Name ___________________________

Computer System Architecture
6.823 Quiz #1

October 7th, 2005

Professor Arvind

Dr. Joel Emer

Name:___________________

This is a closed book, closed notes exam.

80 Minutes

15 Pages

Notes:
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.

Writing name on each sheet	________	2 Points
Part A	________	22 Points
Part B	________	12 Points
Part C	________	12 Points
Part D	________	32 Points
TOTAL	________	**80 Points**

Name ____________________________

Part A: Addressing Modes on MIPS ISA (22 points)

Ben Bitdiddle is suspicious of the benefits of complex addressing modes. So he has decided to investigate it by incrementally removing the addressing modes from our MIPS ISA. Then he will write programs on the "crippled" MIPS ISAs to see what the programming on these ISAs is like.

For this problem, we assume 18-bit address space so that we can access any location in the memory by the 16-bit immediate field encoded in an instruction. (Remember that all data and instruction words are aligned. Don't worry about byte or half-word data accesses.)

Please refer to the MIPS instruction table on the last page (Appendix B) for each instruction's description and encoding.

Name ____________________________

Question 1 (6 points)

As a first step, Ben has discontinued supporting the displacement (base+offset) addressing mode; that is, our MIPS ISA only supports register indirect addressing (without the offset).

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.

```
LW R1, 16(R2)       ➔
```

Question 2 (8 points)

Now he wants to take a bolder step by completely eliminating the register indirect addressing. The new load and store instructions will have the following format:

```
LW R1, imm16   ; R1 <- M[{imm16,00}₂]
SW R1, imm16   ; M[{imm16,00}₂] <- R1
```

6	5	5	16
Opcode	**Rs**		**Offset**

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.
(Don't worry about branches and jumps for this question.)

```
LW R1, 16(R2)       ➔
```

Question 3 (8 points)

Ben is wondering whether we can implement a subroutine only using absolute addressing. He changes the original ISA such that all the branches and jumps take a 16-bit absolute address, and that `jr` and `jalr` are not supported any longer.

With the new ISA he decides to rewrite a piece of subroutine code from his old project. Here is the original C code he has written.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i=0; i<b; i++){
    result=result+a;
  }
}
```

The C code above is translated into the following instruction sequence on our original MIPS ISA. Assume that upon entry, R1 and R2 contain `b` and `a`, respectively. R3 is used for `i`, and R4 for `result`. By a calling convention, the 16-bit word-aligned return address is passed in R31.

