## ***Fail to Plan, Plan to Fail***

## **Project: Design of a Five Stage Pipelined MIPS-like Processor**

Instructor: Dr. Yifeng Zhu

Fall, 2017

**Project Team**

This is a team project. All teams should comprise of no more than **TWO** members. The success of your project will depend greatly on how effectively and cooperatively your group works as a team.

**Objectives**

1. Implements the pipelined MIPS RISC processor core. The following figure is one example implementation.
2. Learn to work as a team to carry out a complex design task requiring task partitioning, effective communication, and cooperation.
3. Design and conduct experiments, as well as to analyze and interpret experimental data.

**What to hand in on December 7, Thursday (the last class)**

1. Whole project codes.
2. A project report.
3. Project presentation slides. You are required to make a 5-minute project presentation during the last class.

**Special note:**

1. The handout (MIPS Reference Data Sheet) has a typo.

* sll R[rd] = R[rs]<<shamt; It should be R[rd] = R[**rt**]<<shamt *(change rs to rt)*
* srl R[rd] = R[rs]>>shamt; It should be R[rd] = R[**rt**]>>shamt *(change rs to rt)*

As a result, each operand input of the ALU needs a MUX.

1. **Memory Configuration**

Assemblers produce object files from assembly codes. An object file typically includes several distinct sections, such as text segment and data segment. The text segment includes the machine language codes to be executed and the data segment contains a binary representation of initialized data used by the program.

**Memory Layout**

Systems based on the MIPS processors typically divide the memory into three parts: system space (typically the top of memory space), and user space, and reserved space (typically the bottom of memory space). Access the system space and reserved space typically required some privilege. A user’s program is usually placed into the user space. The user space has four distinct parts,

1. *Text segment* (starting at address 0x04000000), which holds the program’s machine language codes,
2. *Static data segment* (starting at address 0x10000000), which contains data whose size is known to the compiler and whose lifetime is the program’s entire execution.
3. *Dynamic data segment* (starting at address 0x10008000), which is allocated by the program as it executes. For example, in C programs, a *malloc()* function will return a new block of memory allocated in the dynamic data segment.
4. *Stack segment* (starting at address 0x7fffffff), which is used by a program to hold procedure call frames. Note that the stack segment grows towards the bottom of the memory space, while the dynamic data segment grows towards the top of the memory space. This is because the system needs to separate these two dynamically expandable segments, including the dynamic data segment and the stack segment, as far apart as possible.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 0xffffffff | Memory map limit address | | 0xffffffff | Kernel space highest address | | 0xfffeffff | Kernel data segment limit address | | 0x8ffffffc | Kernel text limit address | | 0x80000180 | Exception handler address | | 0x80000000 | Kernel space base address | | 0x7fffffff | User space high address | | 0x7fffffff | Data segment limit address | | 0x7fffffff | Stack base address | | 0x7fffffff | Stack pointer $sp | | 0x10040000 | Stack limit address | | 0x10040000 | Heap limit address | | 0x10040000 | Heap base address | | 0x10010000 | .data base address | | 0x10008000 | global pointer $gp | | 0x10000000 | data segment base address | | 0x10000000 | .extern base address | | 0x0ffffffc | text limit address | | 0x00400000 | .text base address | |  |
| **Figure 1. Memory Layout** | |

**.data and .text Sections**

The following gives a simple example MIPS assembly program that calculates the sum of an integer array. In this special program, we use “0” to indicate the end of this array (Note: this is not a generic way).

.data # following words in data segment

A: .word 1 2 3 4 5 6 7 8 9 10 0 # array of 10 words in location A

# The last ZERO indicates the end of this integer array.

.globl sum2 # define global label main

.text # text segment begins here

main:

la $t0, A # la: load address of A in $t0.

