IKSwift

Design of an Inverse Kinematics Accelerator

Yipeng Huang, Lianne Lairmore, Richard Townsend
{yipeng, lairmore, rtownsend}@cs.columbia.edu

March 26, 2014

1 Overview

In this project, we will build specialized hardware to tackle the inverse kinematics problem.

Inverse kinematics is widely used in robotics computing and in computer graphics. The problem takes as input a configuration of mechanical joints, which can be rotational or sliding, that are present in an arm or a leg. Then, it takes as input the limb's current shape and a target shape, solving for the required joint motions to get to the desired shape.

We will build a configurable solver on an FPGA, in hopes of speeding up solutions when compared to running the same algorithm on a regular CPU.

2 Motivation

Computers are increasingly embedded in the real world, often permanently attached to sensors and actuators. One example of these systems are robots. Computer systems that must coordinate with sensors and actuators are distinct from general purpose computers, and the desired hardware to support its applications will also be different.

Specialty hardware such as GPUs can support hard-hitting *sensing* algorithms, especially those that involve image processing. Equally important, but less well studied, is how computer hardware should adapt to support *controlling* actuators. Actuator algorithms have yet to appear in well known computer architecture research workloads. Early studies of computer architecture support for robotics show that general purpose CPUs suffer when running typical robotic workloads [1].

Problems that arise when controlling actuators, such as kinematics, dynamics, obstacle avoidance, and collision detection, have been found to occupy a large portion of computer

runtime in robotics. Live measurements show 33-66 percent dedicated to embedded computer on robot. In particular, the inverse kinematic problem is interesting because it has features of two distinct workload categories: sparse matrix math and graph traversal [2]. This project will focus on building hardware for inverse kinematics.

3 Inverse Kinematics Algorithm

We first present an algorithm that takes as input n homogenous transformation matrices T_i^{i-1} for $i = 1$ to n, and the current position of our end effector in three-dimensional space, which is represented as the vector s. The output of the algorithm is the Jacobian matrix for the current system configuration. Given matrices A and B, we use the notation AB for matrix multiplaction and $A \times B$ for the cross-product in the following pseudocode.

Let J be a 6 x n matrix, where each column corresponds to a joint. The top three scalars of a column represent the position of the joint in 3-space, while the bottom three represent the oritentation of the joint in 3-space.
Let z be the z-axis of the coordinate frame at the base of our robot appendage
for $i = 1$ *to* n **do**
    Let R_i be the 3x3 rotation block derived from T_i^0
    Let $v_i = zR_i$ be the axis of rotation or translation for joint i
    Let p_i be a column vector composed of the top three scalars in the last column of T_i^0. This is the current position of joint i
    if *joint i is rotational* **then**
        Set column i of J to be $\begin{bmatrix}(v_i \times (s - p_i)) & v_i\end{bmatrix}^T$
    else
        Set column i of J to be $\begin{bmatrix}v_i & 0 & 0 & 0\end{bmatrix}^T$
    end
end
Return J

Algorithm 1: Jacobian($T_1^0, T_2^1, \ldots, T_n^{n-1}, s$)

The following algorithm describes the Jacobian Damped Least-Squares method of solving the inverse kinematics problem. As input we are given a set of D-H parameters that fully define the initial positions of the joints of our robot appendage, as well as a three-dimensional target position for our end effector. Formally, for each joint $i = 1 \ldots n$ we have D-H parameters $\theta_i, d_i, a_i, \alpha_i$, and we call our target vector t. The final output of our algorithm is a set of updated D-H parameters that fully define the required position of our joints such that the end effector position is sufficiently close to the target. Although the algorithm given applies

to a general $n-$jointed robot, we will only consider robots with 6 joints in our system, leading to a 6x6 Jacobian matrix.

