1. PowerPC Assembly

- OS X and Linux different ABI
  → Linux less bone-headed
- OS X and Linux slightly different assembly:
  → OS X less bone-headed
  - OS X calls iregs `r0-r31` and fpregs `f0-f31` and
  - Linux calls iregs `0-31` and fpregs `0-31`
    ! Constants 0-31 are also `0-31`
  ⇒ Can use cpp tricks to work around naming differences
- Suffixing many ops with '.' makes them update CR
- PPC has big-endian byte and bit ordering
  - Sign bit bit 0 in first byte
  - most sig bit (MSB) at bit 0
  - LSB at bit 63
  - low_{32} bits = 32-64
  - hi_{32} bits = 0-31
  - bit 0 at (1¡¡63)
- I'm going to use little endian bit ordering, but PPC docs and explicit bit numbers assume big

2. CPP trick for Mixed OS X/Linux Assembly

- Need to rename registers for Linux as shown at right
- Must have differing prologue/epilogue/reg usage to respect differing ABI

```
#ifdef ATL_GAS_LINUX_PPC
   #define r0 0
   #define r1 1
   ...
   #define r31 31

   #define f0 0
   #define f1 1
   ...
   #define f31 31
#endif
```

3. OS X PPC Register Usage

REGISTER	USAGE	CALLEE SAVE
Integer Registers		
r0*	Used in prolog/epilog	NO
r1	Stack pointer	YES
r2	TOC pointer (reserved)	YES
r3	1st para/return	NO
r4	2nd para/return	NO
r5-r10	3-8th para	NO
r11	Environment pointer	NO
r12	Used by global linkage	NO
r13-31	Global int registers	YES
Floating Point Registers		
f0	Scratch reg	NO
f1-13	1-13th fp para	NO
f14-f31	Global fp regs	YES
LR,CTR,XER	special regs	NO
CR[0-1,5-7]	condition regs	NO
CR[5-7]	condition regs	YES

- `r0` interp as 0 in EA calculations
- If fp para, the appropriate number of para-passing iregs are skipped

4. OS X Calling Sequence and Stack Frame

- Stack grows downward
- Caller puts callees' args in its frame
- Frame 16-byte aligned
- Caller saves LR and CR only if it wants to
- 8 word mandatory para area not init (in reg); there for callee use
- Has 224-byte red zone

Stack frame passed to callee

	fp reg save area (optional)
	ireg save area (optional)
	padding (optional)
	Local storage (optional)
24(r1)	Parameter area (>= 8 words)
20(r1)	TOC save area
16(r1)	Link editor doubleword
12(r1)	Compiler doubleword
8(r1)	Link register (LR) save
4(r1)	Condition register (CR) save
0(r1)	ptr to callee's stack

5. Linux PPC Register Usage

REGISTER	USAGE	CALLEE SAVE
Integer Registers		
r0*	Used in prolog/epilog	NO
r1	Stack pointer	YES
r2	TOC pointer (reserved)	YES
r3-r4	1/2 para and return	NO
r5-r10	3-8th integer para	NO
r11-r12	Func linkage regs	NO
r12	Used by global linkage	NO
r13	Small data area ptr reg	NO
r14-30	Global int registers	YES
r31	Global/environment ptr	YES
Floating Point Registers		
f0	Scratch reg	NO
f1	1st para / return	NO
f2-8	2-8th fp para	NO
f9-f13	Scratch reg	NO
f14-f31	Global fp regs	YES

- `r0` interp as 0 in EA calculations
- fp para, does *not* cause skip of iregs

6. Linux PPC Calling Sequence and Stack Frame

- Stack pointer 16-byte aligned.
- Stack pointer (sp) updated atomically via "store word with update".
- Any non-scratch reg f# must be saved to the fp reg save area at location 8*(32-#) from the previous frame (i.e., the register f31 is saved adjacent to previous ptr to callee's stack).
- Any non-scratch reg r# must be saved in the ireg save area 4*(32-#) bytes before the low-addressed end of the fp reg save area.
- Minimum stack frame callee save and LR save.
- No red zone is specifically mandated.
- If number if ipara is ≤ 8 and the number of fppara ≤ 8, then no values are stored in the parameter area, and if this is true for all calls made by the routine, the parameter area will be of size 0.