# la is a pseudo-instruction, which is translated into lui

add $s0, $zero, $zero # s0 = 0

loop:

lw $t2, 0($t0) # $t2 = Mem[$t0 + 0]

add $s0, $s0, $t2 # $s0 = $s0 + $t2

addi $t0, $t0, 4 # $t0 = $t0 + 4

bne $t2, $zero, loop # if ($t2 != 0) goto loop

nop # Branch Delay Slot

exit: j exit # dead loop; program stops here

nop

**Program 1: Calculate the sum of an array**

This simple MIPS program has two segments, indicated by ***.text*** and ***.data*** directive, respectively. The .data directive tells the assembler to store the string in the program’s data segment, and the .text directive tells the assembler to store the instructions in its text segment.

***.globl sum2*** declares that label ***sum2*** is global and can be referenced from other files. For example, another program can call the sum2 function implemented in this assembly program.

The following gives a few example of declaring data in the data segment.

.data

str: .asciiz "the answer = " # str is a string

m: .word 1 # m is a 32-bit integer initialized to 1

a: .byte 1 2 3 4 5 6 # a is an array of six values in successive bytes of memory

c .space 32 # allocate 32 bytes of space in data segment for c

d: .double 1.1, 2.2, 3.3, 4.4 # define an array of 4 floating-point double precision

# (64-bit) numbers in successive memory locations,

f: .float 1.1, 2.2, 3.3, 4.4 # define an array of 4 floating-point single precision

# (32-bit) numbers in successive memory locations.

dd: .dword 15 # a 64-bit integer, equivalent to long long in C

**Program 2: Examples of data declaration**

**Heap and Stack**

The heap and stack, as shown in Figure 1, are significant areas of the system’s address space. These are not known to the assembler or linker in the same way as the .text or .data sections. Typically the stack and the heap are initialized and maintained by the runtime system. In MIPS, the stack pointer ($sp) is initialized to 0x7fffffff on default while the heap base address is 0x10040000.

1. **Procedure Call Convention**

The MIPS CPU contains 32 general-purpose registers that are numbered 0–31.

* Register $0 always contains the hardwired value 0. Registers $at (1), $k0 (26), and $k1 (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
* Registers $a0–$a3 (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1(2, 3) are used to return values from functions.
* Register $v0 ($2) is used to return an integer value or a memory pointer from a function. $v1 ($2) is also reserved when returning a 64-bit value.
* Registers $t0–$t9 (8–15, 24, 25) are caller-saved registers that are used to hold temporary quantities that need not be preserved across call.
* Registers $s0 - $s7 (16 - 23) are callee-saved registers that hold long-lived values that should be preserved across calls. A function needs to save any of these registers into the stack before they are used in this function.
* Register $ra ($31) stored the return address for subroutine. The jal instruction writes register $ra the return address from a procedure call.
* Register $gp (28) is a global pointer that points to the middle of a 64K block of memory in the static data segment.
* Register $sp (29) is the stack pointer, which points to the last location on the stack. If a 32-bit value is pushed into the stack, then $sp will be reduced by 4. The stack grows down, with higher memory address at the top.
* Register $fp (30) is the frame pointer, which points to the location where the stack pointer was, just before a callee function moved the stack pointer for its own local variables. The $fp is kept fixed within this callee function. When exiting from this function, we just need to set the stack pointer $sp to $fp, which effectively pop all items added by this callee function.

1. **Instruction Set**

You are to design and implement a five-stage 32-bit pipeline MIPS processor that can carry out the instructions specified in Tables 1, 2 and 3. The tables show the instruction subset to be implemented and examples of instruction encoding; the meaning of the instructions remains the same as in MIPS.