Let ϵ be the desired accuracy of our final results;
for $i = 1$ *to* n **do**
  Calculate the homogenous transformation matrix T_i^{i-1} for joint i using the given D-H parameters;
end
Let $T_n^0 = \prod_{i=1}^{n} T_i^{i-1}$ be the full homogenous transformation matrix for the system;
Let s be a column vector composed of the top three scalars in the last column of T_n^0. This is the current position of our end effector;
Let $e = t - s$ be the desired change in the position of our end effector;
Set $e = \begin{bmatrix} e & 0 & 0 & 0 \end{bmatrix}^T$ so we can use it in our Jacobian equations, which deal with 6 x 6 matrices;
Let $J = JACOBIAN(T_1^0, T_2^1, \ldots, T_n^{n-1}, s)$ be the Jacobian matrix for the current system configuration;
while *the* l^2 *norm of* e *is greater than* ϵ **do**
  Let J^T be the transpose of J;
  Let λ be a small positive constant;
  Let I be the identity matrix;
  Use row operations to determine the vector f that satisfies the equation $(JJ^T + \lambda^2 I)f = e$;
  Let $\Delta\theta = J^T f$ be a vector whose ith component is a change in joint i's angle. Note that if joint i is translational along the unit vector v_i, then the joint "angle" measures the distance moved in the direction v_i and the ith component in $\Delta\theta$ will be a change in d_i;
  for $i = 1$ *to* n **do**
    if *joint* i *is translational* **then**
      Set $d_i = d_i + \Delta\theta[i]$;
    else
      Set $\theta_i = \theta_i + \Delta\theta[i]$;
    end
    Recalculate T_i^{i-1};
  end
  Recalculate T_n^0, s, e, and J using our updated homogenous transformation matrices;
end
Return the final set of D-H parameters currently specifying the positions of our joints;

Algorithm 2: JacobianTranspose($\{\theta_1, d_1, a_1, \alpha_1\}, \ldots, \{\theta_n, d_n, a_n, \alpha_n\}, t$)

4 Software Prototype

Our software prototyping has two goals. The first goal is to verify we understand and can translate the algorithm to hardware. The second purpose of our prototype is a way of verifying results from our hardware. We found an open source C++ project on github.com which computes the incremental angle movements for a given robot to reach a target position given a beginning position using three different algorithms [3]. The algorithms being used in the open source project were all ones we had been exploring to implement in hardware. The code was organized and very clean. It was perfect for us to grasp how the Jacobian Transpose, Jacobian Pseudo Inverse, and Damped Least Squares algorithms worked in practice since our only source had been higher level papers. This project also covers our second need, for a way to verify results from our hardware. The fact that the project wasn't implemented by us gives it credibility; it would be easy to implement the algorithm incorrectly in software and then in hardware and not realize the original software was wrong.

Along with using this open source project we are implementing a smaller prototype that will reflect the structures in our hardware and will only use the one algorithm we plan to use. By creating this second software project we will be able to play around with structures we might need to change. For example, it would be easier to edit a project created by ourselves if we wanted to try and see how fixed point or integer math would work instead of floating point. Another example of ways we might use the self implemented project in testing is by testing different computational methods for sin and cos. It is important to test how the accuracy of our algorithm changes if we change computational methods like the ones mentioned above. Editing will be a lot easier to do in software than hardware and less time consuming. It would be pointless wasting time designing hardware that doesn't compute the algorithm accurately enough.

After our software prototype has been implemented along with hardware we will have 3 different programs attempting to compute the same information. This will give us confidence in our end result if all three methods match. As an extra measure we are planning, for at least our software, to connect to software that will draw a robot arm to verify visually that it works.

Since parsing XML documents in software isn't really an important part of learning how to write embedded systems we have decided to use part of the open source C++ project to parse the XML robot configuration files to retrieve the original joint positions and the joint types. The software from the cpp-inverse-kinematics-library will parse a given XML file then give the resulting joint information to sofware we have created. Our software will then obtain a target position from the user and with the target information and the joint configurations it will start computing the next angle positions using the FPGA hardware component we will build.

5 Architecture

For our project we will be using the FPGA as an accelerator for inverse kinematics computations. Software running on the ARM processor on the SoCKit board will be driving the FPGA and will display its outputs on a monitor. The user should be allowed to spec-

ify a robot design via an XML file, which contains the Denavit-Hartenberg parameters for the robot. The software will supply target Cartesian coordinates to the accelerator, which will return updated joint configurations to move towards the given target coordinates. The software will use the joint configuration to update an image on the monitor. The resulting image should result in an animation of a appendage moving towards a target position.

