MUX's and the signals required by the ALU.

The next slide shows the combined data path, and the required control signals.

The actual control logic is yet to be designed.


Inst[25−0]
Shift left 2
26
28
Jump address[31−0]
PC+4[31−28]
Add
4
0 M U X 1
1 M U X 0
Add
Shift left 2
PCSrc
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Inst [31−26]
PC
Read address
Instruction [31−0]
Instruction Memory
Inst[25−21]
rs
Inst[20−16]
rt
Inst[15−11]
rd
Inst[15−0]
Read Register 1
Read Register 2
Registers
Write Register
Write data
Read data 1
Read data 2
BusA
BusB
32
Zero
ALU
Address
Read data
Data Memory
Write data
16
Sign extend
Inst[5−0]
funct
ALU control

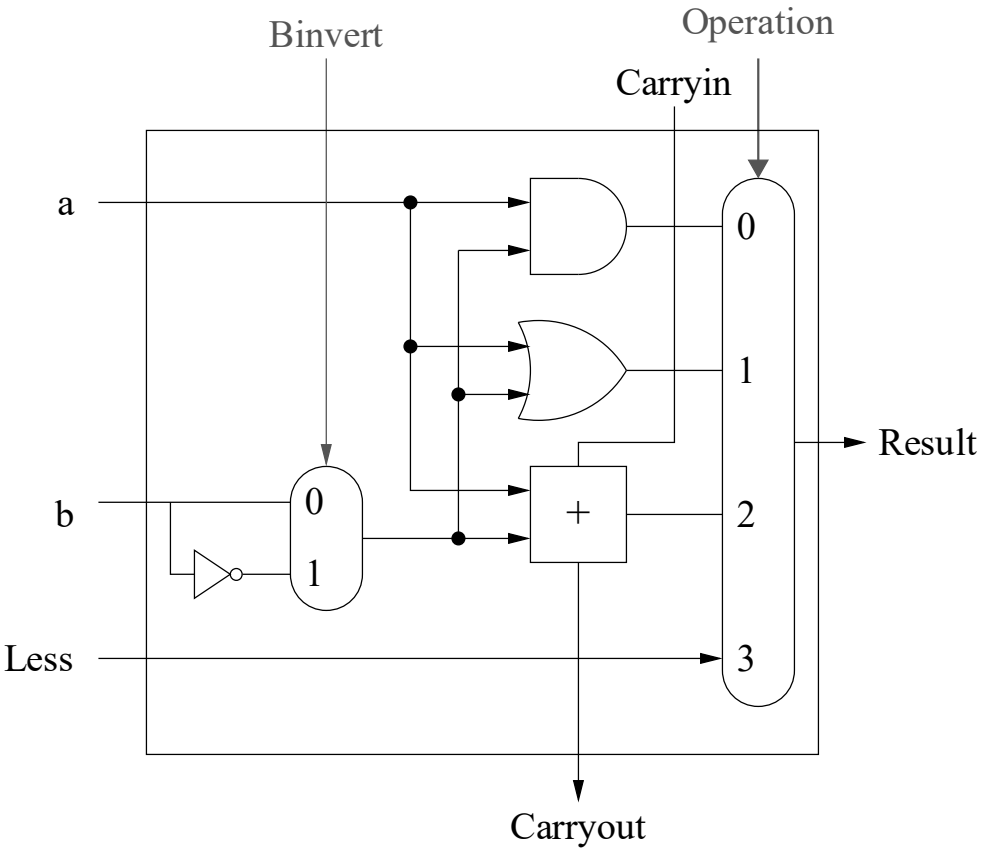
Designing the control logic

The control logic depends on the details of the devices in the control path, and on the individual bits in the op code for the instructions. The arithmetic and logic operations for the r-type instructions also depend on the `funct` field of the instruction.

