Computer Architecture Project - Phase 1

Team 1

Ahmad Khaled	S01 BN 03
Zeinab Rabie	S01 BN 26
Sara Maher	S01 BN 28
Abdelrahman Mohamed	S02 BN 03

Introduction

This project is an assembler for a custom PDP-11 like CPU. CPU architecture is available in reference 1. The assembler takes an assembly file as input and generates its equivalent binary file as an output. Instruction set is available in appendix A.

File Structure

The assembly file consists of two regions. First the code region which contains all the code. The second region contain the variables definitions. Variables regions must always be at the end of the file after the code regions. If any code is exists after the variables regions the assembler would report an error.

Line Structure

An assembly line must be at maximum 255 characters. The line structure is differs according to the regions.

Code Region

Every code lines consists of four fields:

1. Label
2. Instruction
3. Operands
4. Comment

Labels could appear in two ways. First in the same line followed the other three fields in which case label name could end with a colon or it could be ignored. The second case is a separate line in which case it must be followed by a colon and the next line must begin with a valid instruction (labels nesting is not allowed). Labels must follow the following rules:

- Starts with an alphabetic letter or an underscore
- The rest of the letter could only be letters (upper or lower), numbers or underscores (any other characters are considered invalid and is considered an error)
- Names are case sensitive
- Can't be an instruction (case doesn't matter in this case), any variable or label declared before
- Number characters must be less than 255 character

Instruction must be a valid instruction from the instruction set in appendix A. Instructions are case insensitive.

Operands must follow the instruction if the instruction takes any operands. In case of two operands the operands must be separated by a comma (spaces are neglected). Operands could be any of the following:

1. A CPU register (R0, R1, R2, …, R7)
2. A variable

3. A label (only for branch instructions and JSR)
4. An immediate value which must be an integer between -32768 →32767 and must be preceded by # (e.g. #137)

Operands have addressing modes every mode represent a different interpretation for the operand. There is 8 different addressing modes:

Mode	Syntax	Operation
Register direct	Ri, where i ∈ {0,1,2,3,4,5,6,7}	Op ← Ri

Direct auto increment	(Ri)+, where i ∈ {0,1,2,3,4,5,6,7}	Op ← [Ri]
		Ri ←Ri + 1

Direct auto decrement	-(Ri), where i ∈ {0,1,2,3,4,5,6,7}	Ri ←Ri - 1
		Op ← [Ri]

Direct indexed	X(Ri), where i ∈ {0,1,2,3,4,5,6,7} and X is an integer between -32768 →32767 or a variable	Op ← [[PC] + Ri]
Register indirect	@Ri, where i ∈ {0,1,2,3,4,5,6,7}	Op ← [Ri]

Indirect auto increment	@(Ri)+, where i ∈ {0,1,2,3,4,5,6,7}	Op ←[[Ri]]
		Ri ←Ri + 1

Indirect auto decrement	@-(Ri), where i ∈ {0,1,2,3,4,5,6,7}	Ri ←Ri - 1
		Op ←[[Ri]]

Indirect indexed	@X(Ri), where i ∈ {0,1,2,3,4,5,6,7} and X is an integer between -32768 →32767 or a variable	Op ← [[[PC] + Ri]]

Note: variables could be used directly as operands in which case it would be considered indexed mode were X would be the offset between the PC and the variable address and Ri would be the PC. Immediate values are treated as register indirect mode where Ri is R6 (PC).

Comments must start with a semicolon and anything after the semicolon is completely ignored by the assembler.

Variables Region

Lines in variables region is divided into three fields:

1. The keyword DEFINE (case insensitive)
2. Variable name
3. Data

The keyword DEFINE is used to declare variables so it's always the first word in any variable declaration. Variable name follow the same rules as labels names. Data could be a single value or an array of values that are separated by comma (spaces are ignored). Data values must be an integer number between `-32768` →`32767`.

Other Lines

Assembler also allow empty lines and comment lines. Empty lines doesn't contain any characters at all. Comment line is a line that starts with a semicolon. Both of these lines are absolutely ignored by the assembler.