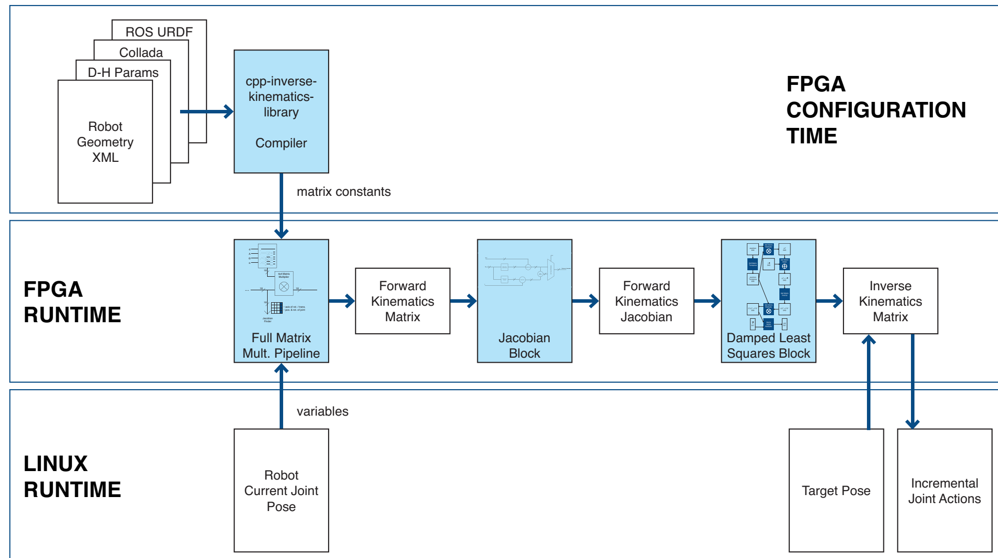

Figure 1: An architecture view of the software and hardware tools we will use for this design.

6 Hardware/Software Interface

We will design a "joint" peripheral through which the software and hardware components of our system will interact. The driver for the joint peripheral will provide an ioctl that copies a struct to and from the user with the following components:

Field	Comments
`unsigned char target[3]`	The x,y,z, coordinates of the target position for our end-effector
`unsigned char joint`	0,1,...,JOINT_DOF-1
`unsigned char joint_type`	The ith bit is 1 if the ith joint is rotational, 0 for translational
`unsigned char parameter`	THETA,L_OFFSET,L_DISTANCE,ALPHA
`unsigned short magnitude`	

The `joint` field represents which joint we're referring to (we have a total of `JOINT_DOF` joints), the `joint_type` field keeps track of the type of every joint in system, the `parameter` field is a constant signifying which D-H parameter we're reading or writing, and the magnitude field holds the value of that D-H parameter.

The registers used by the device driver are represented with the following struct, following the example given in lab 3:

```
struct joint_dev{
struct resource res; /* Resource: our registers */
void __iomem *virtbase; /* Pointer to registers */
u8 joint_type; /* ith bit is 1 if ith joint is rotational, 0 for translational */
u16 target[3]; /* Target position */
u16 dh_params[JOINT_DOF * 4] /* Every joint has 4 parameters */
} dev;
```

The `joint_type` and `target` fields mirror the same fields in the user-level struct and are only set once during the configuration stage of our pipeline. The `dh_params` array in the given struct will be used to read and write magnitudes for various D-H parameters in the given configuration. For example, the magnitude of the third joint's theta parameter would be stored at `dh_params[2*4 + THETA]`. Note that since the only parameters that our algorithm modifies are θ_i and d_i for joint i, these are the only components of our struct that will be modified after the start of our algorithm.

7 Hardware

We will represent numbers as 16-bit fixed point numbers in our hardware.

We are limited in the number of digital signal processors and lookup tables available in the FPGA. We pay attention to reusing units that use a lot of area, and time multiplex their use so they are used multiple times in the algorithm.

In the following subsections, we describe the submodules of the accelerator. Then, we describe the custom functional units we have to build in order to assemble our submodules. We assemble these custom functional units using IP designs generated by Altera MegaFunctions.

7.1 D-H Parameter Homogeneous Transformation Block

Frames are a set of axes and coordinates that describe 3D space. Frames can be global or local. A local frame would be useful in describing the x, y, z positions of an object in space, along with the orientation (direction it is pointing) in space. Each link in a robot appendage has a frame associated with it.