```
Subroutine: xor  R4, R4, R4   ; result = 0
            xor  R3, R3, R3   ; i = 0
loop:       slt  R5, R3, R1
            bnez R5, L1       ; if (i < b) goto L1
return:     jr   R31          ; return to the caller
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
```

If you can, please rewrite the assembly code so that the subroutine returns without using a `jr` instruction (which is a register indirect jump). If you cannot, explain why.

Part B: Microprogramming (12 points)

In this question we ask you to implement a special return instruction, *return on zero* (**`retz`**), which uses the same encoding as a conditional branch instruction on MIPS:

`retz Rs, Rt`

6	5	5	16
`retz`	**Rs**	**Rt**	**unused**

`retz` instruction provides fast return from a subroutine call using **Rt** as the stack pointer. The instruction first tests the value of register **Rs**. If it is **not** zero, simply proceed to the next instruction at **PC+4**. If it is zero, the instruction does the following: (1) it reads the return address from memory at the address in register **Rt,** (2) increments **Rt** by 4, and (3) jumps to the return address.

For reference, we have included the actual bus-based datapath in Appendix A (Page 14) and a MIPS instruction table in Appendix B (Page 15)**.** You do not need this information if you remember the bus-based architecture from the online material. **Please detach the last two pages from the exam and use them as a reference while you answer this question.**

Question 4 (12 points)

Fill out Worksheet 1 for `retz` instruction. You should try to optimize your implementation for the minimal number of cycles necessary and for which signals can be set to don't-cares. You do not have to worry about the busy signal. You may not need all the lines in the table for your solution.

You are allowed to introduce *at most* one new µBr target (Next State) for J (Jump) or Z (branch-if-Zero) other than FETCH0.

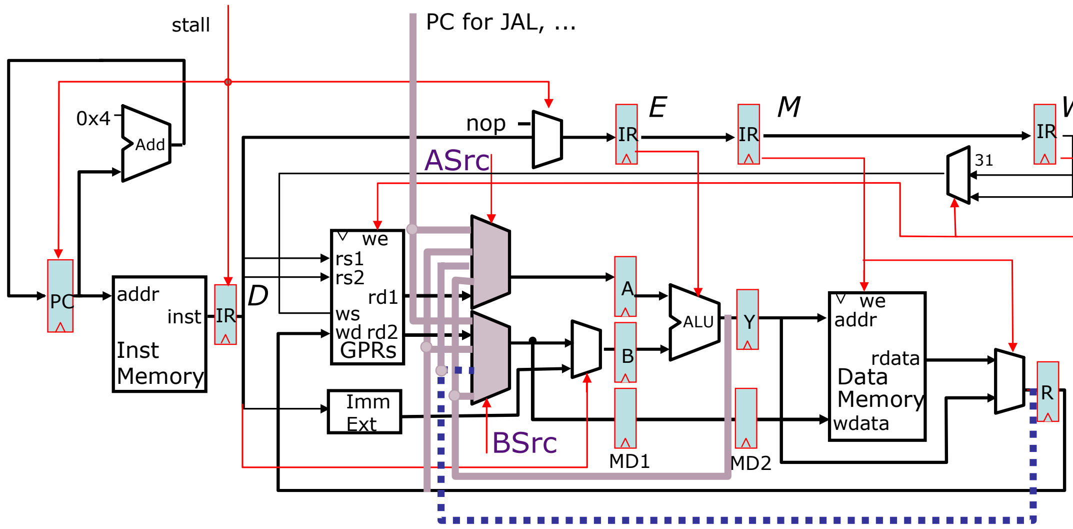
Name ______________________________

State	PseudoCode	Ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Im m	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
retz0																

Worksheet 1 for Question 4

Part C: Fully-Bypassed Simple 5-stage Pipeline (12 points)

In Lecture 6, we have introduced a fully bypassed 5-stage MIPS pipeline. We reproduce the pipeline diagram and the symbol definitions used in the stall and bypass conditions below.



