14 | Data Path Storage Components, Design, and Control Unit (Chapter 7)

The **MIPS datapath** are components and connections needed to implement the MIPS instruction set. This allows for:

- Arithmetic and logical instructions (ex. Add/subtract, and/xor)
- Register and immediate formats (we're not going to do too much with this)
- Memory reference instructions (immediate format instructions)
- Transfer of control instructions: branch and jump

The datapath includes a **control unit**:

- Input from the instruction
- Generates control signals from the components in the datapath
- Always knows which instruction is being executed

Storage Components

Storage components in the MIPS datapath include **register file**, **data memory**, and **instruction memory**. The following are what they have in common:

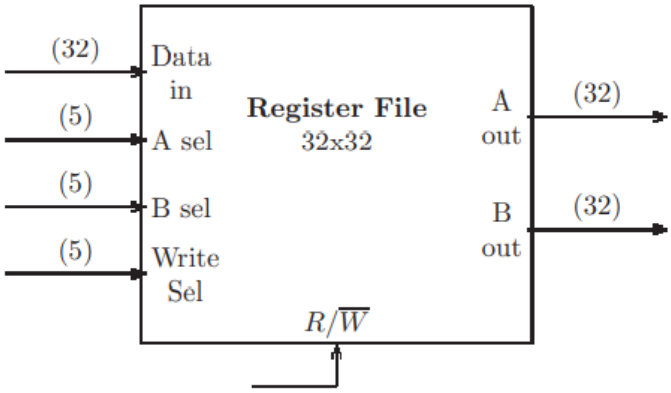
- Consist of 32-bit words
- Any word may be selected, via an 'address' (ex. 32 registers, a register address, a 5 bit number, to select one of those registers)
- Control signals can enable read/write (ex. instructions may want to change registers)
    - These determine whether a storage component is being changed
    - If changed = write, if not = read
- At least one output bus and at least one input bus

The **register file** is where all the 32 general registers are (each register is 32-bits). Thus, we call this the 32 by 32 file.

- Any one of these registers can be selected with a 5 bit number (0 - 31)
- There are two output buses (each 32-bits, A and B) and one input bus (32-bits, data in)

- The select buses are each 5 bits (each select one of the registers)
    - A select: will select a register to put on the A output
    - B select: will select a register to put on the B output
    - Write select: will select a register to be written from the data-in bus

- How do we know whether a register is going to be written? *One R/W control signal*
    - 0 = write (represented by a horizontal bar above); 1 = read (no bar)
    - If you want to change something, have a 0 come in; otherwise, have a 1
    - Even if you have a 0 coming in, you'll still have output in the A and B buses

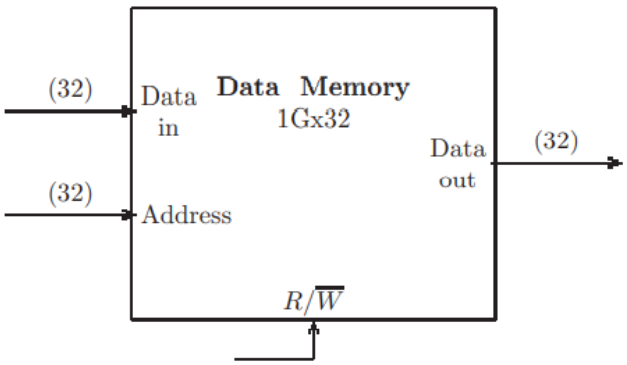


Data memory consists of 1G (meaning 2^{30}) 32-bit words. Each word is 4 bits, so in actuality, this is 4G bytes, or 1G words.

- One output bus: data out; one input bus: data in

- One address bus (32 bits): byte addressable
    - Can select any word of memory to be put on the output bus or to be changed from the input bus
    - Selects a data word

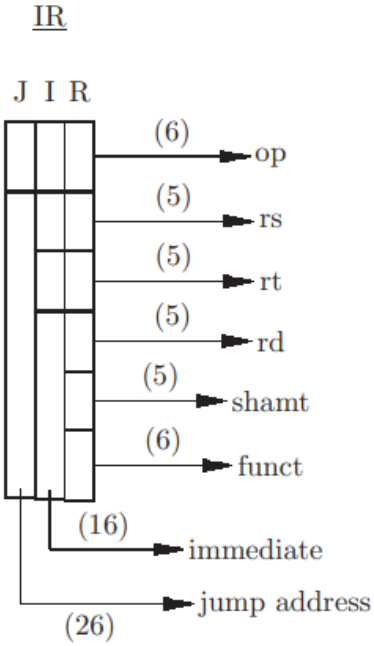
- One R/W control signal (same meanings as before for binary)
- Always going to be storing/retrieving at least one full word



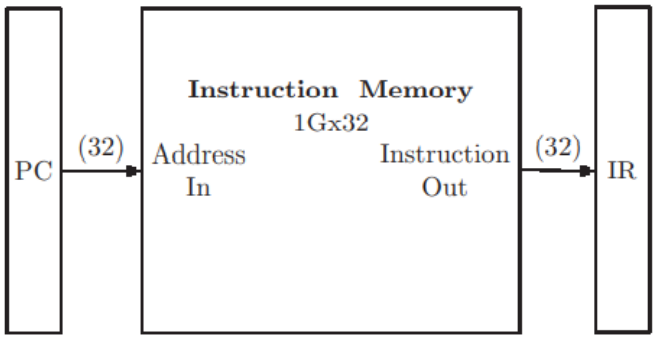
Instruction memory is a component, along with PC and IR registers.