Stack frame passed to callee

	fp reg save area (optional)
	ireg save area (optional)
	CR save area (optional)
	Local storage (optional)
8(r1)	Parameter area (optional)
4(r1)	Link register (LR) save
0(r1)	ptr to callee's stack

7. PowerPC System Registers

1. **CR**: Condition Register - 32 bits, set implicitly or by [f]cmp
   - `mfcr r`x: rx_{0-31} = CR; rx_{32-63} = 0
   - `mtcr r`x: CR = rx_{0-31}
   - Have bunch of boolean inst as well
2. **CTR**: Count Register
   - `mfctr r`x: rx_{0-31} = CTR; rx_{32-63} = 0
   - `mtctr r`x: CTR = rx_{0-31}
3. **LR**: Link Register - Used to save address to return to
   - `mflr r`x: rx = LR
   - `mtlr r`x: LR = rx
4. **XER**: Fixed-pt Exception Register
   - `mfxer r`x: rx_{0-31} = XER; rx_{32-63} = 0
   - `mtxer r`x: XER = rx_{0-31}
5. **FPSCR**: FP Status & Control Register
   - `mffs r`x: rx_{0-31} = FPSCR; rx_{32-63} = undef
   - `mtfs r`x: FPSCR = rx_{0-31}
   - Other bit/mask forms as well
6. **MSR**: Machine State Register
   - `mfmsr r`x: rx = MSR
   - `mtmsr r`x: MSR = rx

8. PowerPC Assembly Overview

- Three-operand assembly (some 4-op mnem):
  - `<mnem> <dest>, <src1>, <src2>`
- ld/st only inst that take addresses
- inst that take immediates end in `i` (eg, `andi`).
  - Most inst take signed 16-bit immediates
- Both 16-bit imm & 32-bit reg are sign extended for most ops
- Other inst have only reg operands
- No prefix for reg or constants!
- To make inst affect CR, suffix them with '.'
- Inst that are cracked (dec into 2 internal inst) on G5 marked by^c
  - take up extra room in group
- Inst that are microcoded (decoded into > 2 inst) marked by^m
  - Only 1 microcoded inst can be fetched from inst fetch buff into internal pipe per cycle

9. PowerPC Addressing Modes

Two general forms of addressing:

1. Offset: `imm16(rx)`
   - @ = rx + imm16
2. Indexed : `rx, ry`
   - @ = rx + ry

!! r0 interp as 0 (except when ry):

- `64(r0)`: @ = `64`
- `r0, r1`: @ = `r1`
- `r1, r0`: @ = `r1+r0`

Two types of ld to handle 64 bits

1. Load and zero
   - load to lower 32 bits, clear upper
   - suffix : `z`
2. Load Algebraic
   - load to lower 32 bits, sign extend upper
   - suffix : `a`

10. PPC Integer Transfer Operations

Mnemonic	Operands	Action
`lwz`	rz, i(rx)	rz_{0-31} = *(rx+i), rz_{32-63} = 0
`lwzu`^{c}	rz, i(rx)	rz_{0-31} = *(rx+i), rz_{32-63}=0, rx += i
`lwzx`	rz, rx, ry	rz_{0-31} = *(rx+ry), rz_{32-63} = 0
`lwzux`^{m}	rz, rx, ry	rz_{0-31} = *(rx+ry), rz_{32-63} = 0, rx += ry
`lwa`^{c}	rz, i(rx)	rz_{0-31} = *(rx+i), rz_{32-63} = *(rx+i)$_{31}$
`lwau`^{m}	rz, i(rx)	rz_{0-31} = *(rx+i), rz_{32-63}=*(rx+i)$_{31}$, rx += i
`lwax`^{c}	rz, rx, ry	rz_{0-31} = *(rx+ry), rz_{32-63} = *(rx+ry)$_{31}$
`lwaux`^{m}	rz, rx, ry	rz_{0-31} = *(rx+ry), rz_{32-63} = *(rx+ry)$_{31}$, rx += r
`stw`	rz, i(rx)	*(rx+i) = rz_{0-31}
`stwu`^{c}	rz, i(rx)	*(rx+i) = rz_{0-31}, rz += i
`stwx`^{c}	rz, rx, ry	*(rx+ry) = rz_{0-31}
`stwux`^{m}	rz, rx, ry	*(rx+ry) = rz_{0-31}, rx += ry
`mr`	rz, rx	rz = rx (*register copy*)
`li`	rz, i	rz_{0-15} = i, rz_{16-63} = 0
`lis`	rz, i	rz_{0-15} = 0, rz_{16-31} = i, rz_{32-63} = 0