Subscripts D, E, M, and W denote instruction decode, execute, memory, and write back stages, respectively.

ws = *Case* opcode
- ALU ⇒rd
- ALUi, LW ⇒rt
- JAL, JALR R31

we = Case opcode
- ALU, ALUi, LW ⇒(ws≠0)
- JAL, JALR ⇒on
- ... ⇒off

we-bypass$_E$ = Case opcode$_E$
- ALU, ALUi, ⇒(ws≠0)
- ... ⇒off

we-stall$_E$ = Case opcode$_E$
- LW ⇒(ws≠0)
- JAL, JALR ⇒on
- ... ⇒on

re1 = *Case* opcode
- ALU, ALUi, LW, SW, BZ
- JR, JALR ⇒on
- J, JAL ⇒off

re2 = *Case* opcode
- ALU, SW ⇒on
- … ⇒off

Question 5 (8 points)

In Lecture L6, we gave you an example of bypass signal (ASrc) from EX stage to ID stage. In the fully bypassed pipeline, however, the mux control signals become more complex, because we have more inputs to the muxes in ID stage.

Write down the bypass condition for the path between M (Memory) -> D (Decode) stages into register B. (The path is shown with a dotted line in the figure.)

Bypass $_{\textbf{MEM->ID(B)}}$ =

Question 6 (4 points)

Please write down an instruction sequence (with fewer than 5 instructions) which activates the bypass logic in Question 5.

Part D: Princeton Architecture (32 points)

Unlike Harvard-style (separate instruction and data memories) architectures, Princeton-style machines have a shared instruction and data memory. In order to reduce the memory cost, Ben Bitdiddle has proposed the following two-stage Princeton-style MIPS pipeline to replace a single-cycle Harvard-style pipeline in our lectures.

Every instruction takes exactly two cycles to execute (i.e. *instruction fetch* and *execute*), and there is no overlap between two sequential instructions; that is, fetching an instruction occurs in the cycle following the previous instruction's execution (no pipelining).

Assume that the new pipeline does **not** contain a branch delay slot. Also, don't worry about self-modifying code for now.

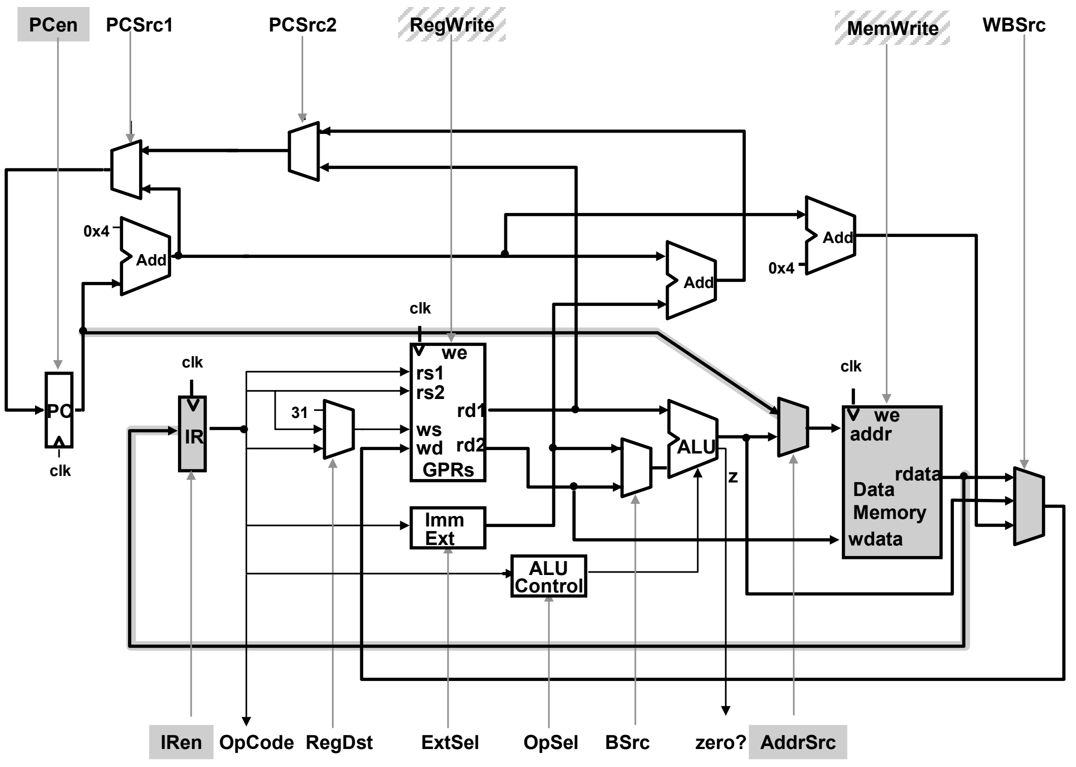


Figure D-1. Baseline Two-stage Princeton-style MIPS Pipeline

Name ____________________________

Question 7 (8 points)

Please complete the following control signals. You are allowed to use any internal signals (e.g. OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.).

Example syntax: PCEn = (OpCode == ALUOp) or ((ALU.zero?) and (not (PC == 17)))

You may also use the variable S which indicates the pipeline's operation phase at a given time.