Table 1: Basic Instruction Set (R-type)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| R-Type Instruction | | | | | | |
| Instruction | OPCode | RS | RT | RD | Shamt | Funct |
| **add** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100000 |
| **addu** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100001 |
| **sub** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100010 |
| **subu** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100011 |
| **and** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100100 |
| **or** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100101 |
| **nor** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 100111 |
| **slt** $3, $2, $1 | 000000 | 00010 | 00001 | 00011 | 00000 | 101010 |
| **sll** $3, $2, 1 | 000000 | 00000 | 00010 | 00011 | 00001 | 000000 |
| **srl** $3, $2, 1 | 000000 | 00000 | 00010 | 00011 | 00001 | 000010 |
| **sra** $3, $2, 1 | 000000 | 00000 | 00010 | 00011 | 00001 | 000011 |
| **jr** $4 | 000000 | 00100 | 00000 | 00000 | 00000 | 001000 |
| **mul** |  |  |  |  |  |  |
| **nop** | 000000 | 00000 | 00000 | 00000 | 00000 | 000000 |

Note:

* *SLT rd, rs, rt*. To record the result of a less-than comparison. Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR rd. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).
* *nop*: NOP (No Operation) is the assembly idiom used to denote no operation. The actual instruction is interpreted by the hardware as SLL r0, r0, 0.
* *jr rs*, To execute a branch to an instruction address in a register. PC ← rs. Jump to the effective target address in GPR rs. Execute the instruction following the jump, in the branch delay slot, before jumping.
* *sra rd, rt, sa*, (shift word right arithmetic) To execute an arithmetic right-shift of a word by a fixed number of bits. The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.
* *srl rd, rt, sa*, (shift right logic)To execute a logical right-shift of a word by a fixed number of bits. The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.
* *sll* and *srl* (shift left logic and shift right logic). The textbook specifies these two instructions incorrectly. The textbook says R[*rd*] = R[*rs*] << *sa* or R[*rd*] = R[*rs*] >> *sa*. The MIPS standards says R[*rd*] = R[*rt*] << *sa* or R[*rd*] = R[*rt*] >> *sa*. The source register should be *rt*, instead of *rs*.

Table 2: Basic Instruction Set (I-type)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I-Type Instruction | | | | |
| Instruction | OPCode | RS | RT | Immediate |
| **andi** $2, $at, 20 | 001100 | 00001 | 00010 | 0000000000010100 |
| **ori** $2, $at, 20 | 001101 | 00001 | 00010 | 0000000000010100 |
| **slti** $2, $at, 20 | 001010 | 00001 | 00010 | 0000000000010100 |
| **addi** $2, $at, 20 | 001000 | 00001 | 00010 | 0000000000010100 |
| **addiu** $2, $at, 20 | 001001 | 00001 | 00010 | 0000000000010100 |
| **beq** $at, $2, 20 | 000100 | 00001 | 00010 | 0000000000010100 |
| **bne** $at, $2, 20 | 000101 | 00001 | 00010 | 0000000000010100 |
| **bgez** $at, 20 | 000001 | 00001 | 00000 | 0000000000010100 |
| **bgtz** $at, 20 | 000111 | 00001 | 00000 | 0000000000010100 |
| **lw** $at, 20($2) | 100011 | 00010 | 00001 | 0000000000010100 |
| **sw** $at, 20($2) | 101011 | 00010 | 00001 | 0000000000010100 |
| **lui** $at, 20 | 001111 | 00000 | 00001 | 0000000000010100 |

Note:

* ***bgtz rs, offset***. Branch on Greater Than Zero.
* ***bne rs, rt, offset***, Branch on Not Equal
* ***beq rs, rt, offset*** Branch on Equal

Table 3: Basic Instruction Set (J-type)

|  |  |  |
| --- | --- | --- |
| I-Type Instruction | | |
| Instruction | OPCode | Address |
| **j L5** | 000010 | 00000100000000000000001000 |
| **jal L5** | 000011 | 00000100000000000000001000 |

Note:

* ***j target***. **(jump)** Jump to the effective target address.
* ***jal target***. **(jump and link)** Jump to the effective target address. **jal** is commonly used for function calls. **jal** should really be called **laj** for “link and jump”:
  + Step 1 (link): Save address of next instruction into $ra
  + Step 2 (jump): Jump to the given label

**Pseudo-instructions**

While the binary codes provided in this project have already translated the pseudo instructions into real ones, you are still need to understand pseudo-instructions for processor debugging. A pseudo-instruction is not really a MIPS instruction but it is allowed in assembly language code. The assembler translates a pseudo instruction into MIPS code. This makes the job of writing this assembly language code easier. For example,