- **PC register**: Program counter
  - Stores the address of the next instruction to be executed
  - The instructions in our program are all 32-bit words, and they're stored here
  - It's all 1G by 32 bit words again

- **IR register:** Instruction register
  - Stores the instruction currently being executed
  - Loads from instruction memory (address in PC register)
  - Depending on the format of the instruction being executed (like R or I), we imagine bits of the IR being grouped into fields corresponding to instruction fields
  - Output busses split (R = 6 busses, I = 4 busses, J = 2 busses), see how it's split up right:
  - For I output busses, the 16 bits are separated into one output bus together; the control unit will know which is being executed and determine which bus value to be used (it's one 32-bit word with outputs being split off)



- **Instruction memory:**
  - Separate from data memory; unlike MARS, this datapath is separated into separate memory chips
  - Consists of 1G by 32 bit words
  - One output bus: *Instruction Out*, to IR
  - One address bus (32-bits): From PC (selects a data word)
  - IR is loaded when the PC register changes

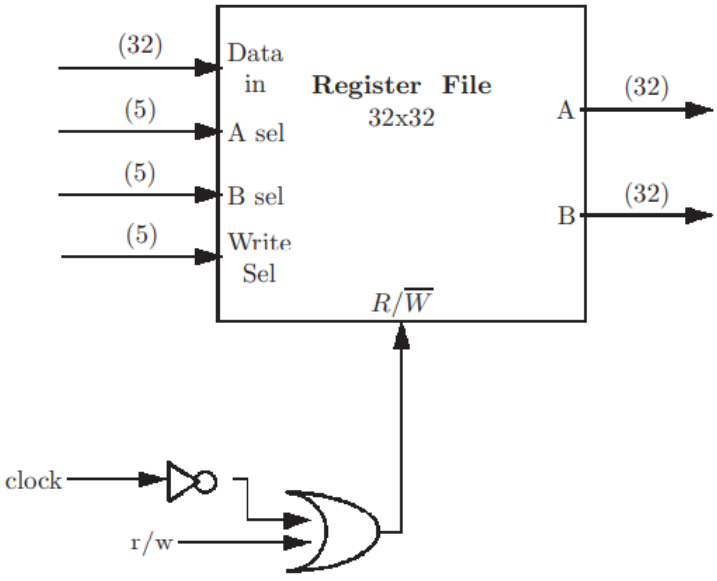


Design of the Datapath

Let's connect the components described above so as to execute MIPS instructions. A '**clock signal**' is used to synchronize the effects of all components in the data path. (just 1 bit that changes repeatedly from 0 to 1 as time goes on, shown above).

Here is a register file:

- The clock signal will be fed into an inverter as one input into an OR gate
- The other input to the OR gate is the r/w signal from the control unit
- The output of the OR gate forms the R/W signal to the register file itself
- Notice whenever clock signal is 1, we'll have a 1 as R/W (which will NOT write to the register file)



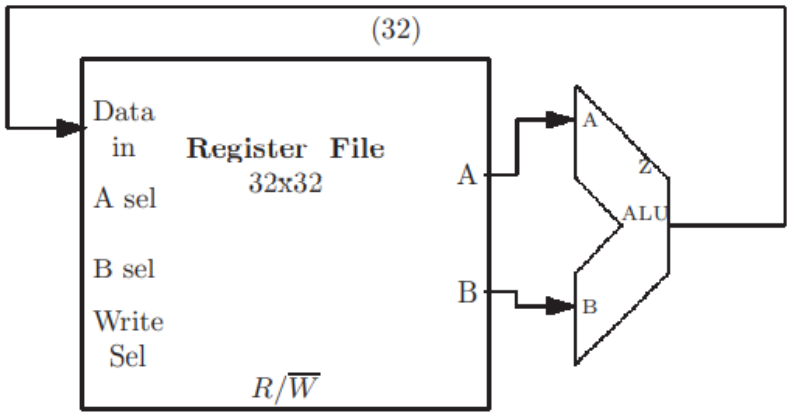
In our design, we'll have a '**single cycle datapath**': one instruction is executed on each clock cycle. If the clock goes up and down 10 times, we will have 5 cycles, thus 5 instructions.

- Clock speed is measured in Hertz (cycles per second), which tells us the speed

- 1 MHz = 10^6 cycles per second
- 1 GHz = 10^9 cycles per second

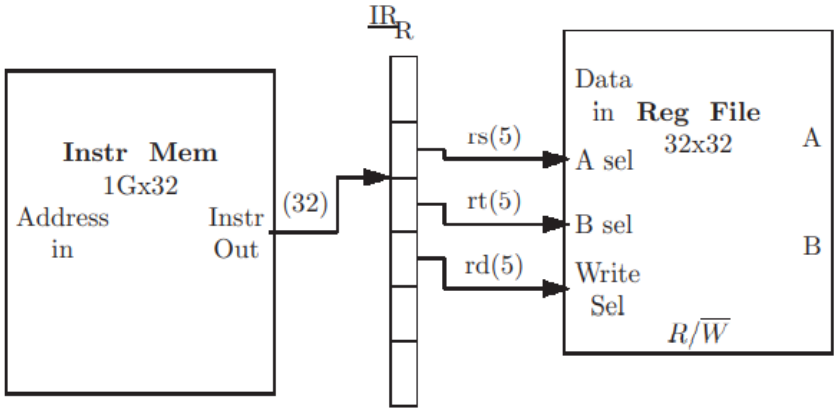
Connecting the Register File and the ALU

- ALU has 2 inputs (A and B, both 32-bits) and a 32-bit output and Z-output
- Result of an arithmetic or logical instruction can go back into the register file
- A select and B select from the Instruction Register
- Write select comes from the Instruction Register
- R/W signal comes from the control unit



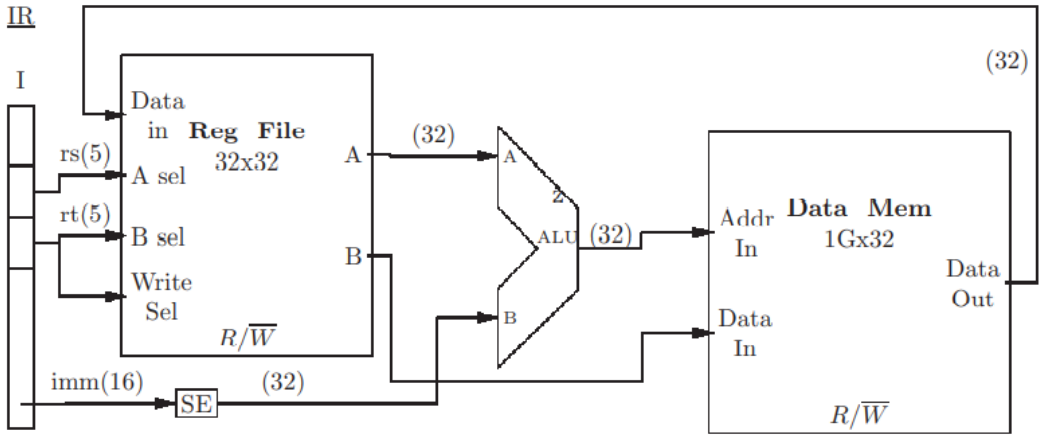
Connecting the Instruction Memory, Instruction Register, and Register File for an R format instruction

