Today

- Finish Basic ISA
- Code examples

Computation Instructions (1)

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
add	Add	reg	reg	op1 = op1 + op2	pc: increment sreg: Set N, Z, C
sub	Subtract	reg	reg	op1 = op1 - op2	pc: increment sreg: Set N, Z, C
inc	Increment register	reg	N/A	op1 = op1 + 1	pc: increment sreg: Set N, Z, C
dec	Decrement register	reg	N/A	op1 = op1 - 1	pc: increment sreg: Set N, Z, C
and	And	reg	reg	op1 = op1 & op2	pc: increment sreg: Set N, Z
or	Or	reg	reg	op1 = op1 | op2	pc: increment sreg: Set N, Z
eor	Exclusive-or	reg	reg	op1 = op1 ^ op2	pc: increment sreg: Set N, Z
not	Not	reg	N/A	op1 = ~op1	pc: increment sreg: Set N, Z

Computation Instructions (2)

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
cp	Compare two registers	reg	reg	**none**	pc: increment N: set if op1 - op2 < 0 Z: set if op1 - op2 is 0 C: set if op1 - op2 > 255

What is it useful for?

Comparison	**ISA ops to jump 50 instructions if the comparison holds**
r30 is equal to r31	cp r30, r31 breq 50
r30 is not equal to r31	cp r30, r31 brne 50
r30 is less than r31	cp r30, r31 brlo 50
r30 is greater than r31	cp r31, r30 brlo 50
r30 is greater than or equal to r31	cp r30, r31 brsh 50
r30 is less than or equal to r31	cp r31, r30 brsh 50

Computation Instructions (3)

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
subi	Subtract immediate	reg 16-31	value 0-255	op1 = op1 - op2	pc: increment Set N, Z, C
cpi	Compare with immediate	reg 16-31	value 0-255	none	pc: increment Set N, Z, C
andi	Subtract immediate	reg 16-31	value 0-255	op1 = op1 & op2	pc: increment Set N, Z, C
ori	Subtract immediate	reg 16-31	value 0-255	op1 = op1 | op2	pc: increment Set N, Z, C

Computation Instructions (4)

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
adc	Add with carry	reg	reg	op1 = op1 + op2 + C	pc: increment Set N, Z, C
sbc	Subtract with carry	reg	reg	op1 = op1 - op2 - C	pc: increment Set N, Z, C
asr	Arithmetic shift right	reg	None	op1 = op1 >> 1 MSB unmodified	pc: increment Set C=old LSB Set N, Z

Input/Output

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
in	Read I/O register	reg	value 0-63	Read op2 into op1	pc: increment sreg: unchanged
out	Write to I/O register	value 0-63	reg	Write op2 to op1	pc: increment sreg: unchanged

Flow control

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials
rjmp	Relative jump	Value: -2048 to 2047	none	none	pc = pc + op1 sreg: unchanged
breq	Branch if equal	Value: -64 to 63	none	none	If Z set, pc = pc + op1 pc increment sreg: unchanged
brne	Branch if not equal	Value: -64 to 63	none	none	If Z clear, pc = pc + op1 pc increment sreg: unchanged
brsh	Branch if same or higher	Value: -64 to 63	none	none	If C set, pc = pc + op1 pc increment sreg: unchanged
brlo	Branch if lower	Value: -64 to 63	none	none	If C clear, pc = pc + op1 pc increment sreg: unchanged

