Register File

- All eight registers tied to same input
- Outputs tied to two multiplexors, since most binary operations need two operands
  - a output
  - b output
- Two registers that will be forwarded to the datapath
- Each multiplexor set individually (same register’s output can be on a and b concurrently)
- Multiple registers can write the same result (individual $r_{<j>}$'s must be 1)

Controlling Register File

- Fourteen wires
  - $a_{sel}$ – three wires to select 1 register for a output
  - $b_{sel}$ – three wires to select 1 register for b output
  - $r_{<i>}$ – to select register when to store value from result bus

ALU

- Logic operations
  - NOT – inverts each bit in the value on the a bus
  - AND – bitwise AND of values on a and b bus
  - OR – bitwise OR of values on a and b
  - XOR – bitwise XOR of values on a and b
- Arithmetic operations
  - Addition
  - Subtraction
  - Both on 2’s complement numbers

ALU functions

- ADD
  $a + b + c_{in}$
- SUB
  $-b = b + 1$
  $a + b + c_{in}$
- ADDA
  $a + c_{in}$
- SUBA
  $a - 1 + c_{in}$

Summary of Current Datapath

<table>
<thead>
<tr>
<th>Signal</th>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{sel}$</td>
<td>3</td>
<td>Select register for a bus</td>
</tr>
<tr>
<td>$b_{sel}$</td>
<td>3</td>
<td>Select register for b bus</td>
</tr>
<tr>
<td>$r_{&lt;i&gt;}$</td>
<td>8</td>
<td>Select registers to be written</td>
</tr>
<tr>
<td>$alu_{sel}$</td>
<td>3</td>
<td>Select ALU function</td>
</tr>
<tr>
<td>$c_{in}$</td>
<td>1</td>
<td>Carry in to low-order bit of ALU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signal</th>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_{of}</td>
<td>1</td>
<td>Carry out of high-order bit</td>
</tr>
<tr>
<td>m_{7}</td>
<td>1</td>
<td>Sign bit of the ALU output</td>
</tr>
</tbody>
</table>

What Happens in Current Datapath?

- A and b outputs contain the values in the registers specified by $a_{sel}$ and $b_{sel}$
- ALU performs the operation specified by $alu_{sel}$ using
  - $a$ and $b$ as operands
  - Externally supplied $c_{in}$
- ALU produces outputs $m$ and $c_{of}$
- Value $m$ is selectively loaded into zero or more result registers – controlled by $r_{<i>}$
Control Statement
• control program made up of control statements
• lists the control signals to be asserted during a given cycle separated by commas
• multiple wire controls are written in terms of their component wires
  - \( \text{alu}_{\text{sel}} \), \( \text{alu}_{\text{sel}2} \), \( \text{alu}_{\text{sel}1} \), and \( \text{alu}_{\text{sel}0} \)

Example
• \( r0\text{write}, r5\text{write}, a\text{sel0}, b\text{sel1}, \text{alu}_{\text{sel}2} \) means
  - register \( r1 \) is placed on \( a \) (\( a\text{sel} = 001 \))
  - register \( r2 \) is placed on \( b \) (\( b\text{sel} = 010 \))
  - sum of \( a \) and \( b \) is computed by ALU (\( \text{alu}_{\text{sel}} = 100 \))
  - result is stored in \( r0 \) and \( r5 \) (\( r0\text{write} \) and \( r5\text{write} \) are 1)

Branching Basics
• cond is a single bit
• cond is typically a status wire
• could also be an internally generated signal
• branching is independent of control line settings
• branching is typically written at the end of the line, after all control lines

Circular Shifting
// Circular shifting left by 1 bit
// works by testing the high order bit
// if 1 then shift1 shifts a 1 into the low order bit
// else shift0 shifts a 0 into the low order bit
// \( a_{\text{w}r}=c_0 \), \( b_{\text{w}r}=c_0 \), \( \text{alu}_{\text{w}r} \text{AND} \), if \( m_7 \) then goto shift1 else goto shift0 endif;
shift1: \( r<i>\text{write}, a_{\text{w}r}=<i>, b_{\text{w}r}=<i>, \text{alu}_{\text{w}r}=\text{ADD}, c_0=1, \) goto next;
shift0: \( r<i>\text{write}, a_{\text{w}r}=<i>, b_{\text{w}r}=<i>, \text{alu}_{\text{w}r} \text{ADD}, c_0=0; \) next;