- The instruction being executed is output from the instruction memory, loaded into the IR
- The fields rs goes into A select, rt goes into B select, rd goes into Write select
- Address in to Instruction Memory is from PC (not shown)



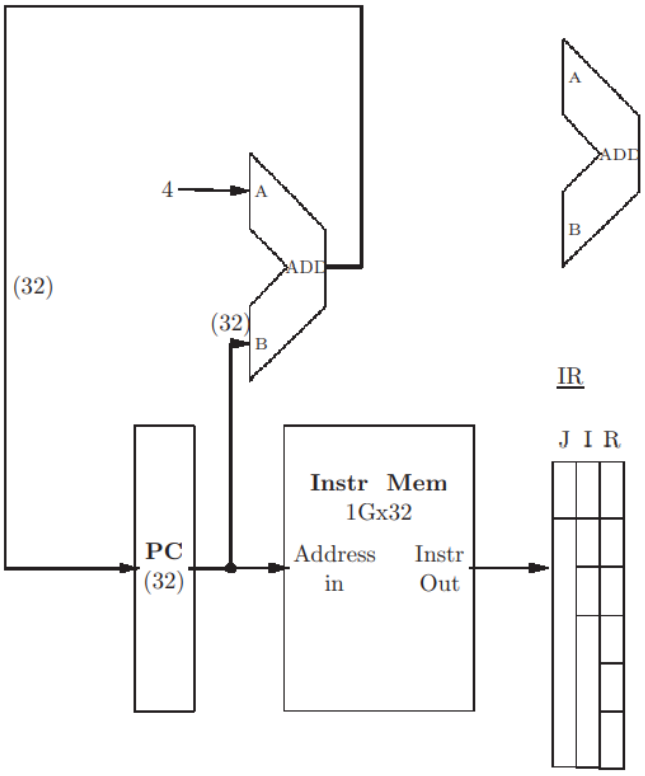
Connecting the Data Memory, Instruction Register, and Register File for load/store instructions

- rs field of the instruction (from IR) is the A select for the register file
    - Selects A input to ALU
- rt field of the instruction is the B select for the register file
    - AND the Write select (for a load word instruction)
- Immediate field is extended to 32 bits with sign extend (SE)
    - Forms B input to ALU
- ALU adds the rs register and the immediate value to form a memory address
- **B output of register file is data in to the data memory (for store word instructions)**



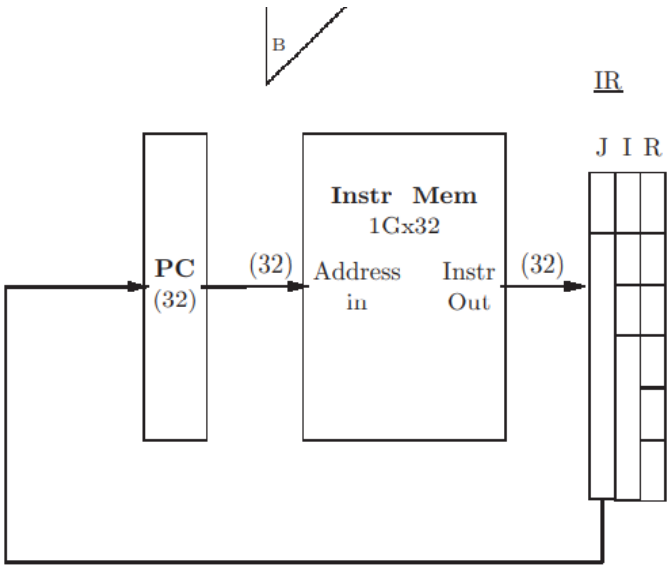
Connecting the PC, Instruction Memory, and instruction Register for sequential execution (no branching)

- A dedicated 32-bit adder is used to increment the PC by 4 (the second adder not needed)
    - We take the value in the PC, branch it off and feed it into the adder
    - The other input to the adder is constant value 4 (32-bit value “4”)
    - Add 4 + whatever’s in the PC, and that result goes back into the PC
    - So in each instruction, the value is increased by 4 each time an instruction is executed
    - When the PC changes, the instruction register changes (comes out of the IM)



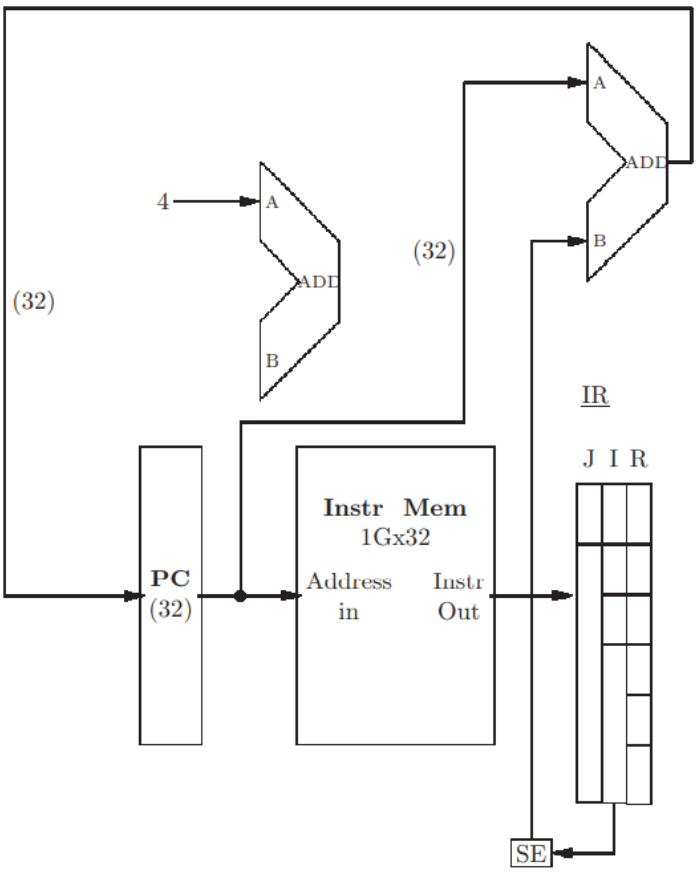
Connecting the PC, Instruction Memory, and instruction Register for (unconditional) jump instruction