The datapath elements we have used are:

- a 32 bit ALU with an output indicating if the result is zero
- adders
- MUX's (2 line to 1-line)
- a 32 register $\times$ 32 bits/register register file
- individual 32 bit registers
- a sign extender
- instruction memory
- data memory

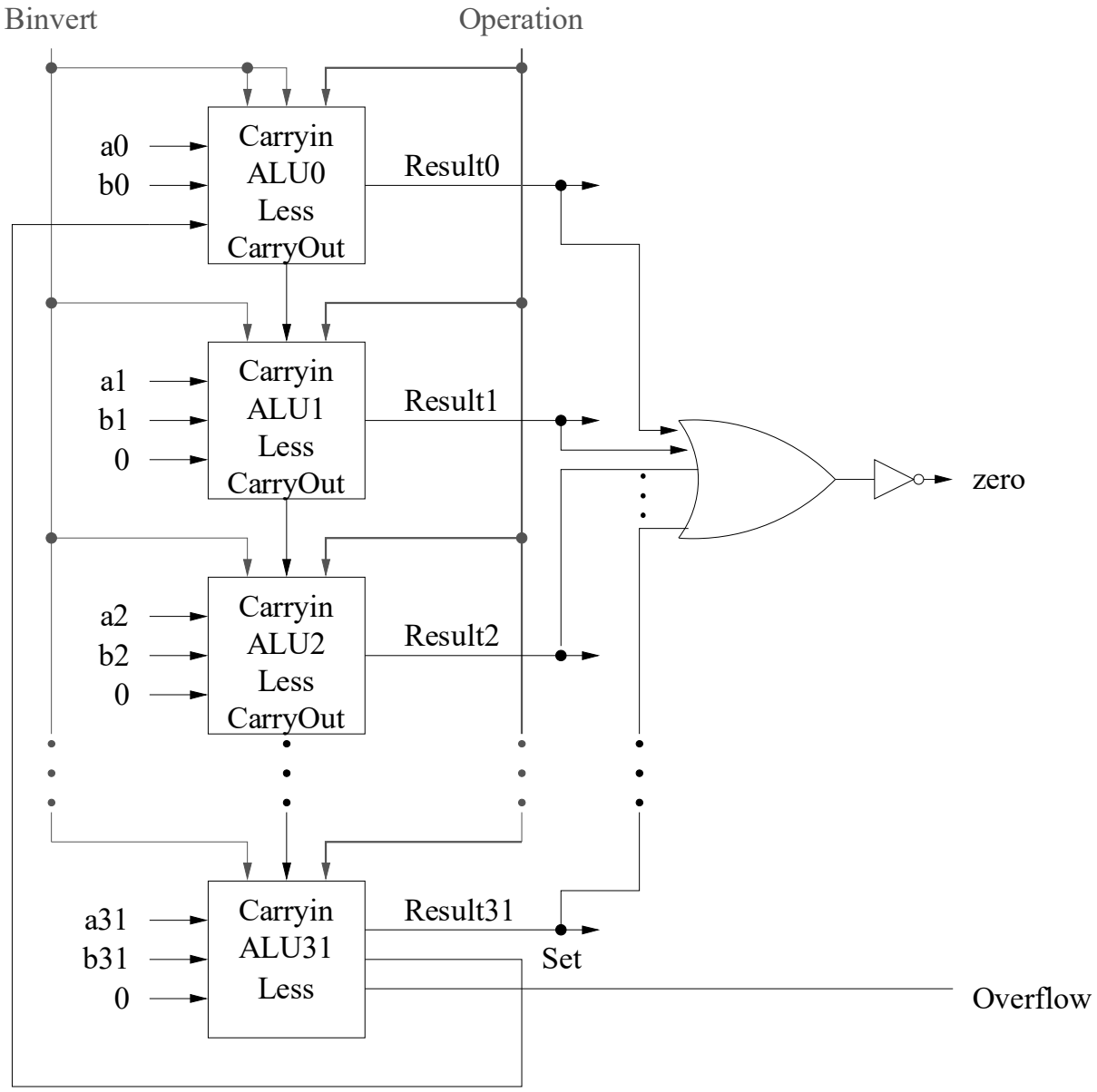
The ALU — a single bit



Note that there are three control bits; the single bit `Binvert`, and the two bit input to the MUX, labeled `Operation`.

The ALU performs the operations `and`, `or`, `add`, xor and `subtract`.