Inst.	Description	Op 1	Op 2	Effect on operands	Effect on specials	Example
Load/Store Instructions						
ldi	Load immediate	reg 16-31	value 0-255	op1 = op2	pc: increment sreg: unchanged	`ldi r16, 45` `ldi r31, -23`
mov	Move	reg	reg	op1 = op2	pc: increment sreg: unchanged	`Mov r1, r2` `mov r3, r4`
ld	Load from RAM	reg	X	op1 = RAM[r26 + 256*r27]	pc: increment sreg: unchanged	`ld r1, X` `ld r2, X`
st	Store in RAM	X	reg	op1 = RAM[r26 + 256*r27] = op2	pc: increment sreg: unchanged	`st X, r1` `st X, r2`
Computation Instructions						
add	Add	reg	reg	op1 = op1 + op2	pc: increment sreg: Set N, Z, C	`add r1, r2` `add r2, r4`
sub	Subtract	reg	reg	op1 = op1 - op2	pc: increment sreg: Set N, Z, C	`sub r1, r2` `sub r3, r7`
inc	Increment register	reg	N/A	op1 = op1 + 1	pc: increment sreg: Set N, Z, C	`inc r1` `inc r12`
dec	Decrement register	reg	N/A	op1 = op1 - 1	pc: increment sreg: Set N, Z, C	`dec r1` `dec r17`
and	And	reg	reg	op1 = op1 & op2	pc: increment sreg: Set N, Z	`and r1, r2`
or	Or	reg	reg	op1 = op1 | op2	pc: increment sreg: Set N, Z	`or r1, r2`
eor	Exclusive-or	reg	reg	op1 = op1 ^ op2	pc: increment sreg: Set N, Z	`eor r1, r2`
not	Not	reg	N/A	op1 = ~op1	pc: increment sreg: Set N, Z	`not r1` `not r2`
cp	Compare two registers	reg	reg	none	pc: increment N: set if op1 - op2 < 0 Z: set if op1 - op2 is 0 C: set if op1 - op2 > 255	`cp r1, r2`
subi	Subtract immediate	reg 16-31	value 0-255	op1 = op1 - op2	pc: increment Set N, Z, C	`subi r16, 10` `subi r17, -15`
cpi	Compare with immediate	reg 16-31	value 0-255	none	pc: increment Set N, Z, C	`cpi r17, 10` `cpi r19, -10`
andi	Subtract immediate	reg 16-31	value 0-255	op1 = op1 & op2	pc: increment Set N, Z, C	`andi r16, 1` `andi r17, 2`
ori	Subtract immediate	reg 16-31	value 0-255	op1 = op1 | op2	pc: increment Set N, Z, C	`ori r16, 1` `ori r17, 1`
adc	Add with carry	reg	reg	op1 = op1 + op2 + C	pc: increment Set N, Z, C	`adc r1, r2` `adc r1, r3`
sbc	Subtract with carry	reg	reg	op1 = op1 - op2 - C	pc: increment Set N, Z, C	`sbc r1, r2` `sbc r2, r3`
asr	Arithmetic shift right	reg	None	op1 = op1 >> 1 MSB unmodified	pc: increment Set C=old LSB Set N, Z	`asr r1` `asr r2`
Input/Output						
in	Read I/O register	reg	value 0-63	Read op2 into op1	pc: increment sreg: unchanged	`in r1, 16`
out	Write to I/O register	value 0-63	reg	Write op2 to op1	pc: increment sreg: unchanged	`out 18, r1` `out 17, r31`
Control-Flow						
rjmp	Relative jump	Value: -2048 to 2047	none	none	pc = pc + op1; sreg: unchanged	`rjmp 4` `rjmp -3`
breq	Branch if equal	Value: -64 to 63	none	none	If Z set, pc = pc + op1 pc increment sreg: unchanged	`breq 2` `breq -3`
brne	Branch if not equal	Value: -64 to 63	none	none	If Z clear, pc = pc + op1 pc increment sreg: unchanged	`brne 2` `brne -3`
brsh	Branch if same or higher	Value: -64 to 63	none	none	If C set, pc = pc + op1 pc increment sreg: unchanged	`brsh 2` `brsh -3`
brlo	Branch if lower	Value: -64 to 63	none	none	If C clear, pc = pc + op1 pc increment sreg: unchanged	`brlo 2` `brlo -3`

Look at Program

Some problematic Instructions

- `ld r1, X`
- `cp r1, r2`
- `breq 2`
  `brlo -2`

Some basic programming
Large numbers

- Combine adjacent registers to form longer values
- r30:r31 = 256*r30 + r31
  (16 bits)
- r29:r30:r31 = 65536*r29 + 256*r30 + r31
  (24 bits)
- r28:r29:r30:r31 = 16777216*r28 + 65536*r29 + 256*r30 + r31
  (32 bits)

Some basic programming
Large numbers

```
add_big_nums:
add r31,r27
adc r30,r26
adc r29,r25
adc r28,r24
```

Some basic programming?
If statement

```
if (x == y):
   stuff to do if equal
else:
   stuff to do if not equal
```

```
compare the two numbers
breq jump_here_if_equal
stuff to do if not equal
…
…
jump to end_of_conditional_code
jump_here_if_equal:
stuff to do if equal
…
…
end_of_conditional_code:
```

Some basic programming if statement

```
if x == y
        x = x + 1
else
        y = y + 1
z = z + x + y
```

```
; x is in r30
; y is in r31
; z is in r29
cp r30, r31
breq 2
inc r31
rjmp 1
inc r30
add r29, r30
add r29, r31
```

while

```
while (x == y):
    do stuff

while (x== y):
    print(x)
    x = x - 2
```

```
begin_while:
cp r0,r1
brne end_while
do stuff
rjmp begin_while
end_while:
```

while

```
while (x == y):
    do stuff

while (x== y):
    print(x)
    x = x - 2
```

```
; x in r30
; y in r31
ldi r30, 45
ldi r31, 45
ldi r29, -1
out 17, r29
cp r30,r31
brne 3
out 18, r30
subi r30, 2
rjmp -5
```