Assembling

The assembling process is performed by pass report the assembly file twice. The reason for that that variables are defined at the end of the file in the variables region so the assembler can't parse without knowing them. Same case also may happen with labels as labels may be used before there declaration which may result in errors or the programmer would be restricted to define all the labels before using them.

First Pass

The goal of this pass is make a labels table and a variables table and some minor processing on the number of fields in the line to to reduce the load of the second pass (check for trivial errors like code in variables region, label in a seperate line without a colon, etc). After this pass is done the assembler points back to the beginning of the file and start the parsing process.

Second Pass

In this pass every line line is parsed and replaced with it's binary code. Binary codes are stored in a list. Instruction may two or three words depending on the instruction and addressing mode. In case of any error found the assembler would report that error and terminate. Single value variable is stored as a single word at it respective address. In case of an array the values are stored in separate consecutive words with the same order. The address of any variable is the next address after the variable before it. The first variable address is the word after the last instruction in the code region.

Instruction may take more than one word so labels and variables positions change in the assembling process. To fix this issue any instruction that uses either of them is initially compiled as the code of the instruction with garbage value for the label or variable. The address of that instruction is then stored in a table with the variable of label name (each in a separate tables). The positions of the variables and labels are updated in the parsing process to their true positions. After the parsing end instructions stored in the fixing tables are fixed with the required addresses or values.

Appendix A

TWO-OPERAND INSTRUCTIONS

Instruction	Operation
MOV	Dst ← [Src]
ADD	Dst ← [Dst] + [Src]
ADC (ADD with carry)	Dst ← [Dst] + [Src] + C
SUB	Dst ← [Dst] – [Src]
SBC (SUB with carry)	Dst ← [Dst] – [Src] – C
AND	Dst ← [Dst] AND [Src]
OR	Dst ← [Dst] OR [Src]
XNOR	Dst ← [Dst] XNOR [Src]
CMP (compare)	[Dst] – [Src] Neither of the operands are affected

Operands may be registers or variables in any addressing mode. Immediate values are allowed only for source operand.

ONE-OPERAND INSTRUCTIONS

Instruction	Operation
INC (Increment)	Dst ← [Dst] + 1
DEC (Decrement)	Dst ← [Dst] – 1
CLR (Clear)	Dst ← 0
INV (Inverter)	Dst ← INV([Dst])
LSR (Logic Shift Right)	Dst ← 0 || [Dst]15->1
ROR (Rotate Right)	Dst ← [Dst]0 || [Dst]15->1
RRC (Rotate Right with Carry)	Dst ← C || [Dst]15->1

ASR (Arithmetic Shift Right)	Dst ← [Dst]15 || [Dst]15->1
LSL (Logic Shift Left)	Dst ← [Dst]14->0 || 0
ROL (Rotate Left)	Dst ← [Dst]14->0 || [Dst]15
RLC (Rotate Left with Carry)	Dst ← [Dst]14->0 || C

Operand is any register or variable in any addressing mode.

BRANCH INSTRUCTIONS

Instruction	**Branch Condition**
BR (Branch unconditionally)	None
BEQ (Branch if equal)	Z = 1
BNE (Branch if not equal)	Z = 0
BLO (Branch if Lower)	C = 0
BLS (Branch if Lower or same)	C = 0 or Z = 1
BHI (Branch if Higher)	C = 1
BHS (Branch if Higher or same)	C = 1 or Z = 1

Only one operand that must be a label. The maximum number of lines between label and the instruction must be between -256 →255.

NO-OPERAND INSTRUCTIONS

Instruction	**Operation**
HLT (Halt)	Stop the processor
NOP (No Operation)	No operation is performed continue code

SUBROUTINE INSTRUCTIONS

Instruction	**Syntax**	**Operation**

JSR (Jump to subroutine)	JSR Label	SP ← [SP] – 1 [SP] ← [PC] [PC] ← [Address]
RTS (Return from subroutine)	RTS	[PC] ← [SP] SP ← [SP] + 1
IRET	IRET	pop flags from stack return back to PC

JSR instruction takes a label as operand. The label must be within the first 2048 line of the program.

References

[1] CPU Architecture