```
        S := I-Fetch | Execute  (toggles every cycle)
```

PCEn =

IREn =

AddrSrc = Case ______________

_____________ => PC

_____________ => ALU

Name ____________________________

Question 8 (8 points)

After having implemented his proposed architecture, Ben has observed that a lot of datapath is not in use because only one phase (either I-Fetch or Execute) is active at any given time. So he has decided to fetch the next instruction during the Execute phase of the previous instruction.

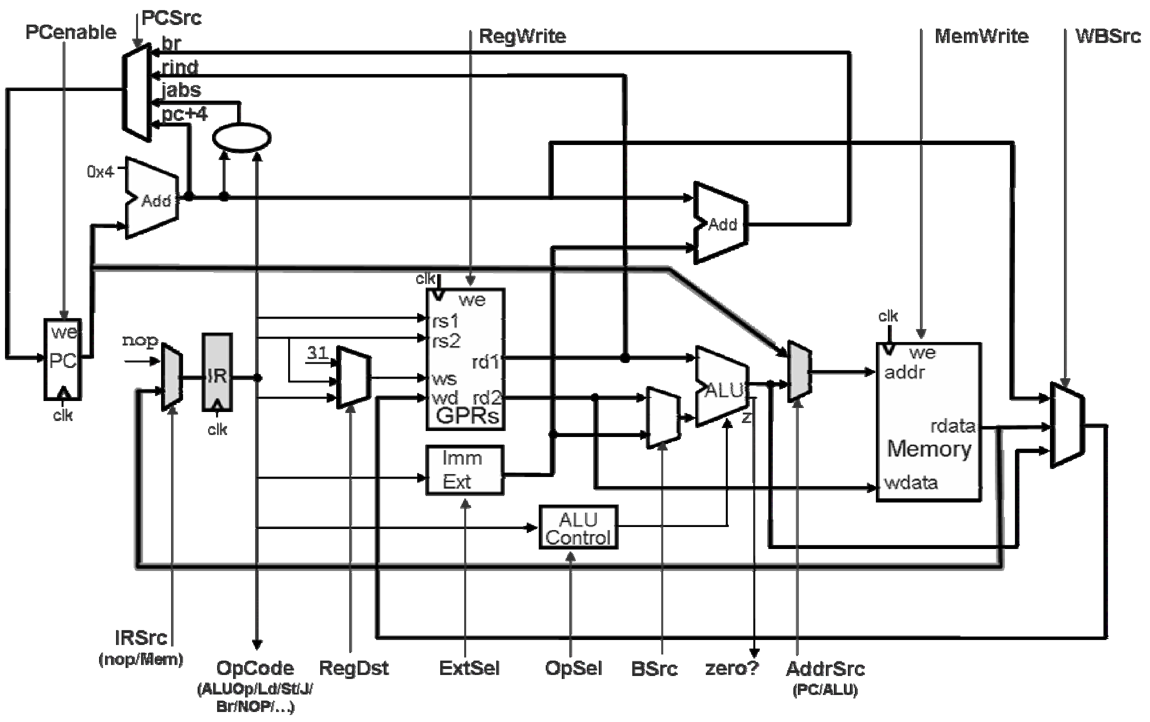


Figure D-2. Modified Two-stage Princeton-style MIPS Pipeline

Do we need to stall this pipeline? If so, for each cause (1) write down the cause in one sentence, and (2) give an example instruction sequence. If not, explain why. (Remember there is **no** delay slot.)

Name ____________________________

Question 9 (8 points)

Please complete the following control signals in the modified pipeline (Question Y). As before, you are allowed to use any internal signals (e.g. OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.)

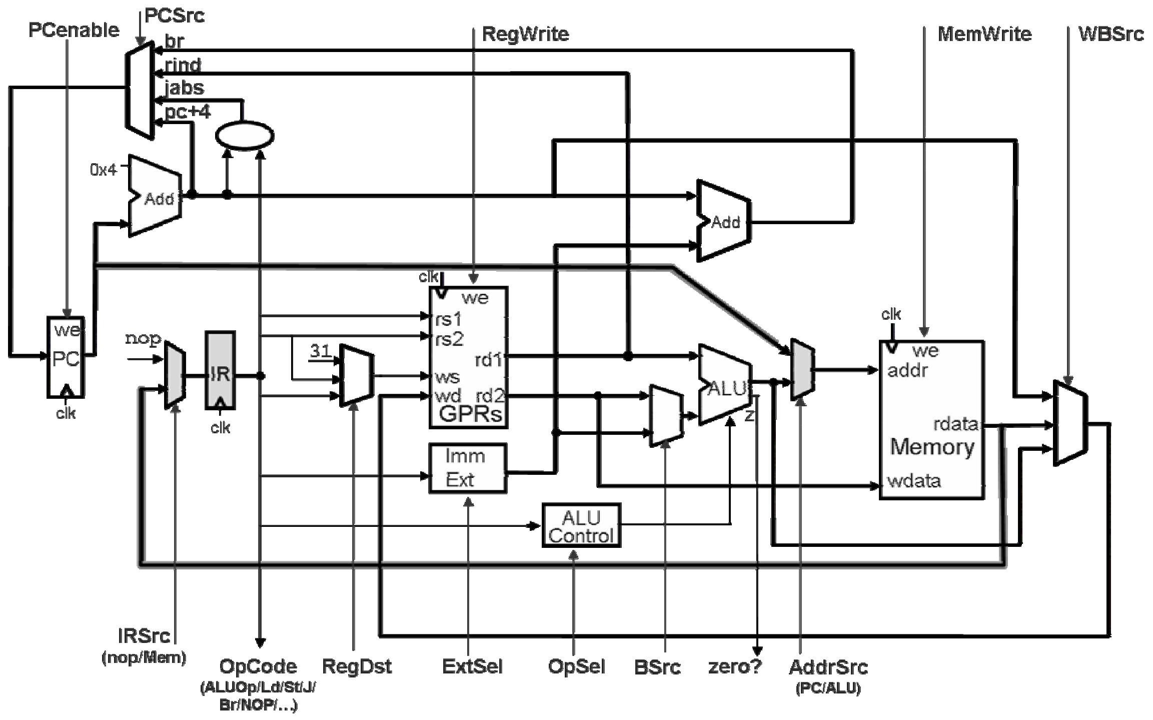
PCEnable =

AddrSrc = Case ______________
_____________ => PC
_____________ => ALU

IRSrc = Case ______________
_____________ => nop
_____________ => Mem

Question 10 (8 points)

Suppose we allow self-modifying code to execute, i.e. store instructions can write to the portion of memory that contains executable code. Does the two-stage Princeton pipeline need to be modified to support such self-modifying code? If so, please indicate how. You may use the diagram below to draw modifications to the datapath. If you think no modifications are required, explain why.