A moving joint that connects two links results in a change in reference frames between the two links preceding and following the joint. We can transform from one frame to the next using homogeneous transforms, which are described as 4 by 4 matrices. For background information on homogeneous transforms refer to [4].

If we use the standard D-H parameters to describe the joint, this change in frames is a homogeneous transformation shown in Figure 2, which when multiplied out in full is the matrix shown in Figure 3.

$$T_i^{i-1} = Rot(Z,\theta_i)\, Trans(Z,d_i)\, Trans(X,a_i)\, Rot(X,\alpha_i)$$

Figure 2: The transformation between two frames linked by a joint, using D-H parameter variables.

Recall that revolute joints are represented as a rotation of joint angle α about the Z axis, and prismatic joints are translations by link offset a along the Z axis. To align the two coordinates frames, we translate along the X axis by the link length d, and rotate by the X axis by the link twist θ.

$$\begin{bmatrix} C\theta_i & -S\theta_i C\alpha_i & S\theta_i S\alpha_i & a_i C\theta_i \\ S\theta_i & C\theta_i C\alpha_i & -C\theta_i S\alpha_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3: The full matrix describing the transformation between two frames linked by a joint.

We can calculate this homogenous transformation using a dedicated hardware block in the FPGA. Figure 4 is the dataflow diagram for a hardware block that calculates all the elements in the transformation matrix. This submodule uses two instances of the sine cosine functional unit, which we describe later.

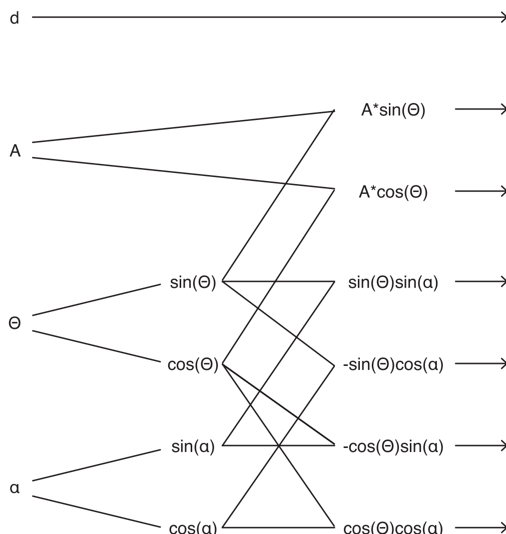

Figure 4: The dataflow diagram for a hardware block that calculates the elements in a homogenous transform matrix.

7.2 Full Matrix Multiplication Pipeline

A series of joints connected together by links in an appendage form a kinematic chain. To solve the inverse kinematics problem, we must first have the forward kinematic description of the robot. We can describe the location and orientation of the end of the appendage with yet another 4 by matrix. This full forward kinematics matrix is simply the product of the matrices that describe each joint. We can calculate this in hardware using the pipeline shown in Figure 5.

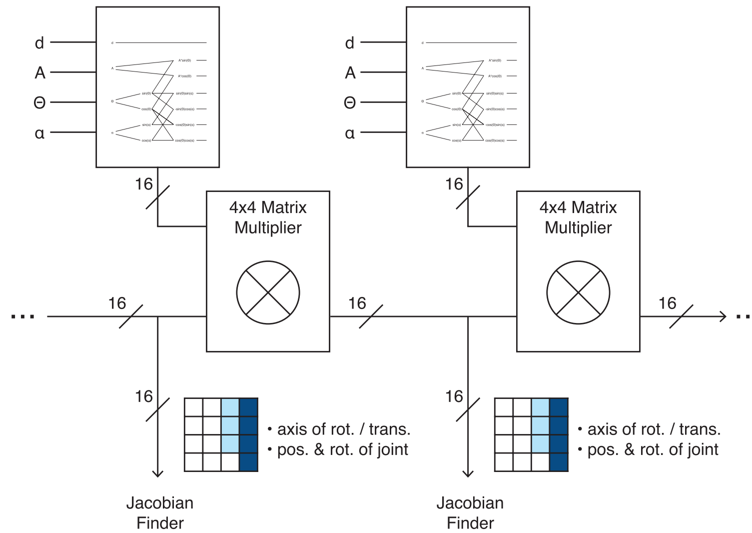

Figure 5: The homogenous transform matrix blocks are chained together to calculate the full transformation matrix of the forward kinematic chain.