- PC is loaded from the jump address in the instruction
- Padded with 0's to form a full 32-bit word
- Neither of the adders are needed in this case (we're jumping to an absolute address)



Connecting the PC, Instruction Memory, and instruction Register for (conditional) branch instruction

- A conditional branch is a *relative branch*
    - Takes what's in the PC and adds or subtracts some 16-bit value to it to produce a new value in the PC, which is the address of the next instruction to be executed
- Take what's in the PC as one piece of the adder; the other part comes from the immediate field (fed through a sign extend, SE, to allow us to have negative values and extended to 32-bits)
- The result of that addition is fed back to the PC
- (*Comparison handled later*)



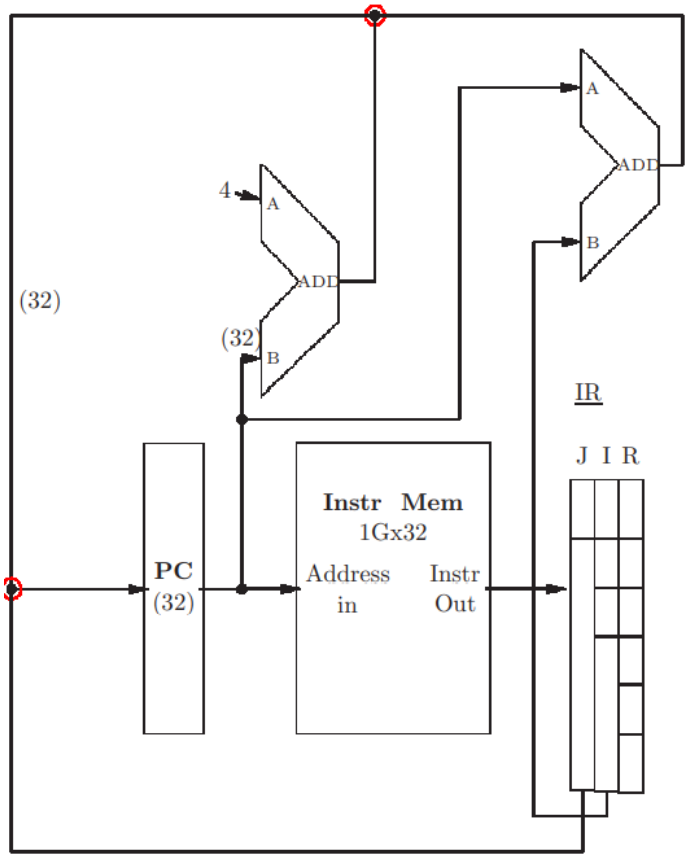
When combining these diagrams to process:

- Sequence (no branch or jump)
- (Conditional) branch (relative branch)
- (Unconditional) jump (absolute branch)

We get some contradictions, designated with red circles.

- We've combined the past 2-3 diagrams for this
  - For a sequential execution, we want to add 4 to the PC and put it back
  - But for relative branch, want to add the two values and then put it back in the PC
  - We can't do both

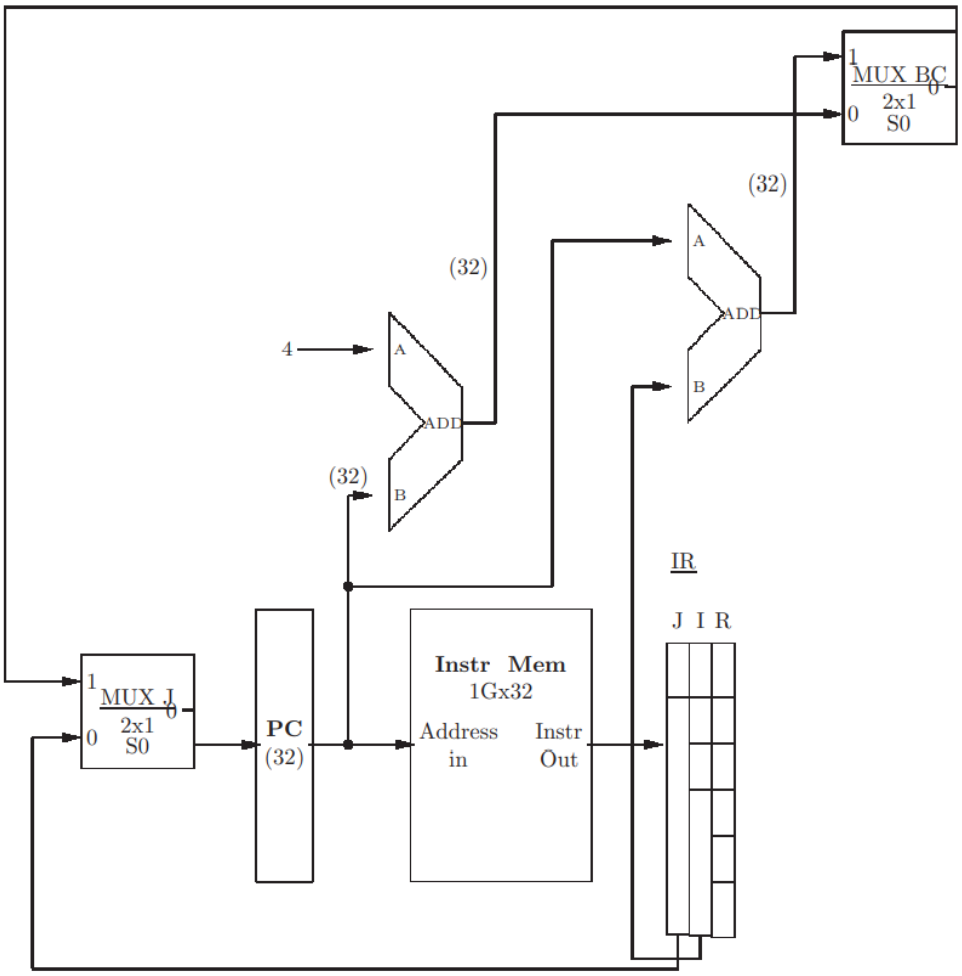
- For absolute address branching, we have a value coming out of IR
  - And another coming in from the adders as well, which creates the contradiction
  - (the one to the left side)
  - Contradictions can be solved with multiplexers (for both, insert 2x1 32-bit MUX)
  - Control signal for the muxes are produced by a control unit, using the fields of IR