- Setting rx as `r0` substitutes zero for rx. (illegal with update forms)
- Yes, STs take last arg as dest!

11. PPC 64 bit Integer LD/ST Operations

Mnemonic	Operands	Action
`ld`	rz, i(rx)	rz = *(rx+i),
`ldu`^{c}	rz, i(rx)	rz = *(rx+i), rx += i
`ldx`	rz, rx, ry	rz = *(rx+ry)
`ldux`^{m}	rz, rx, ry	rz = *(rx+ry), rz = 0, rx += ry
`std`	rz, i(rx)	*(rx+i) = rz
`stdu`^{c}	rz, i(rx)	*(rx+i) = rz, rz += i
`stdx`	rz, rx, ry	*(rx+ry) = rz
`stdux`^{m}	rz, rx, ry	*(rx+ry) = rz, rx += ry

- Setting rx as `r0` substitutes zero for rx. (illegal with update forms)

12. Common Integer Arithmetic Operations

Mnemonic	Operands	Action
`add[o][.]`	rz, rv, ry	rz = rv + ry
`addc`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = rv + ry, set XER[CA] if carry past 32 bit
`addi`	rz, rx, i	rz = rx + i
`addis`	rz, rx, i	rz = rx + (i<<16)
`addic[.`^{c}`]`	rz, rx, i	rz = rx + i set XER[CA] if carry past 32 bits
`sub[o`^{c}`][.]`	rz, rv, ry	rz = rv - ry
`subf[o`^{c}`][.]`	rz, rv, ry	rz = ry - rv
`subfc`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = ry + rv, set XER[CA] if carry past 32 bit
`subfic[o][.]`	rz, rv, i	rz = i - rv, set XER[CA] if carry past 32 bits
`neg[o][.]`	rz, ry	rz = -ry
`extsw[.`^{c}`]`	rz, ry	rz_{0-31} = ry_{0-31}, rz_{32-63} = ry_{31}
`cntlzw[.]`	rz, ry	rz = leading_zeros(ry_{0-31})
`cntlzd[.]`	rz, ry	rz = leading_zeros(ry)

Suffixes:

- '.': set CR[0] to indicate >, <, =
- 'o': inst sets overflow (XER[OV])

- if rx is r0, means zero
- i sign extended to 64 bits
- `nego.` cracked

13. Common Integer Multiply/Divide Operations

Mnemonic	Operands	Action
`mullw[o][.`^{c}`]`	rz, rv, ry	rz = rv_{0-31} * ry_{0-31}
`mulld[o][.`^{c}`]`	rz, rv, ry	rz = (rv * ry)$_{0-63}$
`mulhd[.`^{c}`]`	rz, rv, ry	rz = (rv * ry)$_{64-127}$
`mulhdu[.`^{c}`]`	rz, rv, i	rz = (rv * ry)$_{64-127}$ (unsigned)
`mulli`	rz, rv, i	rz = rv * i
`divw`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = rv_{0-31} / ry_{0-31}
`divwu`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = rv_{0-31} / ry_{0-31} (unsigned)
`divd`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = rv / ry
`divdu`^{c}`[o`^{m}`][.`^{m}`]`	rz, rv, ry	rz = rv / ry (unsigned)

Suffixes:

- '.': set CR[0] to indicate >, <, =
- 'o': inst sets overflow (XER[OV])