7.3 Jacobian Block

The matrix that relates the differential motion of joints to differential motion in cartesian space is called the Jacobian matrix. This matrix describes the velocity relationship between joints and the end of the actuator [5].

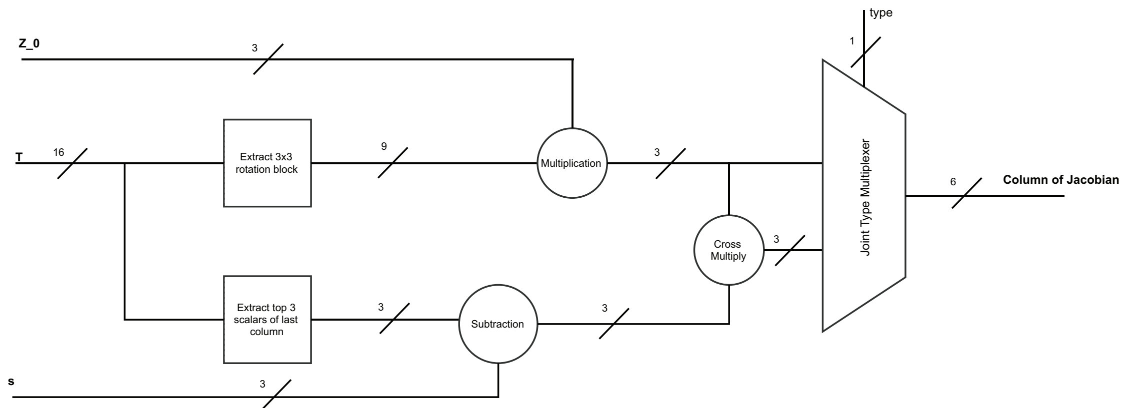

Figure 6: The dataflow diagram for a hardware block that calculates the ith column of the Jacobian matrix.

7.4 Damped Least Squares Block

The Jacobian matrix describes the velocity of the manipulator end point as a function of joint velocities. The inverse kinematics problem is solved if we can find the matrix inverse of the Jacobian matrix, which would describe the requisite joint velocities to obtain the desired velocity of the manipulator.

Finding the inverse Jacobian matrix is not possible in practice. A robot that has fewer than six degrees of freedom (six joints) would not have full control of translation and orientation of its hand, resulting in a non-square, and therefore non-invertible, Jacobian matrix. A robot that is at extreme points in its range of motion may also have a Jacobian matrix that does not have full row rank, and therefore have no inverse Jacobian matrix.

Instead, we will find the Jacobian matrix inverse in the least squared sense by solving the normalized matrix equation. We do this by multiplying both sides of the matrix equation with the Jacobian matrix transpose.

Furthermore, a square matrix may not be invertible when two or more rows cancel out, leading to a matrix that does not have full row rank. Running any matrix inversion algorithm on such a matrix would result in a divide by zero exception. We eliminate this possibility by adding a small bias constant along the diagonal of the Jacobian matrix, preventing the matrix from losing a row.

The pipeline for the damped least squares inverse kinematics algorithm is shown in Figure 7.