Right Shift
• considerably harder, why?
• algorithm will shift a reg right \( k \)-bits
  - circularly shift the register \( n-k \) bits (n number of bits in register)
  - set the \( k \) high order bits to 0
  - setting to 0 is done by creating a constant whose \( n-k \) low order bits are 1, \( k \) high order bits are 0,
    then AND it with the register we are shifting

BNF for Control Programs
```
cntl_program = [{cntl_stmt}]
cntl_stmt = [{cntl_label}][{cntl_signals}][{cntl_branch}]
cntl_label = name: | name[const]
cntl_signals = [{cntl_signal}]
cntl_signal = name: | name = CONST
ctl_branch = goto name: | if cond then goto name endif |
```


**Explanation of BNF**

- names to be expanded are in italics
- names which are to be as written are as bold
- information between {} is optional
- objects with + occur one or more times
- objects with * occur zero or more times
- objects separated by | indicated choice

**Canonical Form BNF**

- only signal names separated by commas are allowed
- also requires a branch to appear on each line – only way a statement can be reached is through branching
- therefore, every line also requires a label

**Control Signals**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLD</td>
<td>0</td>
<td>contents of MAR held constant</td>
</tr>
<tr>
<td>LOAD</td>
<td>1</td>
<td>MAR loaded from output of ALU MAR_alu</td>
</tr>
<tr>
<td>HOLD</td>
<td>0</td>
<td>MDR value can be read by both the result mux and memory</td>
</tr>
<tr>
<td>LOAD_ALU</td>
<td>1</td>
<td>load MDR from ALU output</td>
</tr>
<tr>
<td>LOAD_MEM</td>
<td>2</td>
<td>load MDR from memory</td>
</tr>
</tbody>
</table>

**Special Purpose Registers**

- MAR and MDR are special purpose registers
- they are not directly controllable by the machine language programmer
- computers have many such registers available
- only available for use by the control unit

**Control Wires**

- resultsel – selects either ALU or MDR to be loaded into the register file
- read – specifies a read operation
- write – specifies a write operation
- both of these go directly to the memory and are not used by the datapath or the memory interface
Usage

- if both MDRsel and MARsel are zero we hold
- thus, if these control lines are not mentioned in the control program, their contents do not change
- this is important for the upward compatibility of the design

Load

- load r<i>, r<j>
  - performs r<i> <- MEM[r<j>]
  - load MAR with r<j>
  - read memory for as many cycles as necessary
  - move value from MDR to r<i>
  - alu sel = AND, mar sel = LOAD;
  - memread : read, mdr sel = LOAD_MEM,
  - if wait then goto memread endif;
  - result sel = MDR, r<i> write;

Store

- store r<j>, r<k>
  - performs MEM[r<j>] <- r<k>
  - load MAR with r<j>, MDR with r<k>
  - write to memory until memory completes
  - alu sel = AND, mar sel = LOAD_ALU;
  - memwrite: write, if wait then goto memwrite endif;

Temporaries

- in some cases temporary registers are needed
  - hold intermediate values
  - hold constants
  - need more registers than are available
- registers r4 through r6 should be used as temporaries (in hwk, on exams, etc.)
- this will also allow current control programs to function in the future

Control Signals – Summary

<table>
<thead>
<tr>
<th>Signal</th>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_sel</td>
<td>3</td>
<td>select register for a bus</td>
</tr>
<tr>
<td>b_sel</td>
<td>3</td>
<td>select register for b bus</td>
</tr>
<tr>
<td>r_sel</td>
<td>8</td>
<td>select registers to be written</td>
</tr>
<tr>
<td>alu_sel</td>
<td>3</td>
<td>select ALU function</td>
</tr>
<tr>
<td>c_in</td>
<td>1</td>
<td>carry in to low-order bit of ALU</td>
</tr>
<tr>
<td>mar_sel</td>
<td>1</td>
<td>specifies whether MAR should be loaded</td>
</tr>
<tr>
<td>mdr_sel</td>
<td>2</td>
<td>specifies whether MDR is loaded</td>
</tr>
<tr>
<td>result</td>
<td>1</td>
<td>value to be loaded into the register file</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>read from memory</td>
</tr>
<tr>
<td>write</td>
<td>1</td>
<td>write to memory</td>
</tr>
</tbody>
</table>