The 32 bit ALU



ALU control lines	Function
000	`and`
001	`or`
010	`add`
110	`subtract`
111	`xor`

We will design the control logic to implement the following instructions (others can be added similarly):

Name	Op-code					
	Op5	Op4	Op3	Op2	Op1	Op0
R-format	0	0	0	0	0	0
`lw`	1	0	0	0	1	1
`sw`	1	0	1	0	1	1
`beq`	0	0	0	1	0	0
`j`	0	0	0	0	1	0

Note that we have omitted the immediate arithmetic and logic functions.

The `funct` field will also have to be decoded to produce the required control signals for the ALU.

A separate decoder will be used for the main control signals and the ALU control. This approach is sometimes called *local decoding*. Its main advantage is in reducing the size of the main controller.

The control signals

The signals required to control the datapath are the following:

- `Jump` — set to 1 for a `jump` instruction
- `Branch` — set to 1 for a `branch` instruction
- `MemtoReg` — set to 1 for a `load` instruction
- `ALUSrc` — set to 0 for r-type instructions, and 1 for instructions using immediate data in the ALU (`beq` requires this set to 0)
- `RegDst` — set to 1 for r-type instructions, and 0 for immediate instructions
- `MemRead` — set to 1 for a `load` instruction
- `MemWrite` — set to 1 for a `store` instruction
- `RegWrite` — set to 1 for any instruction writing to a register
- `ALUOp` (k bits) — encodes ALU operations except for r-type operations, which are encoded by the `funct` field

For the instructions we are implementing, `ALUOp` can be encoded using 2 bits as follows:

`ALUOp[1]`	`ALUOp[0]`	Instruction
0	0	memory operations (`load, store`)
0	1	`beq`
1	0	r-type operations

The following tables show the required values for the control signals as a function of the instruction op codes:

Instruction	Op-code	RegDst	ALUSrc	MemtoReg	Reg Write
r-type	0 0 0 0 0 0	1	0	0	1
lw	1 0 0 0 1 1	0	1	1	1
sw	1 0 1 0 1 1	x	1	x	0
beq	0 0 0 1 0 0	x	0	x	0
j	0 0 0 0 1 0	x	x	x	0

Instruction	Op-code	Mem Read	Mem Write	Branch	ALUOp[1:0]	Jump
r-type	0 0 0 0 0 0	0	0	0	1 0	0
lw	1 0 0 0 1 1	1	0	0	0 0	0
sw	1 0 1 0 1 1	0	1	0	0 0	0
beq	0 0 0 1 0 0	0	0	1	0 1	0
j	0 0 0 0 1 0	0	0	0	x x	1

This is all that is required to implement the control signals; each control signal can be expressed as a function of the op-code bits.
For example,
RegDst $= \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \overline{\text{Op1}} \cdot \overline{\text{Op0}}$
ALUSrc $= \text{Op5} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}$

All that remains is to design the control for the ALU.

The ALU control

The inputs to the ALU control are the `ALUOp` control signals, and the 6 bit `funct` field.

The `funct` field determines the ALU operations for the r-type operations, and `ALUOp` signals determine the ALU operations for the other types of instructions.

Previously, we saw that if `ALUOp[1]` was 1, it indicated an r-type operation. `ALUOp[0]` was set to 0 for memory operations (requiring the ALU to perform an `add` operation to calculate the address for data) and to 1 for the `beq` operation, requiring a subtraction to compare the two operands.

The ALU itself requires three inputs.

The following table shows the required inputs and outputs for the instructions using the ALU:

Instruction	ALUOp	`funct`	ALU operation	ALU control input
`lw`	0 0	x x x x x x	add	0 1 0
`sw`	0 0	x x x x x x	add	0 1 0
`beq`	0 1	x x x x x x	subtract	1 1 0
`add`	1 0	1 0 0 0 0 0	add	0 1 0
`sub`	1 0	1 0 0 0 1 0	subtract	1 1 0
`and`	1 0	1 0 0 1 0 0	AND	0 0 0
`or`	1 0	1 0 0 1 0 1	OR	0 0 1
`xor`	1 0	1 0 1 0 1 0	XOR	1 1 1