14. Common PPC Boolean Bit-Level Operations

Mnemonic	Operands	Action
`and[.]`	rz, rv, ry	rz = rv & rz
`andc[.]`	rz, rv, ry	rz = rv & (~rz)
`eqv[.]`	rz, rv, ry	bit in rz set if same bit in rx and ry match
`nand[.]`	rz, rv, ry	rz = ~(rv & rz)
`nor[.]`	rz, rv, ry	rz = ~(rv | rz)
`or[.]`	rz, rv, ry	rz = rv | rz
`orc[.]`	rz, rv, ry	rz = rv | (~rz)
`or[.]`	rz, rv, ry	rz = rv ^ rz
`andi.`	rz, rv, i	rz_{0-15} = rv & i, $rz_{16-63} = 0$
`andis.`	rz, rv, i	rz_{16-31} = rv & (i<<16), $rz_{32-63} = rz_{0-15} = 0$
`ori`	rz, rv, i	rz = rv | i
`oris`	rz, rv, i	rz = rv | (i<<16)
`xori`	rz, rv, i	rz = rv ^ i
`xoris`	rz, rv, i	rz = rv ^ (i<<16)

- Only ops ending in '.' update CR[0]

15. PPC Shift Operations

Mnemonic	Operands	Action
`slw[.`^{c}`]`	rz, rv, ry	rz_{0-31} = (rv_{0-31} << ry); rz_{32-63} = 0
`sld[.`^{c}`]`	rz, rv, ry	rz = (rv << ry)
`slwi[.]`	rz, rv, i	rz_{0-31} = (rv_{0-31} << i); rz_{32-63} = 0
`sldi[.]`	rz, rv, i	rz = (rv << i)
`srw[.`^{c}`]`	rz, rv, ry	rz_{0-31} = (rv_{0-31} >> ry); rz_{32-63} = 0
`srd[.`^{c}`]`	rz, rv, ry	rz = (rv >> ry)
`srwi[.]`	rz, rv, i	rz_{0-31} = (rv_{0-31} >> i); rz_{32-63} = 0
`srdi[.]`	rz, rv, i	rz = (rv >> i)
`sraw[.`^{c}`]`	rz, rv, ry	rz_{0-31} = (rv_{0-31} >> ry); rz_{32-63} = 0, dup sign bit
`srad[.`^{c}`]`	rz, rv, ry	rz = (rv >> ry), dup sign bit
`srawi[.`^{c}`]`	rz, rv, i	rz_{0-31} = (rv_{0-31} >> i); rz_{32-63} = 0, dup sign bit
`sradi[.`^{c}`]`	rz, rv, i	rz = (rv >> i), dup sign bit

- Also have mask & rotate inst
- Also have Extract & justify inst

$\Rightarrow$ Very useful, but we're not covering them.

16. PPC Condition Codes

Conditions signaled in condition register (CR):

- 8 different 4-bit CRs in 32-bit CR:
  - CR[0] implicit CR for iops
  - CR[1] implicit CR for fops
  - CR[2-7] used explicitly

- Int CR bits (big endian):
  0. **LT**: set if rv < ry
  1. **GT**: set if rv > ry
  2. **EQ**: set if rv = ry
  3. **SO**: set if overflow
- FP CR bits (big endian):
  0. **FL**: set if rv < ry
  1. **FG**: set if rv > ry
  2. **FE**: set if rv = ry
  3. **FU**: set if unordered

17. Common PPC Comparison Instructions

Mnemonic	Operands	Action
`cmpw`	cr$z$, rv, ry	set CR[z] rv_{0-31} ? ry_{0-31}
`cmplw`	cr$z$, rv, ry	set CR[z] rv_{0-31} ? ry_{0-31} (unsigned)
`cmpwi`	cr$z$, rv, i	set CR[z] rv_{0-31} ? sign_ext(i)
`cmplwi`	cr$z$, rv, i	set CR[z] rv_{0-31} ? zero_ext(i) (unsigned)
`cmpd`	cr$z$, rv, ry	set CR[z] rv ? ry
`cmpld`	cr$z$, rv, ry	set CR[z] rv ? ry (unsigned)
`cmpdi`	cr$z$, rv, i	set CR[z] rv ? sign_ext(i)
`cmpldi`	cr$z$, rv, i	set CR[z] rv ? zero_ext(i) (unsigned)