Name ____________________________

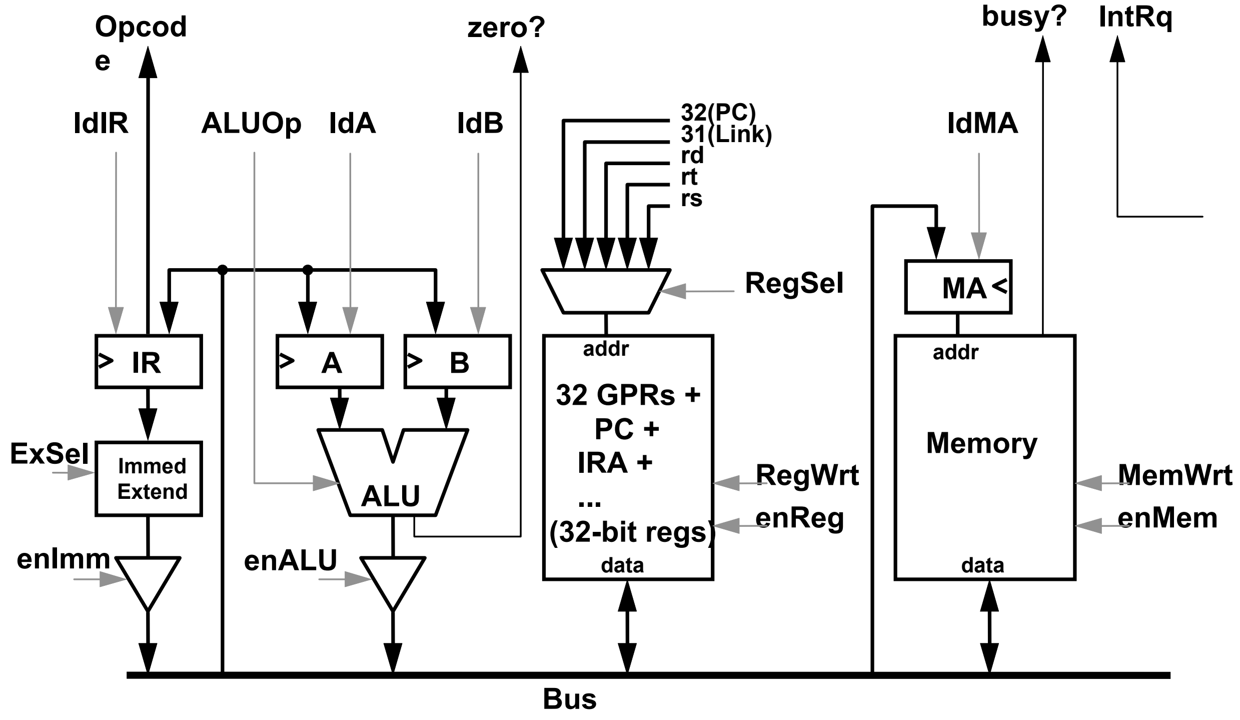
Appendix A. A Cheat Sheet for the Bus-based MIPS Implementation

Remember that you can use the following ALU operations:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B

Table H5-2: Available ALU operations

Also remember that µBr (*micro*branch) column in Table H5-3 represents a 2-bit field with four possible values: N, J, Z, and D. If µBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch). If it is Z (branch-if-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., it's a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). If µBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state.



Name ____________________________

Appendix B. 6.823 MIPS Instruction Table

Category	Instruction	Usage (Example)	Meaning	Encoding Format*
Arithmetic	add	`add Rd, Rs, Rt`	`Rd = Rs + Rt`	R-format
	subtract	`sub Rd, Rs, Rt`	`Rd = Rs - Rt`	R-format
	add immediate	`(addi Rt, Rs, 1)`	`(Rt = Rs + 1)`	I-format
	add unsigned	`addu Rd, Rs, Rt`	`Rd = Rs + Rt`	R-format
	subtract unsigned	`subu Rd, Rs, Rt`	`Rd = Rs - Rt`	R-format
	add immed unsigned	`(addiu Rt, Rs, 1)`	`(Rt = Rs + 1)`	I-format
Logical	and	`and Rd, Rs, Rt`	`Rd = Rs & Rt`	R-format
	or	`or Rd, Rs, Rt`	`Rd = Rs | Rt`	R=format
	and immed	`(andi Rt, Rs, 100)`	`(Rt = Rs | 100)`	I-format