Status Signals - Summary

<table>
<thead>
<tr>
<th>Signal</th>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_out</td>
<td>1</td>
<td>carry out of high-order bit</td>
</tr>
<tr>
<td>m_sign</td>
<td>1</td>
<td>sign bit of the ALU output</td>
</tr>
<tr>
<td>wait</td>
<td>1</td>
<td>memory operation complete</td>
</tr>
</tbody>
</table>
Instruction Fields

- instructions use general purpose registers
  - r0-r3 are general purpose
  - r4-r6 temp registers for control programs
  - r7 program counter
- instruction formats will contain all or some of
  - opcode – what instruction to perform
  - ri – one of the registers to be used
  - rj – one of the registers to be used
  - rk – one of the registers to be used
  - constant – a constant used by the instruction

Registers Fields

- ri – always the destination register
- rj and rk – source registers, read during instruction execution
- any combination of above can be the same register
- since r0-r3 are general purpose, we need 2 bits to specify each of the above
- if all three fields are used 6 bits are thus needed

Other Fields

- constant or literal
  - two’s complement constant
  - selected by result multiplexor and loaded into a datapath register
- instructions may or may not have the above fields
- opcode – must have field
  - determines the operation to be performed
  - width of field determines the number of instructions

Instruction Width

- if we have two sources and a destination register, we use 6 bits
- data path is 8 bits wide, so this leaves 2 bits left
- so what to do?
- use multiple words for the instruction
- for myth8 we will use two bytes – 16 bits

Instruction Formats

- type one instruction
  - uses opcode, ri, rj, rk, constant
  - opcode width + constant width = 10 bits
  - set opcode field to 6 bits – allows 64 operations
  - constant is 4-bits
- type two instruction
  - uses opcode, ri, constant
  - 16 – 2 – 6 = 8 bits for constant
  - all instructions will fit into these two types
  - not always a clear choice of type

Control-Code

- fetch0: a_reg=7, b_reg=7, alu_sel=AND, r6_write, mar_write=LOAD;
- fetch1: r7_write, a_reg=6, alu_sel=ADDA, c_in, read, ir1_write=LOAD;
  if wait then goto fetch1 endif;
- fetch2: a_reg=7, b_reg=7, alu_sel=AND, r6_write, mar_write=LOAD;
- fetch3: r7_write, a_reg=6, alu_sel=ADDA, c_in, read, ir1_write=LOAD;
  if wait then goto fetch1 endif;
- fetch4: goto opcode[ir_opcode];
### Instruction Length
- **fixed size instructions**
  - all instructions are 16 bits
  - fetch phase is independent of what instruction is being fetched
  - until the final indexed jump
- **variable length instructions**
  - fetch phase needs to determine the size of the instruction, normally based on opcode
  - and fetch the remainder of the instruction
  - this complicates the fetch phase

### Summary
- control unit receives these status lines
  - $m_7$ – high order bit (sign bit) of the ALU output
  - $c_{out}$ – carry out of the ALU bit 7
  - $v$ – overflow, (equal to $c_{out7} \oplus c_{out6}$)
  - $ir$ – instruction being executed
  - wait – memory operation still being performed
- ISA has been described
  - 64 opcodes
  - two instruction types
    - opcode, three registers, and 4 bit literal
    - opcode, one register, and 8 bit literal

### Summary (contd)
- fetch-execute cycle has been discussed
- memory interface has been augmented
  - 16-bit instruction register
  - some bits are status bits, others are data
  - opcode of $ir$ is used to branch in execute cycle
  - register specifier fields allow registers to be named
- register file has been divided
  - general purpose registers r0–r3
  - program counter r7
  - temporary value registers r4–r6

### Using Ir Fields in Control Unit
- adding hardware to determine $a_{def}$, $b_{def}$, and $r<i>_write$.
  - many control states can be saved
- can specify hardcoded registers (when using temporaries)
- or specify registers directly from the appropriate ir fields