Addressing Modes

- Addressing modes refers to how we access memories
- We have introduced 3 addressing modes for data
  - Immediate (ldi)
  - Register indirect (ld r1, X)
  - Register autoincrement (ld r1, X+)
- We have introduced one addressing modes for code
  - PC relative

Summary

- ISA is instruction set architecture
  - Defines architecture state (number, width of registers)
  - Addressing modes
  - Instructions
  - Encoding
- AVR ISA
- Wednesday: Assembly language

CS 252
Lecture 14; 2015 Oct 12th; Transcribed Lecture notes

More examples of 32-bit Floating Point Representation

Appended to last Wednesday, October 7th, Lecture 13 Notes

Finish Basic Instruction Set Architecture (ISA) [see slides for full table]

Storage Instructions

Instr	**Description**	**Op1**	**Op2**	**Effect on Operands**	**Effects on Specials**
ldi	load immediate	reg 16-31	value -128 to 127		
mov	move	reg	reg		
ld	load from RAM	reg	X		
st	store in RAM	X	reg		

Computation Instructions

Instr	**Description**	**Op1**	**Op2**	**Effect on Operands**	**Effects on Specials**
add	add	reg			
sub	subtract	reg			
inc	increment register	reg			
dec	decrement register	reg			
and	bitwise and	reg			
or	bitwise or	reg			
eor	bitwise exclusive-or	reg			
not	bitwise not/invert	reg			
cp	compare two registers. Used for determining whether to branch	reg	reg	none	pc: increment N: set if op1-op2<0 Z: set if op1-op2 is 0 C: set if op1-op2>255
subi	subtract immediate				
cpi	compare with immediate				

andi	and immediate (typo in slides)				(typo in slides, should not have C)
ori	or immediate (typo in slides)				(typo in slides, should not have C)
adc	add with carry				
sbc	subtract with carry				
asr	arithmetic shift right. shift bits, equivalent to dividing by powers of 2			MSB unmodified	

What is compare useful for?

Used to check if 2 registers are equal. If so, it will set flags. Then control-flow instruction will use the flag to determine whether to branch.

How did these instructions came about?

Arbitrary, depends on the application.

These instructions do not look very nice. This is because these ISA were developed 10+ years ago when resources was very limited. However, we keep using these ugly instructions because of backwards compatibility.

Every ISA will have registers that look slightly different. The instructions will look slightly different. The source and destination register may also have different positions.

Input/Output Instructions

Instr	**Description**	**Op1**	**Op2**	**Effect on Operands**	**Effects on Specials**
in	read I/O register				
out	write to I/O register				

We are only concerned with I/O register 16, 17, and 18 because they connect to outside pins. IO register 16 maps to PIND, IO register 17 maps to DDRD, and IO register 18 maps to PORTD. See bottom right of AVR simulator.

Control-Flow Instructions

Instr	**Description**	**Op1**	**Op2**	**Effect on Operands**	**Effects on Specials**
rjmp	relative jump				
breq	branch if equal				
brne	branch if not equal				
brsh	branch if same or higher				
brlo	branch if lower				

Use the cp instruction before using these instructions.

Code Examples: Link High Level Python to Low Level AVR Assembly Language

High Level Programming Language (Python)

```
x = 20
while(x != 1):
     y = x % 2
     if(y == 0):
          x = x / 2
     else:
          x = 3 * x + 1
     print(x)
print("Done")
```

Low Level Assembly Language (AVR)

```
ldi r16,20     ; load 20 into register r16
ldi r17,1      ; load 1 into register r17
ldi r31,-1     ; load -1 into register r31
out 17,r31     ; write value in register r31 to IO register 17
out 18,r16     ; write value in register r17 to IO register 16
cp r16,r17     ; check if x == 1, so compare, if equal, then our Z flag will be set
breq 11        ; if equal, then branch
mov r18,r16    ; load to do y = x % 2
andi r18,1     ; bitwise and of 1, to check the least significant bit, if 1, then it is odd
cp r18,r17     ; compare to check if odd
breq 2         ; if equal, then branch forwards by 2 steps
asr r16        ; else if not equal, then arithmetic shift, which is divide by 2
rjmp -9        ; branch back to printing new value on screen to do check all over again
mov r19,r16    ; now to 3 * x + 1, take x and move it into a register
add r19,r19    ; add it to itself two more times to get 3 * x
add r16,r19    ; same as above
inc r16        ; increment 3 * x to finally get 3 * x + 1
rjmp -14       ; now jump back to top of program to do this 1 more time
```

jump vs branch?

rjmp will change control flow unconditionally

branch will change control flow based on a condition flag