Multiplexers have been used to resolve the two contradictions here:

- If we have a 0 coming in the upper right MUX, it's sequential
  - If it's 1, then it's a branch to a relative address
  - Selects output of one of the adders to distinguish sequence and relative branch

- Two signals coming into the PC calls for a MUX
    - If it's 0 coming in the bottom arrow, it means load the PC from the address field of the IR
    - If it's 1, load the PC from the output of the MUX on the top
    - Selects output of the other MUX, or address field from a Jump instruction, as input into the PC



The Control Unit

The **control unit** sends control signals to the ALU, R/W inputs, and MUXes. It's implemented for a subset of the MIPS instruction set (see right):

- R format instructions:
    - and, or, add, sub, nor
- Memory reference instructions (I format)
    - lw, sw
- Transfer of control instructions:
    - beq, bne (I format)
    - j (J format)

How do we determine whether a (conditional) branch takes place?

- Control Unit directs the ALU to subtract (the Op Select)

- The two registers being compared are subtracted
  - If we get a 0 result, the two registers being compared are equal
  - If we get a nonzero result, the two registers being compared are not equal

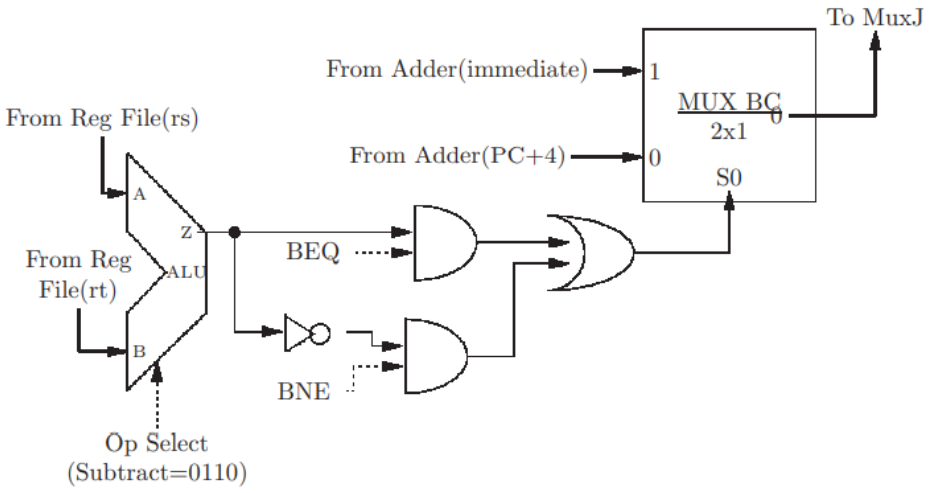
- So, we'll look at the Z-output, and it's either going to be 9 or 1
  - Z = 0 AND instruction is beq, we want to do a relative branch
    - beq's op code is 0x04 = 00 0100
    - Boolean expression BEQ = $op'_5\ op'_4\ op'_3\ op_2\ op'_1\ op'_0$

  - Z = not 0 AND instruction is bne, we want to do a conditional branch
    - bne's opcode is 0x05 = 00 0101
    - Boolean expression BNE = $op'_5\ op'_4\ op'_3\ op_2\ op'_1\ op_0$

  - In either case (connected with an OR gate), we select the 1 input into the MUX

- From Adder (immediate) is the output (if we get 1) from the adder that adds the immediate field to the PC (program counter) (and thus a relative branch)

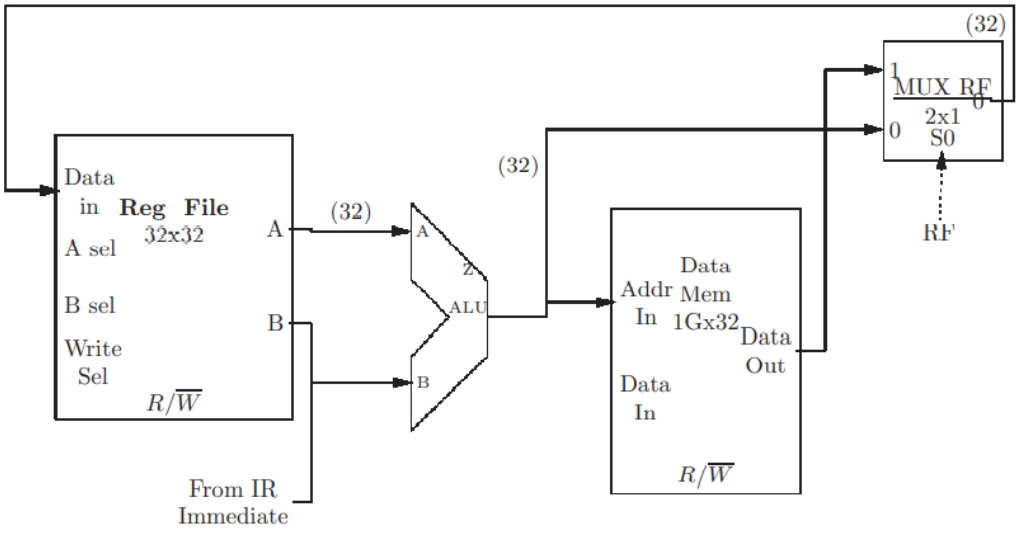
- If we get a 0 coming in, we add 4 to the PC and fall through to the next instruction (and thus no branch)