***la $t0, array***

where $t0 is pointer into an array of integers. If this integer array is assigned address 0x00aa0bb0, then the assembler will assign the value 0x00aa0bb0 to $t0. However, while the memory address is 32 bits, one machine instruction is only 32 bits wide. Accordingly, this assignment can be implemented by using one instruction. As a result, two instructions are required to implement this instruction.

lui $t0, 0x00aa # $t0 gets value 0x 00aa 0000

ori $t0, $t0, 0x0bb0 # $t0 gets value 0x 00aa 0bb0

Some other pseudo codes shown in the test program includes:

|  |  |
| --- | --- |
| li $s1, 7 | lui $at, 0  ori $at7, $at, 7 |
| ble $t2, $t3, label | slt $at, $t3, 10  beq $at, $0, label |
| blt $a0, $t0, label | slt $at, $a0, $t0  bne $at, $0, label |
| bgt $t1, $t2, label | slt $at, $t1, $t1  bne $at, $0, label |
| bge $t1, $t2, label | slt $at, $t1, $at0  beq $at, $0, label |
| la $t0, array | lui $t0, 0x00aa # $t0 gets value 0x 00aa 0000  ori $t0, $t0, 0x0bb0 # $t0 gets value 0x 00aa 0bb0 |
| # Branch if equal zero  beqz $t1, label | beq $t1, $0, lable |
| # Branch if not equal zero  bnez $t1, label | bne $t1, $0, lable |

In addition, some non-pseudo instructions are allowed to use immediate numbers, instead of registers, for the sake of conveniences.

|  |  |
| --- | --- |
| beq $t4, 0, skip\_add | ori $at, $0, 0  beq $at, $at2, 3 |
| bne $t5, 16, Loop | ori $at, $0, 16  bne $at, $at3, -10 |
| bge $t1, -100, label | slti $at, $t1, 0xffffff9c  beq $at, $0, label |
| bge $t1, 100000, label | lui $at, 0x00000001  ori, $at, $at, 0x000086a0  slt, $at, $t1, $at  beq $at, $0, label |
| bgt $t1, -100, label | addi $at, $t1, 0xffffff9c  slt, $at, $at, $9  bne $at, $0, label |
| bgt $t1, 100000, label | lui $at, 0x00000001  ori, $at, $at, 0x000086a0  slt, $at, $t1, $at  beq $at, $0, label |
| blt $t1, -100, label | slti $at, $t1, 0xffffff9c  bne $at, $0, label |
| blt $t1, 100000, label | lui $at, 0x00000001  ori, $at, $at, 0x000086a0  slt, $at, $t1, $at  bne $at, $0, label |

1. **Instruction Memory and Data Memory**

In this project, you will use the on-board ROM as the instruction memory and the on-board RAM as the data memory. In addition, we assume that the starting memory address of your instruction segment and data segment are 0x00400000 and 0x10000000 respectively. Accordingly, you need to make change to the instruction memory module and data memory module to accommodate these requirements.

In Quartus II, the shift instructions can be simply implemented as:

* To\_stdlogicvector(To\_bitvector(Binput) sll CONV\_INTEGER(shamt))
* To\_stdlogicvector(To\_bitvector(Binput) srl CONV\_INTEGER(shamt))

1. Example Implementation

The control signals are also pipelined and the following gives a simple example.



**RegDst**

**ALUOp[1:0]**

**ALUSrc**

**MemRead**

**MemWrite**

**Branch**

**RegWrite**

**MemtoReg**

****

**Figure 2. An Example of Five Stage Pipelined Processor**

This above diagram provides an example implementation. There are three controllers, including the main control, the hazard detection unit, and the forward unit. The zero input of the MUX before ID/EX pipeline registers are used to flush out the instruction at the ID/EX stage. The branch is resolved at the ID stage. The forwarding unit will take care of two different types of forwarding operations. The register file allows two operations per cycle: write in the first half cycle and read in the second half cycle.