- Can do comparison(s) early, branch later
  $\rightarrow$ Can store multiple results in differing CR
  - Implicit icmp set CR[0]
  - Implicit fcmp set CR[1]

18. Common PPC Branch Instruction

Mnemonic	Operands	Action
`blr[l]`	none	(ret/call) jump to @ in LR
`b[l]`	label	unconditional jump
`beq[l][a]`	[cr$x$,] label	jump if CR[x][EQ] = 1
`bne[l][a]`	[cr$x$,] label	jump if CR[x][EQ] = 0
`blt[l][a]`	[cr$x$,] label	jump if CR[x][LT] = 1
`ble[l][a]`	[cr$x$,] label	jump if CR[x][GT] = 0
`bgt[l][a]`	[cr$x$,] label	jump if CR[x][GT] = 1
`bge[l][a]`	[cr$x$,] label	jump if CR[x][LT] = 0
`bso[l][a]`	[cr$x$,] label	jump if CR[x][SO] = 1
`bun[l][a]`	[cr$x$,] label	jump if CR[x][FU] = 1
`bnu[l][a]`	[cr$x$,] label	jump if CR[x][FU] = 0
`bdnz[l][a]`	label	jump if (−−CTR) ≠ 0

- `bne+ 3, LOOP`
  - Jump to LOOP label based on CR[3], jump likely taken
- `bgtla- r2`
  - Jump @ in r2 based on CR[0], jump unlikely taken, save @ of next inst in LR

- '**l**': store next inst @ in LR
- '**a**': label is absolute address (often in register)
- Can also give branch prediction suffix hint:
  - '+': branch likely to be taken
  - '-': branch unlikely to be taken
- Lot more br types!

19. PPC Floating Point

- FPU is all double precision, single load/store converts
- Computational fpinst take '**s**' suffix, to indicate single precision:
  → Causes single rounding for double register
  – Could instead use double computation, and convert only at end for extra precision
- Most instructions can be suffixed with '**.**' (not too useful most of the time IMHO) so that CR[1] indicates exception status, as shown below.

CR[1]	Name	Description
0	FX	Set on any exception
1	FEX	Set on any *enabled* exception
2	VX	Set on any invalid operation exception
3	OX	Set on overflow

20. PPC Floating Point Load/Stores/Move

Mnemonic	Operands	Action
`lf[d,s]`	fr$z$, i(rx)	fr$z$ = *(rx+i)
`lf[d,s]u`^{c}	fr$z$, i(rx)	fr$z$ = *(rx+i), rx += i
`lf[d,s]x`	fr$z$, rx, ry	fr$z$ = *(rx+ry)
`lf[d,s]ux`^{c}	fr$z$, rx, ry	rz = *(rx+ry), rx += ry
`stf[d,s]`	fr$z$, i(rx)	*(rx+i) = fr$z$
`stf[d,s]u`^{c}	fr$z$, i(rx)	*(rx+i) = fr$z$, rx += i
`stf[d,s]x`	fr$z$, rx, ry	*(rx+ry) = fr$z$
`stf[d,s]ux`^{c}	fr$z$, rx, ry	*(rx+ry) = fr$z$, rx += ry
`fmr[.]`	fr$z$, fr$x$	fr$z$ = fr$x$ (*register copy*)
`fabs[.]`	fr$z$, fr$x$	fr$z$ = |fr$x$|
`fnabs[.]`	fr$z$, fr$x$	fr$z$ = $-$|fr$x$|
`fneg[.]`	fr$z$, fr$x$	fr$z$ = $-$fr$x$
`frsp[.]`	fr$z$, fr$x$	fr$z$ = Double2Float(fr$x$)

- '**s**': convert float to double on ld/st
- '**d** : just load double fpnum into reg