Contradiction at Reg File Data In

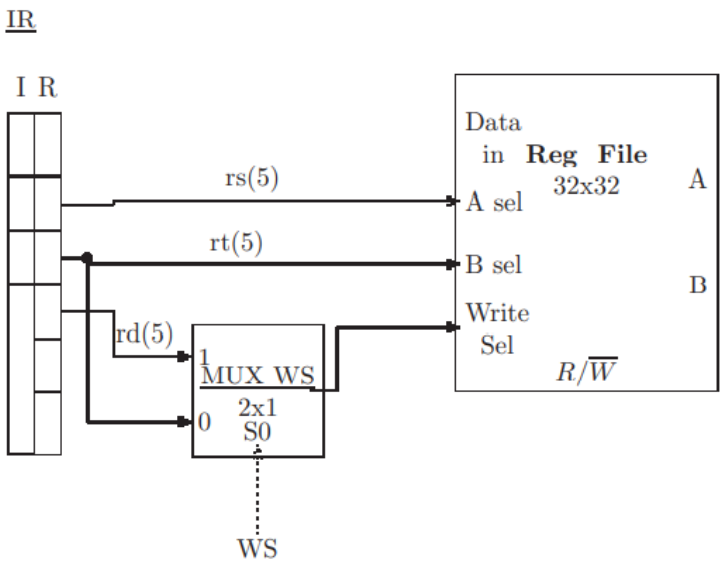
Where is the data coming from? Well, it could come from an R format instruction (like add, the result of the add coming out of the ALU going into the RF) or from data memory out (if it was a load word instruction). These are the two possibilities; we'll feed them into the multiplexer, and the control unit will decide which one of those inputs goes back into the register file.

- The control unit will send a 0 for R format and 1 for load word instruction (data from the data memory needs to go back into the register file).
- The contradiction is now solved with a MUX
- MUX control signal is generated by the control unit



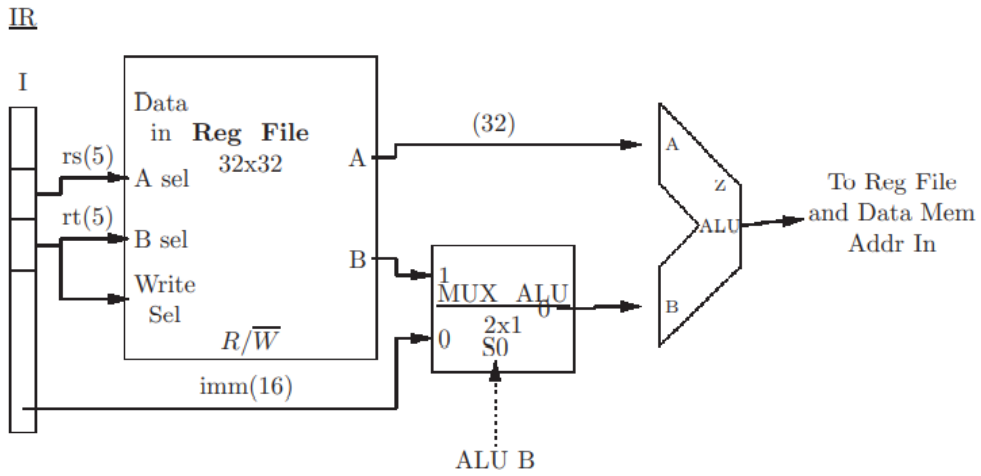
Contradiction at Regular File Write Select

If we have an ordinary R format instruction, then the $rd field of instruction determines which register is being changed for the result of an operation. But suppose we have an immediate instruction (not in a subset) or a load word instruction? Then, the register being changed is determined by the $rt field (shown by the dot on the rt(5) line). Well, the control unit will choose one of the two, and it's resolved with a MUX generated by the control unit.