1. **Test Programs**

We provide MIPS assembly programs to test your processor. You can use MARS to generate the binary instructions and data sections to initialize your data memory and instruction memory, as shown in Appendix 1.

1. **Milestones & Timeline**

You have approximately 6 weeks to complete the project.

* + Thanksgiving break begins Wednesday, November 22
  + Classes resume Monday, November 27

|  |  |  |  |
| --- | --- | --- | --- |
| **Due Date** | **Time** | **Milestones** | **% of Grades** |
| November 2 Thursday | 1  Week | Pipeline #1: Build the top pipeline scheme, support ADD and NOP and instructions   * You overall diagram should including all components shown in Figure 1, such as pipeline registers, three control units, ALU, register files, instruction memory and data memory, * Forwarding from MEM/WB to ALU, Forwarding from EX/MEM to ALU | **10%** |
| November 9 Thursday | 1  Week | Pipeline #2: pass all required R-type instructions | **20%** |
| November 21 Tuesday  (Thanksgiving Break Nov. 22-27) | 1.5 Weeks | Pipeline #3: pass all required I-Type and J-Type instructions   * Hazard detection (Read after load) and stall implementation * Branch is solved at ID state * Hazard detection and stall implementation | **20%** |
| November 30 Thursday | 1  Week | Pipeline #4: pass the test programs. You need to pass eight out of ten test programs. | **40%** |
| Dec. 7  Thursday | 1  Week | Submit report and do project presentation. | **10%** |

1. Bonus

You need to get the approval from the instructor before implementing your bonus. The bonus points depend on the difficulty of the extra implementation.

|  |  |
| --- | --- |
| **Bonus** | points |
| Cool stuff. You defined it. | TBD |

1. **Project Report Guideline**

This guideline only gives you some potential items you can address in your project report. You are NOT limited to those listed below.

* What are the task assignments between you and your project partner? How did you coordinate? Give a table that lists the task assignments and task completion status. Please make it clear and specific.
* What is the status of your processor design?
* What problems have you met upon designing, testing and simulation? How did you solve them?
* What specific design skills or design philosophies you have learned or used in this project?
* What are your suggestions for me to redesign this project for future ECE 473 students?

1. **Project Presentation Guideline**
   * Each team is required to make a 10-minute presentation of their project design during the class. A rule of thumb, 1 minute per slide. Thus you need to 10 power-point slides, excluding the first title slide.
   * Share the class with your experiences of designing and debugging your project? Please do not present how the pipeline works since the class already know it.
   * Any tips you might have to help your project design.
   * How did you and your teammate collaborate?
   * What are your suggestions for the instructor to improve the lab and projects of ECE 473?
   * An estimation of how many hours you have spent on the project.
2. **Project Submission**
3. You need to pre-build the FPGA configuration files for all test programs, i.e., when grading, we only need to download your configuration files directly to our Altera FPGA boards without the time-consuming building process. For example, after successfully compiling your project for test program 1, rename the name of your project ***sof*** file to “*test1.sof*”. During the grading, *test1.sof* can be directly downloaded to the FPGA boards without time-consuming re-compiling.
4. Submit your report and project files in one compressed file. We will copy the file from you.
5. You submission file should include:
   1. You project source codes.
   2. All pre-compiled test files (.sof files).
   3. Your project report
   4. The presentation slides
6. Email submissions are not acceptable due to the large file size and potentially large email delay. The email server might kill emails with a large attachment without notification.
7. **Tips: How to Succeed in this Project**