### Three New Control Signals
- $r_{def}$
  - 0, constant is taken from the control program
  - 1, value is taken from the ir field in the ir
  - only one of $r$ fields through $r3$ can be written
  - can also specify r4 through r7 by asserting $r_{write}$
- $r_{use}$
  - 1, selects $r$ field of ir as the register to put on $a_{def}$
  - 0, use $a_{def}$ lines
- $r_{halt}$
  - 1, selects $r$ field of ir as the register to put on $b_{def}$
  - 0, use $b_{def}$ lines

### Need to Rename Hardware Lines
- add a $v$ before the names
  - to differentiate between hardware and software
  - also our previous control programs can still work
- $v_{r_{write}}$
  - $v_{r_{write}} = 1$ if $r_{write} = 1$
  - $v_{r_{write}} = 0$ if $r_{write} = 0$
- $v_{b_{def}}$
  - $v_{b_{def}} = 1$ if $b_{def} = 1$
  - $v_{b_{def}} = 0$ if $b_{def} = 0$
- $v_{a_{def}}$
  - $v_{a_{def}} = 1$ if $a_{def} = 1$
  - $v_{a_{def}} = 0$ if $a_{def} = 0$
Circuitry to Implement This

• multiplexor for \( v_{a_{out}} \) and \( v_{b_{out}} \)
• demux for \( r_i \), then a multiplexor for the two choices
• register convention simplifies implementation
  – \( r_i \) for the destination register
  – \( r_j \) and \( r_k \) for the source registers
  – also makes circuitry faster
  – deters adding new instruction formats
• low performance are inherently generic
• high performance are inherently specialized

Example

• \( \text{ble } r_j, r_k, \text{rel}_\text{addr} \)
  – instruction is at addr 1
  – branch condition is true
  – target address is \( 1 + 2 \times \text{rel}_\text{addr} \)
  – where does the +2 come from?

How to Compare Two Registers

• simplest is to subtract two registers
• unfortunately this doesn’t always work
• subtracting smallest neg from largest pos
  – overflow occurs
  – underflow might occur in other cases
• correct way is to perform the subtraction, and
  look at the following status signals
  – \( z \): the result of the ALU operation is zero
  – \( s \): the sign bit (also called \( m_7 \))
  – \( v \): overflow (\( c_{out7} \oplus c_{out6} \))

Comparisons

<table>
<thead>
<tr>
<th>comparison</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a-b )</td>
<td>( 2 \times (s=v) )</td>
</tr>
<tr>
<td>( a\geq b )</td>
<td>( s=v )</td>
</tr>
<tr>
<td>( a=b )</td>
<td>( s=v )</td>
</tr>
<tr>
<td>( a\leq b )</td>
<td>( s=v )</td>
</tr>
<tr>
<td>( a \leftrightarrow b )</td>
<td>( s=v )</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>( z )</td>
</tr>
<tr>
<td>( a \leftrightarrow b )</td>
<td>( z )</td>
</tr>
</tbody>
</table>
Tiny Instruction Set
- we have discussed a complete CPU (a very simple one, but it is complete)
- now we should discuss an instruction set
- of course it will be a tiny instruction set
- will allow us to examine how to implement an instruction set
- as well as examine tradeoffs in designing instruction sets

Instruction Set
- 7 instructions
  - 0: noop
  - 1: ri <- rj + rk
  - 2: ri <- const8
  - 3: bzero rj, rel_addr
  - 4: ri <- rj
  - 5: Mem[rj] <- rk
  - 6: ri <- Mem[rj]

Control Programs
- 0: noop just a goto fetch0
- 1: just an ALU add then goto fetch0
- 2: result = IR_CONST8, ri goto fetch0;
- 3: a little more complex
  - if c then goto fetch0 else goto branch endif;
- 4: ri sel, rj sel, alu sel = ADD, goto fetch0;
- 5: previously described store
- 6: previously described load

Indexed Jumps Allowed
- allow only a few indices to be used in jumps
- for us we only use one: ir opcode
  - (c ? addrtrue + addfalse) + (index ? ir opcode: 0)
    - (x ? a:b) if x = 1, select a, otherwise b
    - first select either true or false address
    - then choose to add to opcode base address or 0

Placement in Control Store
- fetch instructions are at 0-4
- base opcode addresses at 5-68
- rest of the statements would start at 69
- branching hardware can implement
  - if then else
    - unconditional branches
    - sequential execution