	or immed	`(ori Rt, Rs, 100)`	`(Rt = Rs |100)`	I-format
	shift left logica**l	`(sll Rt, Rs, 10)`	`(rt = rs<<10)`	I-format
	shift right logical**	`(slr Rt, Rs, 10)`	`(rt = rs>>10)`	I-format
Data transfer	load word	`(lw Rt, 100(Rs))`	`Rt=Mem[Rs+100]`	I-format
	store word	`(sw Rt, 100(Rs))`	`Mem[Rs+100]=Rt`	I-format
	load upper immed	`lui Rt, 100`	Rt = 100*2^{16}	I-format
Conditional branch	branch on equal	`(beq Rs, Rt, 25)`	`if(Rs==Rt)goto PC+4+(25<<2)`	I-format
	branch on not equal	`(bne Rs, R5, 25)`	`if(Rs!=Rt)goto PC+4+(25<<2)`	I-format
	branch on zero	`(beqz Rs, 25)`	`if(Rs==0)goto PC+4+(25<<2)`	I-format
	branch on not zero	`(bnez Rs, 25)`	`if(Rs!=0)goto PC+4+(25<<2)`	I-format
	set on less than	`slt Rd, Rs, Rt`	`Rd=(Rs<Rt) ? 1:0`	R-format
	set less than immed	`(slti Rt, Rs, 100)`	`Rt=(Rt<100) ? 1:0`	I-format
	set less than unsigned	`sltu Rd, Rs, Rt`	`Rd=(Rs<Rt) ? 1:0`	R-format
	set less than immed unsigned	`(sltiu Rt, Rs, 100)`	`Rt=(Rt<100) ? 1:0`	I-format
Uncond. jump	jump	`(j 2500)`	`goto (2500<<2)`	J-format
	jump register	`jr Rs`	`goto Rs`	R-format
	jump and link	`(jal 2500)`	`R31=PC+4; goto (2500<<2)`	J-format

* See the table below.
** Slightly different from the original MIPS encoding

MIPS instruction encoding formats

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS inst = 32 bits
R-format	opcode	rs	rt	rd	unused	funct	Arithmetic, logical
I-format	opcode	rs	rt	addr/immed value			Transfer, branch, imm
J-format	opcode	target address					`j, jal`