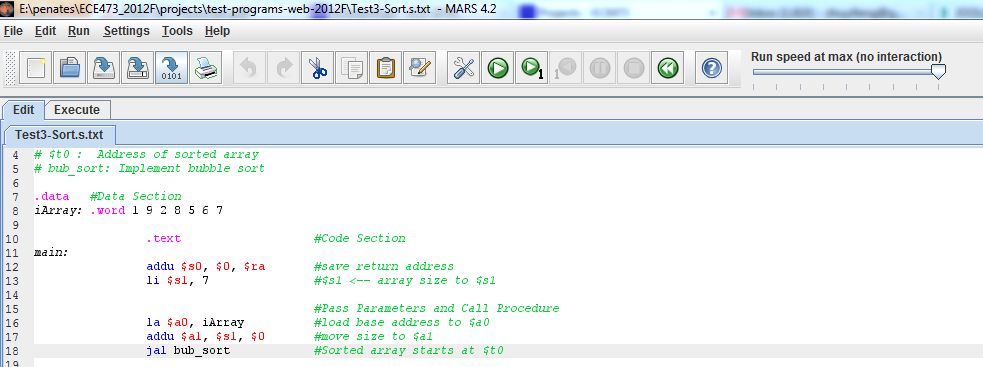
*Fail to Plan, Plan to Fail.*

Based on experience from past semesters, we have several suggestions for you to succeed in this project.

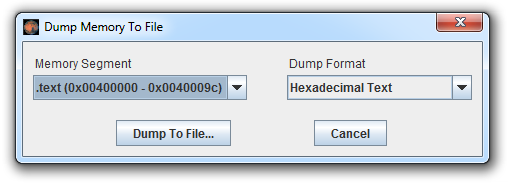
1. Learn VHDL or Verilog. Learn VHDL or Verilog by yourself before you look into the project. Once you know them well, the implementation will be much easier.
2. Work aggressively. This is a heavy project and time consuming. You can NOT finish this project with two or three days, even you spend the whole days on it.
3. Keep you design simple. For example, it is bad idea to allow many component modules to be able to modify the program counter. Complex cross connections between components modules make the debug difficulty.
4. Use vector form simulation to debug. Running your test code on the test boards can quickly tell you whether your design is correct or not. However, it is difficult for debugging due to the limited flexibility. Using waveform simulation can allow you to observe all data and control signals.
5. Make progress gradually. Make a good milestone plan and follow it strictly.
6. Small incremental testing. Make sure each instruction work correctly before you run the test programs. Create a very small program with one or two instructions to test your design.
7. Verification against SPIM or MARS simulator. Compare your running results against the Mars simulation instruction by instruction. Make sure the machine states are consistent.

**Appendix 1: Generate Binary Code and Data Segments to Test Your Processor**

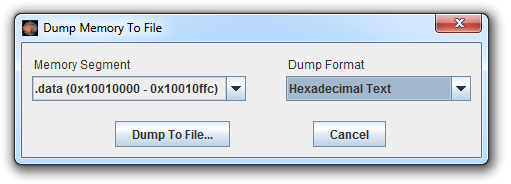
MARS 4.2 or above allow you to directly save the binary instruction and data into a text file.



Save the code segment in hexadecimal text format:



Save the data segment in hexadecimal text format



**Appendix 2: ECE473 Project Team Evaluation**

Please rate the contribution of each of your teammate, and your own, in terms of whether you/he did their FAIR SHARE on the group work. Contribution can take many forms, and may rest on ideas, writing, labor, leadership, etc.

I will consult this form in deciding whether to adjust the group grade up or down for an individual. For each member of the team, including yourself, assign a percent score that quantifies the relative contribution of his/her efforts (the total should be 100%). Also give a suggested grade increment, where a grade increment is the difference between, for example, a B and a B+.

Note that these increments should sum to about 0 – It makes no sense to give everyone a +1, because the superior performance will already be reflected in the group grade. Thus, for a team that worked well together and with full and equal participation, all the increments should be 0. Explain your reasoning for your score by describing each member’s contributions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Proportion of  Total Project | Suggested Grade Increment (circle one) |
| Yourself |  |  | +2 +1 0 -1 -2 |
| Team Mate |  |  | +2 +1 0 -1 -2 |
|  |  | Note: The sum should be 100% | Note:  The sum should be 0 |