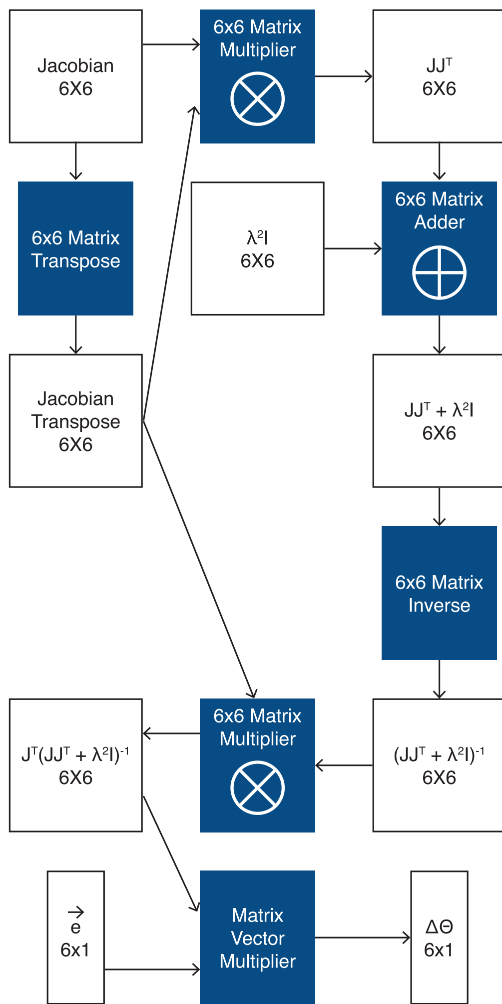

Figure 7: The pipeline for calculating joint movements using the damped least squares algorithm.

7.5 Custom Functional Units

In this section we describe the building blocks of the accelerator—our custom functional units—which we assemble to create the submodules described in previous sections. These custom functional units are assembled from IP designs generated by Altera MegaFunctions.

7.5.1 16-bit Sine and Cosine

We use Taylor series to estimate sine and cosine functions. The Taylor series expansion for sine is $sin(x) = x - x^3/3! + x^5/5!...$ Similarly, the Taylor series expansion for cosine is $cos(x) = 1 - x^2/2! + x^4/4!...$ Figures 8, 9 show the effect of including more terms on accuracy.

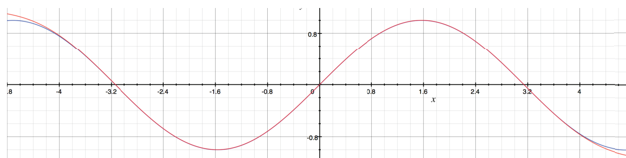

Figure 8: The Taylor series estimate of sine becomes more accurate as more terms are added. Blue is the ideal function. Red is the six term estimate.

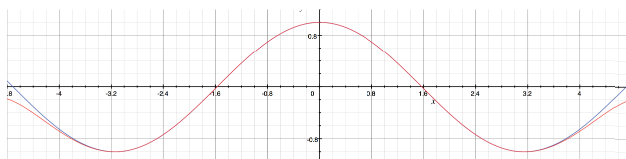

Figure 9: The Taylor series estimate of cosine becomes more accurate as more terms are added. Blue is the ideal function. Red is the six term estimate.

For our design, we will use just six terms to estimate sine and cosine. This level of accuracy is acceptable because the robot joint angles are constrained to positive and negative one radian. Low order estimates for sine and cosine suffice for function values near the origin.

Such a functional unit would require 20 multipliers and would finish in 20 cycles. Specialized multipliers that square a single variable or multiply a variable by a constant coefficient cost less than a regular multiplier, so we use those where possible to trim area costs.

7.5.2 4x4 Matrix-Matrix Multiplication

The full transformation matrix pipeline needs to multiply D-H transformation blocks. Instead of instantiating a costly, dedicated 4x4 matrix multiplication functional unit, we use the 6x6 matrix multiplication functional unit needed for the damped least squared algorithm, which would otherwise be idle while the accelerator is finding the Jacobian matrix.

When using the 6x6 matrix multiplier for multiplying 4x4 matrices, the additional pair of rows and columns that pad the 4x4 matrices will be zero.

7.5.3 6x6 Matrix-Matrix Multiplication

Matrix-matrix multiplication is highly parallel—multiplying 6x6 matrices requires 6^3 multiplications that may occur in parallel. We cannot instantiate 216 multipliers on the FPGA, so instead we will do 36 multiplies or 72 multiplies at once.

Strassen's algorithm could further reduce the multiplications needed for 6x6 matrix multiplication.

7.6 Submodule Timing Design

The FPGA has a limited number of DSPs which are used to implement multipliers, so we must time multiplex their use. We schedule the use of our functional units so that parts of the inverse kinematics algorithm can run in parallel, and so that functional units can be reused in different parts of the algorithm. By minimizing area costs, we may be able increase the precision of our numbers from 16-bit to 24-bit or even 32-bit. Larger number representations could improve convergence of the algorithm, and make the accelerator easier to use.