21. Common PPC Floating Point Computation Instructions

Mnemonic	Operands	Action
`fmadd[s][.]`	fr$z$, fr$v$, fr$x$, fr$y$	fr$z$ = fr$v$ * fr$x$ + fr$y$
`fmsub[s][.]`	fr$z$, fr$v$, fr$x$, fr$y$	fr$z$ = fr$v$ * fr$x$ - fr$y$
`fnmadd[s][.]`	fr$z$, fr$v$, fr$x$, fr$y$	fr$z$ = -(fr$v$ * fr$x$ + fr$y$)
`fnmsub[s][.]`	fr$z$, fr$v$, fr$x$, fr$y$	fr$z$ = -(fr$v$ * fr$x$ - fr$y$)
`fadd[s][.]`	fr$z$, fr$x$, fr$y$	fr$z$ = fr$x$ + fr$y$
`fsub[s][.]`	fr$z$, fr$x$, fr$y$	fr$z$ = fr$x$ - fr$y$
`fmul[s][.]`	fr$z$, fr$x$, fr$y$	fr$z$ = fr$x$ * fr$y$
`fdiv[s][.]`	fr$z$, fr$x$, fr$y$	fr$z$ = fr$x$ / fr$y$
`fcmpo`	cr$z$, fr$x$, fr$y$	cmp, any NaN a NaN
`fcmpu`	cr$z$, fr$x$, fr$y$	cmp, only signalling NaN a NaN

22. Simple DAXPY in PPC Assembly

```
                  r3      f1:r4/r5         r6        r7         r8        r9
void ATL_UAXPY(int N, double alpha, double *X, int incX, double *Y, int incY)
```

Moving Ptrs:

```
#define Mjoin(pre, nam) my_join(pre, nam)
#define my_join(pre, nam) pre ## nam
#define N       r3
#define X       r6
#define Y       r8
#define alpha   f1
#define rX      f2
#define rY      f3
    .text
    .globl Mjoin(_,ATL_UAXPY)
Mjoin(_,ATL_UAXPY):
LOOP:
    lfd     rX, 0(X)
    lfd     rY, 0(Y)
    fmadd   rY, alpha, rX, rY
    stfd    rY, 0(Y)
    addi    X, X, 8
    addi    Y, Y, 8
    addic.  N, N, -1
    bne+    LOOP
DONE:
    blr
```

Indexed:

```
Mjoin(_,ATL_UAXPY):
    sldi    II, N, 3
    add     X, X, II
    add     Y, Y, II
    neg     II, II
LOOP:
    lfdx    rX, X, II
    lfdx    rY, Y, II
    fmadd   rY, alpha, rX, rY
    stfdx   rY, Y, II
    addic.  II, II, 8
    bne+    LOOP
DONE:
    blr
```

23. Simple SASUM

```
            /*                    r3       r4              r5
            float ATL_UASUM(int N, float *X, const int incX) */

#define Mjoin(pre, nam) my_join(pre, nam)
#define my_join(pre, nam) pre ## nam
#define II      r3
#define X       r4
#define rsum    f1
#define rX      f2
        .text
        .globl Mjoin(_,ATL_UASUM)
Mjoin(_,ATL_UASUM):
        xor     r5, r5, r5
        stw     r5, -4(r1)
        lfs     rsum, -4(r1)
        sldi    II, II, 2
        add     X, X, II
        neg     II, II
LOOP:
        lfsx    rX, X, II
        fabs    rX, rX
        fadds   rsum, rsum, rX
        addic.  II, II, 4
        bne+    LOOP
DONE:
        blr
```

```
#define Mjoin(pre, nam) my_join(pre, nam)
#define my_join(pre, nam) pre ## nam
#define N       r3
#define X       r4
#define rsum    f1
#define rX      f2
        .text
        .globl Mjoin(_,ATL_UASUM)
Mjoin(_,ATL_UASUM):
        xor     r5, r5, r5
        stw     r5, -4(r1)
        lfs     rsum, -4(r1)
        mtctr   N
LOOP:
        lfs     rX, 0(X)
        fabs    rX, rX
        fadds   rsum, rsum, rX
        addi    X, X, 4
        bdnz+   LOOP
DONE:
        blr
```