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 $r<j>_{write}$'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
  - $r<j>_{write}$ – to tell each 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

control signals

Signal	Bits	Purpose
a_{sel}	3	select register for a bus
b_{sel}	3	select register for b bus
$r<i>$	8	select registers to be written
alu_{sel}	3	select ALU function
c_{in}	1	carry in to low-order bit of ALU

status signals

Signal	Bits	Purpose
c_{out}	1	carry out of high-order bit
m_7	1	sign bit of the ALU output

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_{out}
- value m is selectively loaded into zero or more result registers – controled 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
  - alu_{sel} – alu_{sel2}, alu_{sel1}, and alu_{sel0}

Example

- $r0_{write}$, $r5_{write}$, a_{sel0}, b_{sel1}, alu_{sel2} means
  - register r1 is placed on a ($a_{sel} = 001_2$)
  - register r2 is placed on b ($b_{sel} = 010_2$)
  - sum of a and b is computed by ALU ($alu_{sel} = 100_2$)
  - result is stored in r0 and r5 ($r0_{write}$ and $r5_{write}$ 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_{sel}=<i>, b_{sel}=<i>, alu_{sel}=AND, if m_7 then goto shift1 else goto shift0 endif;

shift1: $r<i>_{write}$, a_{sel}=<i>, b_{sel}=<i>, alu_{sel}=ADD, c_{in} = 1, goto next;

shift0: $r<i>_{write}$, a_{sel}=<i>, b_{sel}=<i>, alu_{sel}=ADD, c_{in}=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

$ctrl\_program \Leftarrow \{ctrl\_stmt\}^+$

$ctrl\_stmt \Leftarrow \{ctrl\_label\}\{ctrl\_signals\ ,\}^+\{ctrl\_branch\};$

$ctrl\_label \Leftarrow name : \mid name[const]$

$ctrl\_signals \Leftarrow \left\{ctrl\_signal\ \{,ctrl\_signal\}^+\right\}$

$ctrl\_signal \Leftarrow name \mid name = CONST$

$ctrl\_branch \Leftarrow$ **goto** $name \mid$

if $cond$ **then goto** $name$ **endif** $\mid$

if $cond$ **then goto** $name$ **else goto** $name$ **endif** $\mid$

goto $name$ [cond_bits]

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

name	value	purpose
HOLD	0	contents of MAR held constant
LOAD	1	MAR loaded from output of ALU

MAR_{sel}

name	value	purpose
HOLD	0	MDR value can be read by both the result mux and memory
LOAD_ALU	1	load MDR from ALU output
LOAD_MEM	2	load MDR from memory

MDR_{sel}

Control Signals contd

name	value	purpose
ALU	0	output of the ALU
MEM	1	value of the MDR register

$result_{sel}$

- read – specifies to read the memory
- write – specifies to write to the memory
- wait – specifies whether the operation completed

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>
  a_{sel}=<j>, b_{sel}=<j>, 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
  a_{sel}=<j>, b_{sel}=<j>, alu_{sel}=AND, mar_{sel} = LOAD;
  a_{sel}=<k>, b_{sel}=<k>, 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

Signal	Bits	Purpose
a_{sel}	3	select register for a bus
b_{sel}	3	select register for b bus
r<i>	8	select registers to be written
alu_{sel}	3	select ALU function
c_{in}	1	carry in to low-order bit of ALU
mar_{sel}	1	specifies whether MAR should be loaded
mdr_{sel}	2	specifies whether/from where MDR is loaded
$result_{sel}$	1	value to be loaded into the register file
read	1	read from memory
write	1	write to memory

Status Signals - Summary

Signal	Bits	Purpose
c_{out}	1	carry out of high-order bit
m_7	1	sign bit of the ALU output
wait	1	memory operation complete