Figure 10 shows the timing design of the accelerator. The diagram lists the functional unit hardware resources along the vertical axis, and displays which clock cycles those resources are active along the horizontal axis. The top half of the diagram shows the first 150 clock cycles of the algorithm, which is dedicated to finding the forward kinematics Jacobian matrix. The bottom half of the diagram is the second 150 clock cycles of the algorithm, which carries out the damped least squares algorithm.

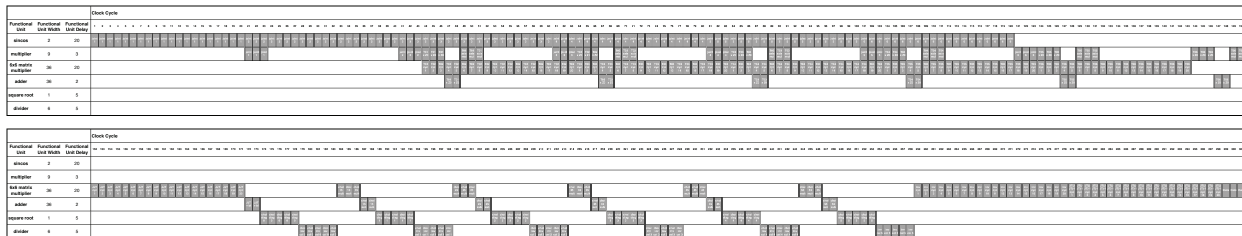

Figure 10: The timing design of the accelerator. The diagram lists the functional unit hardware resources along the vertical axis, and displays which clock cycles those resources are active along the horizontal axis. In each cell we provide a brief note on which part of the algorithm occurs in that cycle.

While it may appear that there is minimal parallelism in the damped least squares algorithm, we point out that the matrix multiplier, divider, and adder units are all parallel, SIMD-style functional units. The matrix multiplier in particular does 36 parallel multiplications at once.

Once the design is implemented, we will find the longest register-to-register critical path, and pipeline those stages to increase operating frequency. We suspect these paths would occur in the divider or square root IP designs. We will also decrease the pipeline depth of non-critical paths, possibly within the multiplier units, to decrease latency. We could also further optimize the design by further increasing the degree of parallelism in the matrix multiplier.

8 Milestones

Completed Milestones

1. Design block diagram of our system
2. Find C code that implements various inverse kinematics algorithms
3. Determine how to represent input and output with respect to the user (textual input, graphical output)

Milestone 1

1. Write our own implementation of the damped least-squares algorithm in C
2. Design top-level module describing the interface between the hardware and software sections of our system
3. Design our joint peripheral device driver
4. Determine how best to decrease the number of DSP blocks our system uses in the FPGA

Milestone 2

1. Associate the different blocks in the diagram of our system with corresponding sections of C code in our implementation
2. Construct timing diagrams for each of our submodules
3. Begin coding the submodules of our system in SystemVerilog

Milestone 3

1. Full implementation of our system
2. Develop testbenches for the different modules in our system

Final Project Presentation

1. Finish testing our system both in simulation and on the FPGA
2. Write up our final report and prepare our final presentation

9 Appendix: FPGA Utilization Estimate

In order to estimate the area and timing costs of the accelerator, we created a table enumerating the costs of each of the custom functional units and submodules we proposed, in terms of the pipeline depth of the block, and the DSP, lookup table, and register costs of the block. These figures are based on estimates given by the Altera MegaFunction User Manual and by generating the IP designs in Quartus MegaWizard. These estimates are shown in Figure 11.

References

[1] S. Caselli, E. Faldella, and F. Zanichelli, “Performance evaluation of processor architectures for robotics,” in *CompEuro ’91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pp. 667–671, May 1991.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, “A view of the parallel computing landscape,” *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[3] kirillv@github.com, “cpp-inverse-kinematics-library,”

[4] “Introduction to homogeneous transformations & robot kinematics,”