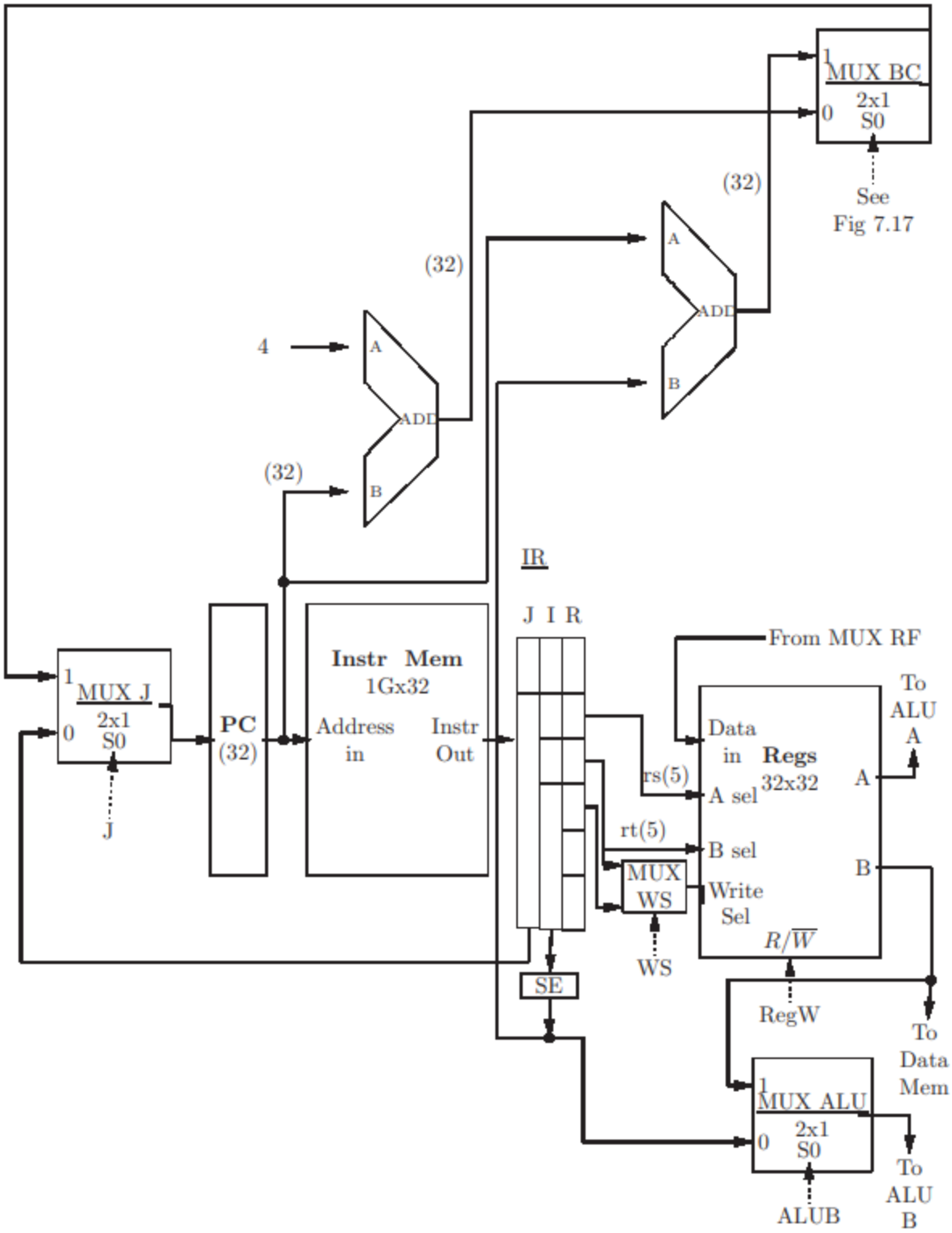


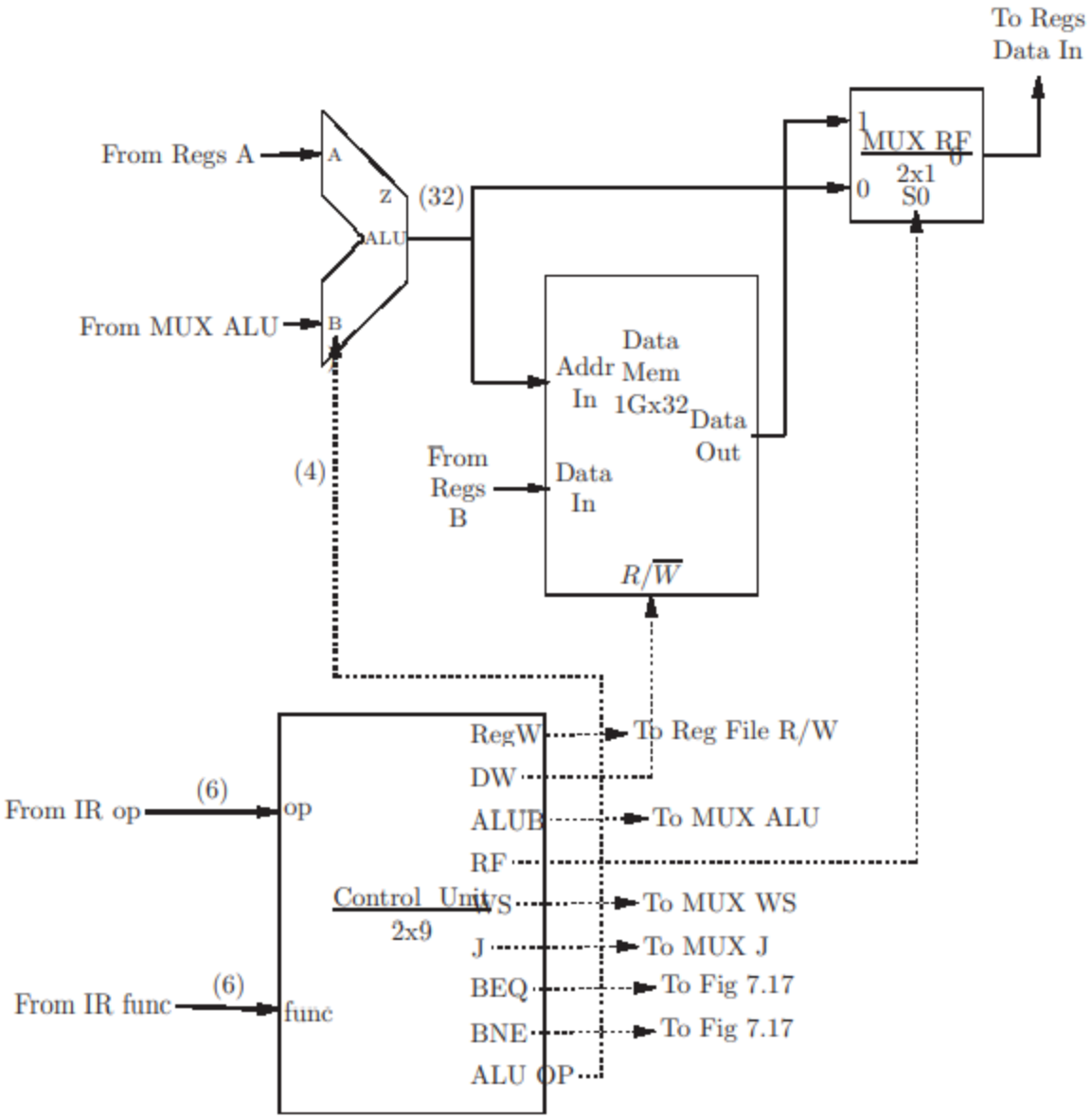
Contradiction at ALU B input

It should be taken from register file B output for R format instructions, but taken from immediate field of instruction for I format and memory reference instructions. The control unit will generate a MUX control signal. The bottom-most arrow should be going through an SE (sign exchange) unit.