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_{sel}=7, b_{sel}=7, alu_{sel}=AND, $r6_{write}$, mar_{sel}=LOAD;
- fetch1: $r7_{write}$, a_{sel}=6, alu_{sel}=ADDA, c_{in}, read, $ir0_{sel}$=LOAD;
  if wait then goto fetch1 endif;
- fetch2: a_{sel}=7, b_{sel}=7, alu_{sel}=AND, $r6_{write}$, mar_{sel}=LOAD;
- fetch3: $r7_{write}$, a_{sel}=6, alu_{sel}=ADDA, c_{in}, read, $ir1_{sel}$=LOAD;
  if wait then goto fetch1 endif;
- fetch4: goto opcode[ir_{opcode}];

Instruction Length

- fixed size instructions
  - all instructions are 16bits
  - 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_{sel}, b_{sel}, 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

- ri_{sel}
  - 0, constant is taken from the control program
  - 1, value is take from the ri field in the ir
  - only one of regs r0 through r3 can be written
  - can also specify r4 through r7 by asserting $r<i>_{write}$
- rj_{sel}
  - 1, selects rj field of ir as the register to put on a_{sel}
  - 0, use a_{sel} lines
- rk_{sel}
  - 1, selects rk field of ir as the register to put on b_{sel}
  - 0, use b_{sel} 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
- $vr<l>_{write}$
  - $ir_{ri}=l$ if $ri_{sel} = 1$
  - $r<l>_{write}$ if $ri_{sel} = 0$
- va_{sel}
  - ir_{rj} if $rj_{sel} = 1$
  - a_{sel} if $rj_{sel} = 0$
- vb_{sel}
  - ir_{rk} if $rk_{sel} =1$
  - b_{sel} if $rk_{sel} = 0$

Circuitry to Implement This

- multiplexor for va_{sel} and vb_{sel}
- demux for ir_{ri}, then a multiplexor for the two choices
- register convention simplifies implementation
  - ir_{ri} for the destination register
  - ir_{rj} and ir_{rk} for the source registers
  - also makes circuitry faster
  - deters adding new instruction formats
- low performance are inherently generic
- high performance are inherently specialized

Example

- ble rj, rk, rel_addr
  - instruction is at addr l
  - branch condition is true
  - target address is l + 2 + rel_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

comparison	equivalent to
a>b	$\overline{z} \wedge (s=v)$
a>=b	(s=v)
a<b	$s \oplus v$
a<=b	$z \vee (s \oplus v)$
a=b	z
a <> b	$\overline{z}$

Example

```
// blt rj,rk,rel_addr
//    branch iff m7 ⊕ v
rjsel, rksel, alusel = SUB, cin,
      if m7 then goto neg else goto pos endif;
neg: rjsel, rksel, alusel = SUB, cin,
      if v then goto fetch else goto branchlt endif;
pos: rjsel, rksel, alusel = SUB, cin,
      if v then goto branchlt else goto fetch endif;
branchlt: resultsel = IR_CONST4, r6write;
   asel=7, bsel=6, alusel=ADD, r7write, goto fetch;
```


Signal Summary

- 30 total control signals
  - register file: ri_{sel}, rj_{sel}, rk_{sel}, $r<i>_{write}$, a_{sel}, b_{sel}
  - ALU: alu_{sel}, c_{in}
  - memory: mar_{sel}, mdr_{sel}, read, write
  - instruction register: $ir0_{sel}$, $ir1_{sel}$
  - result: $result_{sel}$
- 16 status signals
  - instruction register: ir_{opcode}, ir_{ri}, ir_{rj}, ir_{rk},
  - ALU: c_{out}, m_7, v
  - memory: wait

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: $result_{sel}$=IR_CONST8, ri_{sel}, goto fetch0;
- 3: a little more complex
  - rj_{sel}, alu_{sel}=SUBA, $r6_{write}$, $result_{sel}$=IR_CONST4,
    if c_{out} then goto fetch0 else goto branch endif;
  - branch: $r7_{write}$, a_{sel}=7, b_{sel}=6, alu_{sel}=ADD, goto fetch0;
- 4: ri_{sel}, rj_{sel}, alu_{sel}=ADDA, 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 ? $addr_{true}$:$addr_{false}$) + ($index_{sel}$? 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