[5] H. H. Asada, “Introduction to robotics chapter 5 differential motion,”

	count	precision	delay	DSP	% DSP Use	ALM	% ALM Use	Regs	18 x 18 Mult
Functional Units									
LPM_ADD_SUB adder subtractor	1	20-bit	2	0	0%	11	0%	0	0
LPM_MULT square multiplier	1	16-bit	3	0	0%	146	0%	151	0
LPM_MULT variable multiplier	1	16-bit	3	2	2%	294	1%	274	2
LPM_DIVIDE divider	1	30-bit	5	0	0%	642	2%	0	0
ALTSQRT square root	1	20-bit	5	0	0%	94	0%	0	0
sincos									
LPM_MULT square multiplier	3	16-bit		0	0%	438	1%	453	0
LPM_MULT variable multiplier	7	16-bit		14	13%	2058	5%	1918	14
coefficient multiplier	10	16-bit		20	18%	2940	7%	2740	20
subtotal	1	16-bit	20	34	30%	5436	13%	5111	34
4x4 matrix multiplier									
LPM_MULT variable multiplier	16	16-bit		32	29%	4704	11%	4384	32
LPM_ADD_SUB adder subtractor	16	20-bit		0	0%	176	0%	0	0
subtotal	1	16-bit	14	32	29%	4880	12%	4384	32
3x1 vector vector cross product									
LPM_MULT variable multiplier	6	16-bit		12	11%	1764	4%	1644	12
LPM_ADD_SUB adder subtractor	3	20-bit		0	0%	33	0%	0	0
subtotal	1	16-bit	5	12	11%	1797	4%	1644	12
6x6 6x1 matrix vector multiplier									
LPM_MULT variable multiplier	6	16-bit		12	11%	1764	4%	1644	12
LPM_ADD_SUB adder subtractor	6	20-bit		0	0%	66	0%	0	0
subtotal	1	16-bit	20	12	11%	1830	4%	1644	12
6x6 matrix multiplier									
LPM_MULT variable multiplier	36	16-bit		72	64%	10584	25%	9864	72
LPM_ADD_SUB adder subtractor	36	20-bit		0	0%	396	1%	0	0
subtotal	1	16-bit	20	72	64%	10980	26%	9864	72
6x6 cholesky decomposition									
ALTSQRT square root	1	20-bit		0	0%	94	0%	0	0
LPM_DIVIDE divider	5	30-bit		0	0%	3210	8%	0	0
LPM_MULT variable multiplier	25	16-bit		50	45%	7350	18%	6850	50
LPM_ADD_SUB adder subtractor	25	20-bit		0	0%	275	1%	0	0
subtotal	1	16-bit	80	50	45%	10929	26%	6850	50
6x6 lower triangular matrix inversion									
LPM_DIVIDE divider	6	30-bit		0	0%	3852	9%	0	0
LPM_MULT variable multiplier	9	16-bit		18	16%	2646	6%	2466	18
LPM_ADD_SUB adder subtractor	6	20-bit		0	0%	66	0%	0	0
subtotal	1	16-bit	50	18	16%	6564	16%	2466	18
D-H Transformation Block									
sincos	2	16-bit		68	61%	10872	26%	10222	68
LPM_MULT variable multiplier	6	16-bit		12	11%	1764	4%	1644	12
subtotal			23	80	71%	12636	30%	11866	80
6 Degree of Freedom Full Matrix Block									
D-H Transformation Block	1	16-bit		68	61%	12636	30%	11866	80
4x4 matrix multiplier	1	16-bit		32	29%	4880	12%	4384	32
subtotal			152	32	29%	17516	42%	16250	112
Jacobian Block									
LPM_ADD_SUB adder subtractor	3	20-bit		0	0%	33	0%	0	0
3x1 vector vector cross product	1	16-bit		12	11%	1797	4%	1644	12
subtotal	1	16-bit	7	12	11%	1830	4%	1644	12
Damped Least Squares Block									
6x6 matrix multiplier	1	16-bit		72	64%	10980	26%	9864	72
LPM_ADD_SUB adder subtractor	6	20-bit		0	0%	66	0%	0	0
6x6 cholesky decomposition	1	16-bit		50	45%	10929	26%	6850	50
6x6 lower triangular matrix inversion	1	16-bit		18	16%	6564	16%	2466	18
6x6 6x1 matrix vector multiplier	1	16-bit		12	11%	1830	4%	1644	12
subtotal	1	16-bit	212	152	136%	30369	73%	20824	152
GRAND TOTAL									
				196	175%	49715	120%	38718	276
FGPA RESOURCES									
Cyclone V SX C6 (5CSXFC6D6F31)				112	100%	41509	100%	166036	224

Figure 11: An estimate of area and timing costs of the accelerator submodules and custom functional units.