Putting it all together: These are very large pictures of the datapath for the MIPS architecture, and it all is just the same things we've been working through (first shows everything except the ALU or the data memory):


1
MUX BC
2x1
0 S0
See
Fig 7.17
(32)
A
ADD
B
(32)
4
A
ADD
B
(32)
IR
J I R
From MUX RF
1
MUX J
2x1
0 S0
J
PC
(32)
Instr Mem
1Gx32
Address
in
Instr
Out
Data
in
Regs
32x32
A sel
B sel
Write
Sel
R/W
rs(5)
rt(5)
MUX
WS
WS
A
B
To
ALU
A
SE
RegW
To
Data
Mem
1
MUX ALU
2x1
0 S0
ALUB
To
ALU
B



See the outputs in the control unit.

- **RegW** determines whether we're reading or writing for the register file
  - 0 = write, 1 = don't write

- **DW** means "data write", goes to data memory (if 0, writing it to data memory; if 1, not)
  - 0 = write, 1 = don't write
  - Data memory is written by a store word (sw) instruction; all other instructions should NOT be written into data memory, so unless the opcode is 0x2b, then it's always 1

- **ALUB** signal comes from the register file or the immediate field of the instruction; control signal determines through the ALUB signal, which is chosen
    - 0 = selected from B output of register file
    - 1 = selected from immediate field of instruction

- **RF** goes to MUX RF, choosing between data memory and the ALU output (determines which goes back to the register file)
    - 0 = selected from ALU output
    - 1 = selected from data memory out

- **WS** if it's an ordinary R format instruction, it'll come from $rd, but if it's a load word instruction, it'll come from $rt. It determines which register is being written from Data In
    - 0 = selected from $rt field of instruction
    - 1 = selected from $rd field of instruction
    - (I guess this is more like lw is only one that has 1 for this)

- **J**; if it is a 0, then PC is loaded from the address field in the instruction register; if it's a 1, the PC is loaded from the output of the multiplexer (MUX BC) which would mean we would either have a relative branch or fall through to the next instruction
    - 0 = selected from instruction bits 0...25 (jump instruction)
    - 1 = selected from output of MUXBC (not jump instruction)
    - (Also, everything is 1 except for j, it seems; I assume it's true for jal too)

- **BEQ** and **BNE** (shown in first picture); if BEQ is true, BEQ is 1 and the same is for BNE

- **ALU OP** is 4 bits, determining the ALU operation (looked in Chapter 6, determines which operation the ALU should be doing like arithmetic or logical operations)

And how about the inputs for the control unit?

They come from the instruction register. There was an opcode (6 bits) and a function field, for R format instructions (also 6 bits). These are inputs into the instruction register. Looking at the instruction unit, the high order bits are the opcode; the low order bits are from the function code if it is a R format instruction) at the IR in the first picture. This is how the control unit knows what instruction is being executed, using that info to generate the control signals.

- Opcode bits 0...5
- Function code bits 26...31

How do we get the control unit outputs? Let's take the add instruction as an example.

- RegW: You want to write to the register file (store output?) (**0**)
- DW: Do you want to write or change data memory with an add instruction? No (**1**)
- ALUB: Let's look at MUX ALU. The B input to the ALU should come from the B output of the register file (not from the immediate field of the instruction). (**1**)
- RF: What's coming back into the register file? Should it come from ALU or data memory? Since we're doing an add instruction, it should come from ALU (**0**)
- WS: Let's look at MUX WS. Do we want to select from the $rd or $rt field? We want to choose the register being written with the result. Since it's the add instruction... (**0**)
- J: Is for a jump (if it's a 0, it'll take the address from the instruction register, but if it's a 1, it'll take the output of the MUX BC, which would take the output of the adder and send it down to the MUX J) (**1**)
- BEQ and BNE: Is add instruction BEQ or BNE? No. So both (**0**).
- ALU OP: The ALU OP for addition (from chapter 6) is **0010**

Let's try beq.

- Do we want to change one of the registers? No.
- Do we want to write into data memory? No.
- What do we want coming into the B input of the ALU? We want it coming from the B output of the register file
- We don't want to change the register file, so it doesn't matter which is chosen. It can choose the output from data memory or from the ALU.
- This determines which register is selected for being written. Again, we're not changing the register, so it doesn't matter what comes in here.
- This isn't a jump instruction so we don't want the jump bus.
- Is this a branch on equal instruction? Yes. Is this a BNE? No.
- We want the ALU to do a subtraction (looking back on chapter 6)

Instruction	RegW	DW	ALUB	RF	WS	J	BEQ	BNE	ALU OP
and	0	1	1	0	0	1	0	0	0000
or	0	1	1	0	0	1	0	0	0001
add	0	1	1	0	0	1	0	0	0010
sub	0	1	1	0	0	1	0	0	0110
nor	0	1	1	0	0	1	0	0	1100
slt	0	1	1		0	1	0	0	0111
lw	0	1	0	1	1	1	0	0	0010
sw	1	0	0	?	?	1	0	0	0010
beq	1	1	1	?	?	1	1	0	0110
bne	1	1	1	?	?	1	0	1	0110
j	1	1	?	?	?	0	?	?	????

Figure 7.22: Table showing the outputs of the control unit for each instruction. Don't cares are indicated by question marks.

Example: Write an expression for each output as a function of the 12 input bits. Let's do RegW.

When is this output going to be a 1? It's going to be a 1 if we have sw, beq, bne, and j. For all other instructions, RegW is 0. Thus, here are the opcodes for each one:

sw: opcode = 10 1011
beq: opcode = 00 0100
bne: opcode = 00 0101
j: opcode = 00 0010

RegW = $op_5\ op_4'\ op_3\ op_2'\ op_1\ op_0$ + **[sw]**
$op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0'$ + **[beq]**
$op_5'\ op_4'\ op_3'\ op_2\ op_1'\ op_0$ + **[bne]**
$op_5'\ op_4'\ op_3'\ op_2'\ op_1\ op_0$ **[j]**